Mining Driving Preferences and Efficient Personalized Routing

Joachim Klokkervoll, Samuel Nygaard, Mike Pedersen,
Lynge Poulsgaard, Philip Sørensen and Henrik Ullerichs

{jklokk12, snpe12, mipede12, lkpo12, ppsa12, huller15}@student.aau.dk

January 5, 2017

Abstract

Current navigation systems only support limited personalized settings and typically only provide the shortest or fastest route to a target without considering other features in the road network. Proposing routes based on personal preferences, such as avoiding traffic lights or preferring tunnels over bridges, is desirable. These preferences towards different features can be determined from previous driving behavior, instead of being explicitly specified by each driver. We propose a general framework for personalized routing optimized for query speed. The personalization is achieved by using Customizable Contraction Hierarchies with a personalized cost function extracted from a data set consisting of 203 million GPS records distributed across 183 drivers. The records of each driver are separated into meaningful trips and map matched to roads in OpenStreetMap to obtain accurate GPS trajectories and features for each trip. Gradient Descent in TensorFlow is used to find the preference vector for each driver, i.e. to what degree the driver favors or disfavors each feature. Customizable Contraction Hierarchies is utilized to propose personalized routes based on the personal preference vectors. Analysis of the preference vectors indicate that drivers in general have similar preferences. However, we show that further improvement is possible when using personalized routing for the individual user compared to the general preferences of all users. We also achieve efficient personalized routing, with runtime performance of up to 173 times faster than traditional shortest path algorithms. We also show that performance of computing preferences is increased by one order of magnitude when using GPU computation.

1. Introduction

Commercialized automotive navigation services often provide impersonal and static route suggestions based on an source and a target. These static suggestions are usually computed by finding the most optimal path based on cost functions provided by the service providers. However, these cost functions are often quite simplistic, e.g. shortest distance, lowest expected travel-time, or lowest fuel-consumption. The cost function is used to determine the weights of the arcs in the road network graph so that existing shortest-path algorithms can be used to solve the routing problem [16].

These simple cost functions are repetitively used for every single user alike thus ignoring individual preferences, such as time versus distance, different turn-types, road type/surface, bridges, and tunnels. The concept of incorporating individual preferences in navigation services is called personalized routing [10, 19].

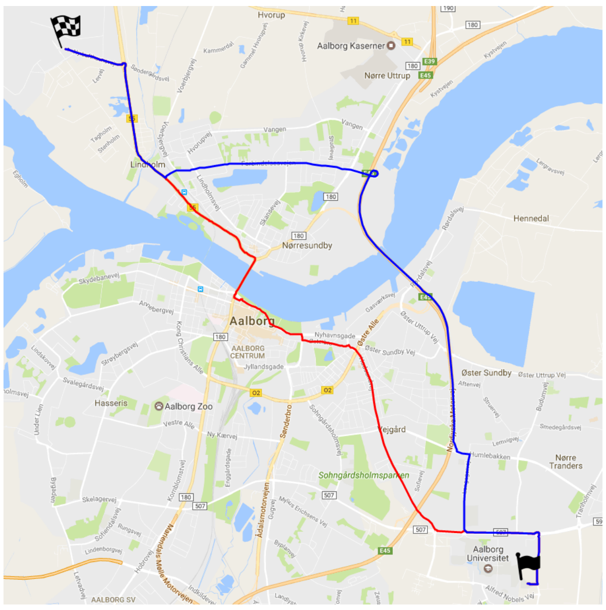

Figure 1: Two routes (West-going and North-going) having the same source and target but different features. Map provided by Google Maps.

A simple example of this can be seen in Figure 1, where two different routes (West-going and North-going) with identical source and target take different paths. The road features along each route are listed in Table 1, which summarizes the difference between the two routes. The features of the West-going route matches a person who abstain from using motorways and prefers shorter routes (at the expense of extra time), whereas the North-going route matches a

Road features	Unit	Route	
		West	North
Distance	m	13430	14844
Motorway	m	0	5054
Highway	m	12540	9624
Urban	m	890	165
Unclassified	m	0	0
Asphalt	m	13430	14844
Other paved	m	0	0
Unpaved	m	0	0
Dist. at urban speed	m	10232	6578
Dist. at rural speed	m	3198	5903
Dist. at motorway speed	m	0	2363
1 Lane	m	13070	12584
2 Lane	m	10	2260
Roundabout	m	346	255
Tunnel	m	0	738
Bridge	m	656	132
Traffic signals		25	11
Crossings		17	3
Traffic calmings		0	0
Left turns		5	3
Right turns		6	5

Table 1: Road features for the routes in Figure 1.

person who wants to get to the target faster, by using road types such as motorways, but cares less about the absolute shortest path. Other interesting feature to compare could be the distance traveled on bridges and through tunnels, where the West-going driver might really dislike tunnels and is suggested to cross the river using a bridge instead.

We only utilize the distance metric, even though the time metric w.r.t both travel time and time of day is essential for routing methods. We refrain from addressing time as a metric in this paper for simplicity as our focus is on operational framework. However, since it is a natural extension, the time metric and time-dependency is addressed in Section 10.

In our framework, the personalized cost functions are learned using meaningful trips extracted from each driver's raw GPS records that have been map matching onto OpenStreetMap (OSM) using the open-source OSRM library [20]. Preferences are thus found implicitly from historic data rather than each user explicitly stating them. The data set used in this paper contains 203 million GPS records collected every second from 183 vehicles over a two year period.

For using personal cost functions on the graph (updating the cost of traversal), we utilize Customizable Contraction Hierarchies (CCH) described in Section 5.2, which make it possible to add adjustments on the fly when updating the personalized cost.

Even though CCH provides a significant increase in performance, there is a huge amount of data to process. Therefore, a GPU is utilized in our experiments to achieve the best performance, as GPUs have shown to both increase performance and be more energy-efficient compared to CPUs [9].

The goal of this paper is to propose an efficient personalized routing framework, with the end-user's query time and the quality of the proposed route as the primary focus. The contributions of this paper are:

– An efficient framework, for personalized profiling and routing based on CCH and personal GPS traces.

– Mining driving preferences based on GPS data using map matching and road features.

– Showing the applicability of GPU accelerated training for such a framework.

The rest of the paper is structures as follows: The related work is explored in Section 2. Section 3 formalizes the modeling of the problems presented in this paper. The map matching method is explained in Section 4. In Section 5, effective route planning algorithms are described. Section 6 formalizes our implementation of a personalized score. Section 7 outlines the overlying framework. In Section 8 we present the different experiments we conducted. Section 9 concludes on the results of the experiments and Section 10 briefly outlines possible future problems to look at.

2. Related Works

To make an efficient personalized routing framework using GPS data, we need to look at how to efficiently calculate shortest path and how to incorporate personalization. In each of these two categories we will introduce existing work.

Efficient shortest path Bast et al. [5] focus on performance routing on continental scale, with a mix of road and public transportation networks. Their work is somewhat similar to ours, however we differ by focusing solely on road networks using the personal preferences of the user.

Geisberger et al. [16] show a solution, utilizing Contraction Hierarchies, for allowing more efficient and flexible queries, i.e. queries that contain restrictions like avoiding toll roads and overpasses lower than a specified height. The use of Contraction Hierarchies also eliminates the need to store several versions of the road network for different scenarios, e.g. one version with and one without toll roads. Geisberger et al. [16] also discuss optimization of the existing algorithms to further speed up the routing process. In this paper, we also focus on efficient route planning on road networks using Contraction Hierarchies, allowing us to quickly explore and update different cost functions. Our approach differs as we will determine the various parameters for the algorithm automatically as well as applying these methods on the problem of personalized routing. Furthermore, we use Customizable Contraction Hierarchies by Dibbelt et al. [12], an improved version of Contraction Hierarchies that allows more efficient shortest path computations.

Personalization Delling et al. [10] delve into the realm of personalized routing while utilizing GPUs, where they use a proprietary closed-source implementation of Customizable Route Planning. In this paper, we investigate an open-source implementation of the Customizable Contraction Hierarchies method. Delling et al. [10] utilize a rather simple quality score where they essentially comparing the shortest Euclidean length between each observed GPS record with a computed shortest path (from the source to the target). If, for instance, half of the GPS records is within the distance threshold the computed shortest path would get a quality score of 0.5, they utilize their framework using a threshold distance of 10 m. We differ from this paper by utilizing another personalization method to compare traces with calculated routes explained in Section 6.

Letchner et al. [19] presents a system, TRIP, able to do personalized routing based on data from the Seattle municipally. They present the first ever framework based on real data gathered from 109 cars using 8 GPS devices installed for a two-week period in each car, which is similar to the format of our data. A novel contribution at the time was the time-dependent speeds in the road network inferred from the observed GPS records. The presented solution uses a cost function based on the so-called inefficiency ratio, which is the ratio between the fastest route at a given time and the actual driven route for each route. This inefficiency ratio is applied, per driver, to all road segments that the driver has used and can be described as a measure of how much additional travel time a driver is willing to use to travel on these road segments, or how preferred they are to the driver. Letchner et al. define a cost measure using this ratio so that the cost for roads with a low ratio is also low. This cost measure is somewhat similar to our approach, however we find a specific preference value, similar to the inefficiency ratio, for each considered feature, where Letchner et al. only apply their ratio to the travel time. Another distinction is that their ratio if found using simple averaging over all observed GPS records for a single driver, whereas we utilize machine learning to find suitable parameters.

Dai et al. [8] examines big trajectory data, using Hidden Markov Models to map match their routes, much like the framework presented in this paper. They also utilize a preference vector to define a user's preferences, however they only consider three different preference features: travel time, distance, and fuel consumption, whereas our framework takes 21 road features into account.

Patel et al. [23] also examines personalized routing, however their goal is to simplify routes by using known landmarks, e.g. work location. Our contribution differs as we try to extract features from roads driven to find similar roads that can be used when routing, whereas they examine reusing often driven routes.

Balteanu et al. [4] explain how to compute shortest paths in OSM with respect to a specific preference distribution. They model these preferences as pairwise trade-offs between different cost factors, e.g. path distance vs. number of traffic signals. Using skyline routes (Pareto-optimal routes) they can derive the personal optimal path between two points in the road-network when given a personal preference gradient for the trade-off pair. Balteanu et al. only describe their proposed framework handling one trade-off pair (two cost factors) at one time as their main contribution was instead to efficiently find the skyline routes needed by the method. They also had to resort to experimenting with synthetic preference distributions as their trajectory data was anonymized. In contrast, we will be looking at incorporating multiple cost factors instead of only two. We will also be testing the real-world applicability of our framework by using personal trajectory data instead of synthetic data.

Yang et al. [30] examines context-aware personalized routing, where the context, e.g. rushing to work, of each trajectory is considered. Their work is also based on skyline routes and in addition the same data foundation that we use and they even use OSM to similarly obtain features (costs) about the road network. We differentiate by the tools we use, in our case CCH and TensorFlow to build our road network graph and model the costs.

3. Problem Modeling

In this section, we describe how the different aspects of the problem area are modeled. Firstly, we describe how the road network and its features are modeled using graph structures. Secondly, we introduce line graphs as a solution to modeling turn costs. Lastly, we describe our modeling of the GPS data and how to split it into meaningful trips.

3.1. Graph

For the domain of personal navigation systems, we define a road network as a directed graph $G = (N_G, A_G)$, containing nodes and arcs. A node $n \in N_G$ corresponds to either a shared point between two or more road segments or the end point of a road segment with a dead end. Each arc $a \in A_G$ represents a road segment and is directional. An arc also contains a set of arc features defined by the function w_G, that takes an arc as input and returns a vector representing the features of the given arc. The different features are described in Section 3.1.2.

A *route* R is defined as a sequence of consecutive arcs in G creating a path between a start and an end node.

Applying a *cost function* F to the arcs in a route gives the cost of the given route. The cost function considers the road features of the arcs along the route, e.g. road category, speed limit, number of lanes, and turns. When a cost is defined it is possible to use a traditional shortest path algorithm to find routes in the graph based on the given cost.

3.1.1. OpenStreetMap

OpenStreetMap (OSM) is an open road network of geographic data, distributed under the Open Database License (ODbL) and maintained by a open-community of mappers. OSM contains a rich database of road networks including a variety of features and data for the different parts of the road network. The road network in OSM is represented as a graph containing nodes and ways, with ways corresponding to multiple connected arcs in our definition of a graph G. OSM uses tags to store the different kinds of data that can be specified for nodes and ways. Each tag has a specified range of values, e.g. the *highway* tag used to indicate a type of road can have values such as `motorway`, `primary`, `residential`, and `service`.

3.1.2. Road features

We can use the tags[1] from OSM to enrich our graph network with features that might influence the routes chosen by different drivers. In this section, we outline the road features we have chosen to look at. There are two types of features: arc features and node features, which describe roads and points along the roads respectively. These features can then be used to define personal costs for traversing a given road segment or passing a node in the road network. The features that we consider are:

Distance: The length of an arc in the road network.

Highway: The road type on an arc. We classify the roads into four road-types: motorways, major roads (`trunk`, `primary`, `secondary`, `tertiary`), minor roads (`residential`, `living_street`, `service`, `track`), and unclassified[2].

Surface: The surface of a road. We use three classes: asphalt, otherwise paved, and unpaved. Roads without this tag is assumed to be paved with asphalt.

Max_speed: The maximum legal vehicle speed of a road segment. Not all roads have this tag, but we will look into a solution in Section 3.1.3.

Lanes: Number of lanes on a given road segment affect the traffic throughput of the road. We penalize one and two lanes (in the same direction), where one lane being default if the tag is missing.

Roundabout, *tunnel* and *bridge*: Indicates whether a road segment is part of either a roundabout, tunnel, or a bridge.

Traffic signals: Indicating that there is a traffic signal on the tagged node.

Crossing: Indicates that there is a pedestrian crossing across the road at the tagged node.

Traffic calming: Indicates that there is traffic calming on the road node designed to slow down traffic, e.g. speed bumps and chicanes.

These features can be represented as a vector, with values for each feature on the road, an example is visualized in Table 1. We model some features, e.g. the highway feature, as several values in the vector defining the length of the arc where this tag is valid, e.g. we have a distance for motorway, highway, urban, and unclassified roads.

Summing up, we use 19 different road features in addition to the two turn costs that we will describe in Section 3.1.4.

[1]A tag is formatted as: *name* referring to a `value`.

[2]The `unclassified` tag value in OSM terminology refers to the lowest level in the road network, i.e. not an unknown classification.

3.1.3. Determining speed classes

As the road features are used in the cost function it is important that each arc contains all the key features, e.g. the *max_speed*. However, the OSM data does not contain the *max_speed* for all arcs. Only 4.9 % of the 666,961 Danish roads has a *max_speed* tag with a numerical value. To get around this problem, the speed limit of the remaining roads must be determined by other means, which we will discuss in this section. Due to the sparsity of the *max_speed* tag we simplify the classification of maximum speeds by reducing the number of dimensions to the following three speed classes: *urban*, *rural*, and *motorway*.

For the 4.9 % of roads with a specified numeric maximum speed we map the numeric speed to speed classes in the following way:

$$\begin{cases} max\_speed \leq 60 & urban \\ 60 < max\_speed \leq 90 & rural \\ 90 < max\_speed & motorway \end{cases}$$

For the remaining 95.1 % of roads we have to infer the speed limit from other data in OSM. As almost all roads in OSM has a meaningful road type tag (*highway*) and because some road types usually imply the speed limit we can roughly estimate the speed limits for 31.8 % of the remaining unclassified roads. The mapping for speed class based on road type can be seen in Table 2.

Road type tag	Speed class
`motorway`	*motorway*
`trunk`	*rural*
`primary`	*rural*
`residential`	*urban*
`track`	*urban*
`living_street`	*urban*
`pedestrian`	*urban*

Table 2: Map table from specific road type tags to a speed class.

However, 68.2 % of the roads without defined maximum speeds does not belong to the road types defined in Table 2 and are, thus, still unclassified. For these remaining roads, we will use the definition of urban areas to determine whether the road is inside an urban area, i.e. classified as *urban*, or outside, i.e. classified as *rural*. To identify if a road is inside an urban area we use the polygon areas defined in OSM with one of the following *land_use* tag values: `residential`, `industrial`, or `retail`, or the polygons with a *place* tag value of either `village` or `town`. The specific method is explained in Appendix A.2 together with a map of the identified urban areas of Denmark. After this final step, all of the roads now have a speed class In the resulting graph the speed classes are transferred to the arcs in the graph resulting in 66.4 % of the arcs having the *urban* class, 33.1 % having the *rural* class, and 0.5 % having the *motorway* class.

3.1.4. Turn cost

When planning a route, road features are not the only influencing factors. Different turns should also be taken into consideration while keeping in mind that the cost of a left turn might differ from the cost of a right turn. To introduce costs on turns we use the principle of node expansion described by Anez et al. [3], where every arc in an intersection is expanded to model the possible turns in the intersection. An example of this is illustrated in Figure 2, where all the allowed turns are depicted as arcs where weights can be individually added to model the features, and thus cost, of each arc. Figure 2 also shows how the line graph can handle turn restrictions. We exclude u-turns for simplicity because they rarely occur in proposed routes.

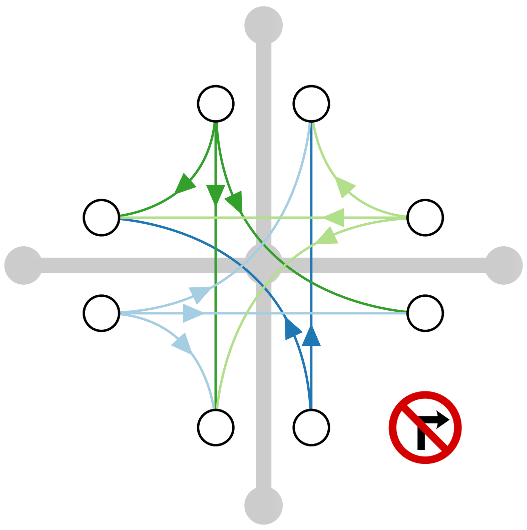

Figure 2: An intersection (thick) with a right turn restriction and with a node expansion (thin) that depicts the possible turns in the intersection.

The node expansion in Figure 2 is also known as a line graph. For simplicity, we will denote the nodes and arcs in the line graph as *line nodes* and *line arcs* respectively. Winter [28, 29] defines a line graph as:

Definition 1. The graph $D(N_D, A_D)$ is the line graph of the directed graph $G(N_G, A_G)$, where N_D is the set of line nodes in D and A_D is the set of line arcs in D.

The functions f_D, l_D, and w_D takes a line arc as input and returns the starting line node, ending line node, and the features of that line arc respectively. There exists similar functions f_G and l_G, for arcs and nodes in G.

The following properties are required for a line graph:

– For each arc $a \in A_G$ there is a line node $n \in N_D$ with $n = d(a)$, where d is a bijective function that maps arcs in G to the corresponding nodes in D and vice versa, such that $d^{-1}(n) = a$. Thus, $N_D = \{d(a_1), d(a_2), \ldots, d(a_n)\}$, for all $a_i \in A_G$.

– For each pair of consecutive arcs $(a_i, a_j) \in A_G \times A_G$, where $l_G(a_i) = f_G(a_j)$, there is an arc $\epsilon \in A_D$ between the corresponding nodes in D, $n_i = d(a_i)$ and $n_j = d(a_j)$, such that $f_D(\epsilon) = n_i$ and $l_D(\epsilon) = n_j$.

We define the feature function as $w_D : A_D \to \{x_1, x_2, \ldots, x_n\}$, where n is the number of features modeled in our road network as described in Section 3.1.2.

All arcs in the original graph G becomes nodes in the complete line graph D. Each pair of arcs in the original graph, surrounding a single node, becomes an arc in the line graph, between the nodes in the line graph corresponding to the arcs in the original graph.

Representing the road network as a line graph introduces some problems. In order to route from a start node s to a target node t in G, we refer to all line nodes $n \in N_D$ for which $f_D(d^{-1}(n)) = s$, as the start line nodes, and all line nodes $n \in N_D$ for which $l_D(d^{-1}(n)) = t$, as the target line nodes, in D. An example can be seen in Figure 3 where node 1 and 6 are the start and target node in G, $\{a\}$ is the start line node and $\{c, f\}$ are the target line nodes.

This results in our single source to single target problem now potentially becoming a many to many routing problem. The naive way to solve this problem is to calculate a route from each source to each target, however this might be very tedious to execute. Another solution is to change the domain by adding a virtual start line node, n_s, and virtual target line node, n_t, and connecting the start and target line nodes to their respective virtual line node with zero cost arcs, as seen in Figure 3. The approach using virtual line nodes changes the graph representation for each given problem, but eliminates the need to take any special considerations when using a routing algorithm.

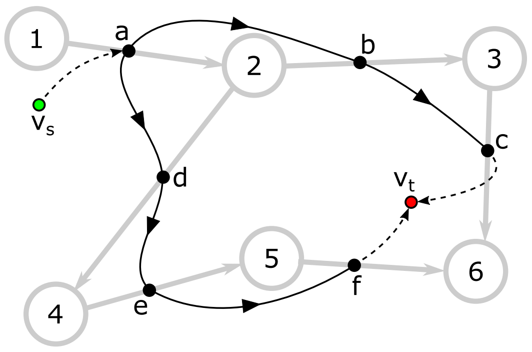

Figure 3: The graph G (gray), line graph D (black), virtual nodes v_s and v_t, and dashed zero cost arcs.

3.1.5. Graph extraction

As described in Section 3.1.1 the graph representing the road network is extracted from OSM. As OSM contains more information than just road structures, e.g. buildings, lakes, and trees, we ignore all arcs without the `highway` tag. The *highway* tag is used to signify that the arc is part of a road, street, or other path including those not suitable for motorized vehicles, such as `footways`, `bridleways`, and `cycleways`. As we are only interested in roads where cars may drive we ignore all arcs that does not allow the use of motorized vehicles, either because of its type, e.g. `footways`, or because of an access tag, e.g. *motorvehicle*=`no`. After this filtration, the road features described in Table 1 are extracted for all arcs and nodes and stored in the graph representation. Lastly, we transform the graph into a line graph as described in Section 3.1.4 to extract turn information from intersecting arcs. Turns are then determined by looking at the difference in bearing for all intersecting arcs, i.e. the angle between the road segments. At this step, we also handle turn restrictions, e.g. no right turn, and turn injunctions, e.g. only left turn, by removing turn-arcs that are not allowed. Let θ_1 and θ_2 be two bearings, we can now define the turn angle as in equation (1). A visualization of the equation can be seen in Figure 4.

$$\begin{aligned}&\text{bearingdiff}(\theta_1, \theta_2) = \\ &\qquad ((\theta_1 - \theta_2 + \pi) \bmod 2\pi + 2\pi) \bmod 2\pi - \pi \qquad (1)\end{aligned}$$

The function bearingdiff calculates the angle between two bearings (θ_1 and θ_2) in signed radians with right turns represented by negative values and left turns by positive values. In our definition, the change in direction must be above $\frac{\pi}{4}$, i.e. 45°, to be classified as a turn. We further specify that angles below $\frac{-\pi}{4}$ are right turns and angles above $\frac{\pi}{4}$ are left turns. This also means that angles falling between $\frac{-\pi}{4}$ and $\frac{\pi}{4}$ are not classified as turns. Figure 4 shows an example of outgoing arcs producing a left turn (b_1), a right turn (b_3), and no turn (b_2) for an incoming arc a.

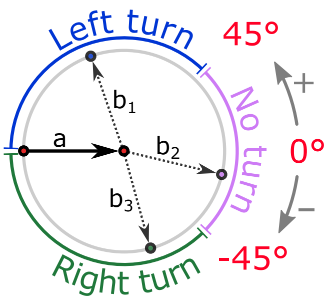

Figure 4: Examples of the different types of turns between two intersecting arcs.

The turns are represented as a new feature vector where all features have a value of 0 and only the classified turn has a value of 1 as seen in equation (2), e.g. if an ordered pair of arcs, (a, b), is classified as a right turn the second to last entry have a value of 1 and all other entries in the feature vector are 0. For each arc, a, we access its bearing from OSM by using the notation $a.\theta$. The turn vector can then be defined as:

$$\text{turn}(a, b) = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1\left[\text{bearingdiff}(a.\theta, b.\theta) < \frac{-\pi}{4}\right] \\ 1\left[\text{bearingdiff}(a.\theta, b.\theta) > \frac{\pi}{4}\right] \end{pmatrix} \qquad (2)$$

Using the method described above the original 3 million nodes and 6 million arcs from OSM are transformed into a line graph with 7.4 million turn arcs. The features of each line arc $w_D : \epsilon$ is defined as: $w_D : \epsilon \to w_G\left(d^{-1}\left(f_D(\epsilon)\right)\right) + w_G\left(l_G\left(d^{-1}\left(f_D(\epsilon)\right)\right)\right) + \text{turn}\left(d^{-1}\left(f_D(\epsilon)\right), d^{-1}\left(l_D(\epsilon)\right)\right)$, i.e. the sum of the features of the arc in the original graph corresponding to the start line node of the line arc $\left(d^{-1}\left(f_D(\epsilon)\right)\right)$, the features of the node in the original graph corresponding to the line arc $\left(d^{-1}(\epsilon)\right)$, and the turn features. Figure 5 shows two arcs and three nodes from the original OSM graph, the corresponding partial line graph, and the elements relevant for finding the features of the line arc ϵ.

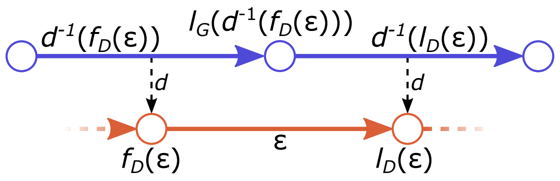

Figure 5: The graph elements influencing the final features of the line arc ϵ in the line graph (red) and the OSM graph (blue).

3.2. GPS data

A *GPS record* r is defined as: $r = (id, t, x, y, speed)$, containing a driver id, a time stamp t, a longitude x, a latitude y, and an optional current speed of the car $speed$.

We define a *trace* T_x as a sequence of GPS records ordered by the time, t, for a given driver id x: $T_x = \langle r_1, r_2, \ldots, r_n \rangle$ where $r_1.t < r_2.t < \cdots < r_n.t$ and $r_i.id = x$ for all $i \in (1, 2, 3, \ldots, n)$.

We define a *trip* as the trace of a single car ride for a single driver. When obtaining, GPS records it is usually a set of traces with no indication of the separate trips. As such we need to split the traces into trips which can be done as described in the next section.

Trip splitting

To split the traces into trips we use a simple method proposed by Froehlich and Krumm [14]: If the vehicle is stopped, i.e. has a speed of 0 or no observed GPS records, for 3 minutes the trace is split and all zero-speed points between splits are discarded. This method can be used even when the GPS records do not have measured speeds, as long as the position and time of each record is known, by using geodesic distance. A custom pseudocode function `tripsify` implementing this simple trip splitting can be seen in Code Listing 1. The full SQL code can be seen in Appendix B. A property of this method is that it can split a single trip with via points (i.e. a restroom stop on the motorway or dropping of children at the school/kindergarten) into multiple trips even though the driver might see such a trip as one continuous trip.

Code Listing 1 Custom pseudocode for `tripsify` function that splits traces into trips.

```
function tripsify(traces)
  Sort traces ascending by driver and time
  trips   := [] # array
  first   := {} # dictionary
  last    := {} # dictionary
  last.id := 0

  for each record, r in traces where speed > 0
    if r.id ≠ last.id
       or more than 3 minutes between r and last
      if first ≠ {}
        new_trip := {driver, {first.time,
             last.time}}
        append new_trip to trips
      end if
      first := r
    end if
    last := r
  end for

  if first ≠ {}
    new_trip := {driver, {first.time, last.time}}
    append new_trip to trips
  end if
  return trips
end function
```

4. Map Matching

When planning a route, we are not directly interested in the GPS coordinates, but rather the actual path taken. For this reason, we need to map the GPS traces to a road map. This is done using the Node.js implementation of OSRM, developed by Luxen and Vetter [20]. OSRM uses a sophisticated Hidden Markov Model (HMM) algorithm [24], created by Newson and Krumm [22]. The HMM is a probabilistic model that assigns probabilities to a sequence of hidden/latent variables given a sequence of input observations [18]. In the HMM created by Newson and Krumm [22] the hidden/latent variables are road segments and the input observations are GPS coordinates. The model calculates the most likely sequence of road segments that explain the input GPS coordinates, also known as *most likely explanation* inference [6]. The notion of *most likely* is based on assumptions on how people drive, and is defined as an internal cost function within OSRM. Because of its probabilistic nature, the HMM is also capable of ignoring outliers in the input data reducing the amount of data sanitation needed prior to using the model.

4.1. Search radius

The execution time of the map matching is largely dependent on the search radius parameter that defines the maximum distance from each GPS point to candidate road segments. With a higher search radius, the map matching might be more accurate but will also need to check more road segments making it slower.

When no search radius is provided, the map matching algorithm uses a search radius of 15 meters, and while being relatively fast it discards many of the GPS records in our data. To avoid losing usable measurements the search radius should be explicitly defined for each GPS record. One method of doing this is to increase the default radius, and as we show in Section 8.1.2 this gives promising results but is unfortunately quite slow. Another way is to define the search radius using the Estimated Position Error (EPE) for each point. The EPE is the maximal distance from the reported position in which the real position lies with 95 % confidence [1]. The EPE can be calculated from the Horizontal Dilution Of Precision (HDOP), which is a measure of the expected precision of a specific GPS reading. The EPE of a GPS measurement is usually outputted along with the location, but in our case this is not the case. Instead, we will estimate the EPE from the HDOP, which we do have for each record, through a simple data analysis. We make this analysis by finding the distance to the closest road for all the GPS points (274M). Using these distances as an expected EPE we can estimate the EPE for a given HDOP value. Grouping each distance by the floor of their respective HDOP produces the medians and intervals seen in Figure 6. The box around each median shows the 50 % distance interval and the whiskers shows the 90 % distance interval. The dashed lines show the 15 meter default search radius, a recently introduced OSRM default approach to this problem, and our custom fit search radius formalized in equation (3). The OSRM default approach to this problem was introduced after we made our analysis and is the result of analyzing 1 million GPS records from mobile users. All the different methods are evaluated in Section 8.1.2.

$$\begin{aligned}\text{CustomSR}_{\text{HDOP}} = \\ \max(21,\, 8.4\ln(\text{HDOP}) + 30.6)\end{aligned} \tag{3}$$

It is worth noting that the distance to the nearest road starts to drop again for HDOP values exceeding 21, which could be explained by the measurements becoming so imprecise that the nearest road no longer is the correct road but is now close to some other correct road, e.g. a road that runs parallel to the road on which the vehicle was driving.

4.2. Mapped route

In OSRM, the result of map matching a single trip is a *mapped route* which consists of a set of *matchings*. Each matching is an ordered collection of mapped GPS

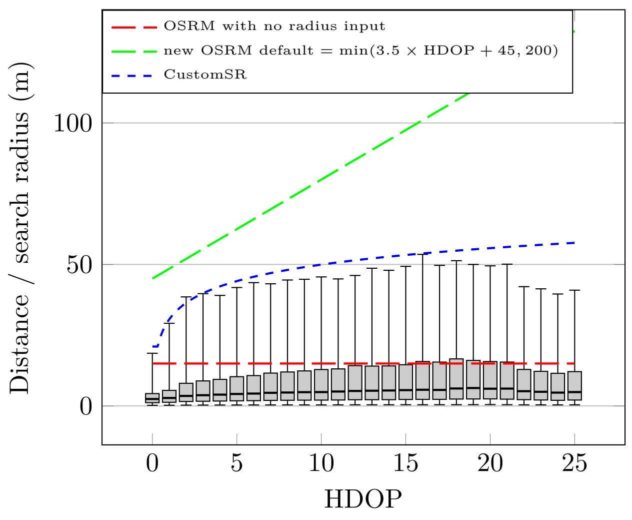

Figure 6: Distribution of the distance to nearest road for each HDOP.

points snapped to the arcs in the road network. Ideally, there is only one matching per trip, but if OSRM is unsure whether a trip is continuous or if the trip contains outliers, it splits the trip into several matchings. As the algorithm excludes these oddities, the matchings may only partially describe the actual trip and as we cannot know the reason for leaving out single points or smaller parts of the trip, it would be difficult to merge the matchings into a single matching.

As the map matching uses probabilities to find matchings, it is also able to provide a *matching confidence* for each matching corresponding to the probability of the matching being correct.

Each matching consists of a *leg* for each GPS record in the matching and contains the map matched position for its GPS record. Each leg also contains the road segment that the GPS record was matched to, as well as the calculated travel distance and time since the last leg. From the road segments of the legs in each matching we extract the OSM nodes in order of traversal as seen in Figure 7. The record/legs are numbered according to the input order and the annotation of each number is the road segment (consisting of 2 or more OSM nodes) associated with the leg. With the OSM node order we can extract the road features for each matching.

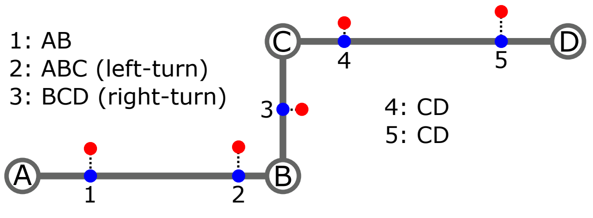

Figure 7: Example of a matching with legs (blue) of an ordered (numbered) GPS trace (red) onto a road network (gray), along with the OSM nodes of each leg.

5. Shortest Path Computation

As mentioned in Section 3, the arcs in the road network graph each have features that can be used with a cost function to define the cost of traversing them. Many algorithms for finding the shortest paths between two graph nodes already exist, such as Dijkstra [13] and many variants hereof. A dominant variant is the *bidirectional Dijkstra* where the shortest path is found using the meeting point of two simultaneous searches (forward and backwards). One shortcoming of Dijkstra's algorithm is that the query time is high for large graphs such as country or continental size graphs [26].

To optimize the query time, preprocessing steps are widely used on graphs, as shown in a survey by Bast et al. [5]. Using preprocessing imposes a high initial cost, but lowers query time on following graph queries. However, it is usually not possible to change the cost function without another heavy preprocessing step, which makes it unsuitable for personalized routing.

A compromise can be found in *customizable* preprocessing. According to Dibbelt et al. [12] there are currently two widely used methods: Customizable Route Planning by Delling et al. [11] and Customizable Contraction Hierarchies by Dibbelt et al. [12]. Both methods will be described briefly in this section. Both use a graph $G(N, A)$, where N is a set of nodes and A is a set of arcs connecting nodes in N, and some cost function c taking arcs as input, returning the cost to traverse that arc.

Both methods work in three phases in the following order:

1. Preprocessing: Slow process working on the graph topology. Only needs to be computed once.
2. Customization: Faster process which can be used to customize weights of the graph, i.e. the cost of traversing the arcs.
3. Querying: Finds shortest path between nodes. Very fast due to the preprocessing and customization.

5.1. Customizable Route Planning

The following section about Customizable Route Planning (CRP) is based on the paper by Delling et al. [11]. CRP is divided into three phases:

Preprocessing creates multiple levels of nested partitions exploiting the topological structure of the road network by minimizing the number of connections (arcs) between each partitioning. This phase is by far the most expensive of the phases, but can be computed without knowing the cost function as it only relies on the topology.

Customization creates a shortcut overlay for the partitions using a cost function. The shortcut overlay consists of shortest path computations between inter-partition arcs. This overlay must be recomputed whenever the cost function is changed, but is not very computationally expensive.

Querying uses bidirectional Dijkstra using the precomputed shortcuts to discard graph partitions from the search. By doing so, the typically large search space of Dijkstra is minimized.

5.2. Customizable Contraction Hierarchies

The following section about Customizable Contraction Hierarchies (CCH) is based on the paper by Dibbelt et al. [12]. CCH is based on and extends the original Contraction Hierarchy method, proposed by Geisberger et al. [15], with a customization phase. CCH, just like CRP, consists of three phases:

Preprocessing exploits the graphs unweighted topology and augments the graph, G, with additional shortcut arcs, A', connecting adjacent cliques of nodes in G. The cliques are found by using a *contraction graph* data structure to keep track of nodes that have been contracted (contained in virtual *supervertices*), and allows for easy neighborhood enumeration. Each node is iteratively contracted, turning it into a supervertex, and merged with neighboring supervertices. When all nodes have been contracted, the supervertices can themselves be contracted to obtain the shortcut arcs, A', and the augmented graph, G', which is the result of preprocessing.

Customization makes use of a priority queue containing the arcs that should be updated. The priority queue is denoted as $U = \{(x_i, y_i, n_i)\}$, where x_i is the head node and y_i is the tail node of an arc and n_i is the new weight. A change in the weight of an arc is propagated throughout the graph.

The performance of contraction hierarchies is highly dependent on the order in which the nodes in the graph are processed and thus benefits from being ordered in some optimal way before the customization to avoid unnecessarily expensive computations. Computing an optimal ordering is in general NP-hard, but several heuristics for real-world road network graphs perform fairly well [15]. As the order must be computed before the customization phase, it has to be metric-independent, i.e. it cannot know the weights of the graph but only the graph structure. This again makes the problem NP-hard, but like before, some heuristics have been shown to work well for real-world problems [12].

We use nested dissection with inertial flow to compute the ordering. This algorithm sorts nodes geographically, computes the maximum flow between the lower to upper half, partitions the graph by the maximum flow, and then recurses on the partitions [25]. We use the implementation in the C++ RoutingKit library [17], which provides reasonably good partitions while being very performant, according to Schild and Sommer [25]. The order is then used to optimize the CCH customization phase.

Querying uses bidirectional Dijkstra using the new graph, G', as the search space. The search in both directions can stop at any point if the current minimum path of the search exceeds the current shortest path found [17].

The RoutingKit library provides a general CCH implementation [17]. It covers all three phases of CCH, includes a node ordering algorithm as described above, and offers calculation of shortest path queries. Another feature in RoutingKit, is the support of many-to-many routing, which alleviates the potential many-to-many routing problem described in Section 3.1.4.

6. Personalization

There exist several methods for determining an individual's driving preferences and driving patterns, as described in Section 2. We model the problem as an optimization problem and use gradient descent to find a minimum. In the following sections, we will go through the details of this method.

6.1. Linear travel costs and feature vectors

To determine the shortest path from a source to a target, it is necessary to define some cost function that determines the cost of each arc. We describe each road segment by a *feature vector* that describes the different features of an arc, as described in Section 3.1.2. The cost of each arc is then defined by a function F that maps each feature vector to a scalar cost. The shortest path is then the path between a source and target for which the sum of $F(w_G(a))$ is minimal for all arcs a in the path.

In this paper, we assume *linear travel costs*, i.e. we assume that F is of the form $F(x) = x \cdot \beta$ where β is a *preference vector* and $a \cdot b$ denotes the dot product. This preference vector represents the weighing of the different road features. For example, left turns might be very expensive, while right turns may be less so. A specific cost function F instantiated by such a weight vector β is called F_β.

The benefits of linear travel cost is that it is easy to implement, fast, and the simplicity of the model makes it less likely to overfit. Another benefit is that linear travel costs are distributive, i.e. $F_\beta(x_1) + F_\beta(x_2) = F_\beta(x_1 + x_2)$. This property is useful, as it allows us to describe the features of a route (i.e. a sequence of arcs) as the sum of the arc features. Consider a route with 3 traffic signal features and 7 right turns. Using the weight vector $[\text{traffic signal} = 10, \text{right turn} = 2]$ yields a route cost of $3 \times 10 + 7 \times 2 = 44$ while using the weight vector $[\text{traffic light} = 6, \text{right turn} = 5]$ makes the cost $3 \times 6 + 7 \times 5 = 53$. The underlying assumption here is that every occurrence of a feature has the same weight, e.g. the tenth left turn is just as bad as the first.

However, the linear model may not be able to accurately model user preferences if they are non-linear, e.g. a cost caused by wind resistance, which is proportional to the speed squared.

6.2. Finding new preference vectors

Consider a user driving from some source to a target. The user has some unknown cost function F, based on the user's actual preferences, and finds some preferred route p, i.e. a shortest path according to the cost function F. Meanwhile, our current approximated cost function F' finds a shortest path p' for the same source and target. We wish to modify F' such that it is closer to the unknown F. Since p is the shortest path according to F then it must be the case that $F(p) \leq F(p')$. Thus, to approximate F, we should modify F' such that $F'(p) \leq F'(p')$ is also the case.

We can see this as an optimization problem: given paths p,p', find an F' such that the loss function $\text{sign}(F'(p) - F'(p'))$ is minimized. This loss function can be seen as a penalty for F' attributing a larger cost to p than to p'.

As there might be many solutions to this problem, we sum over multiple trips to get a better overall solution. Let p_j and p'_j be the j-th path taken by a user and and the corresponding alternative path suggested by the framework, respectively. Our optimization problem is now to find a F' such that $\sum_j \text{sign}(F'(p_j) - F'(p'_j))$ is minimized.

For linear travel costs p,p' are described by feature vectors x, x' respectively, $F(x) = x \cdot \beta$, and $F'(x) = x \cdot \beta'$ where β,β' are preference vectors. Additionally, we constrain the cost function to non-negative costs as many of our preprocessing steps do not support negative costs. Negative costs can also lead to negative cycles in the graph, in which case there are no shortest path. Using these linear travel costs and non-negativity constraints, our problem now is to find β' such that $\sum_i \text{sign}(\max(x_i \cdot \beta', 0) - \max(x'_i \cdot \beta', 0))$ is minimized.

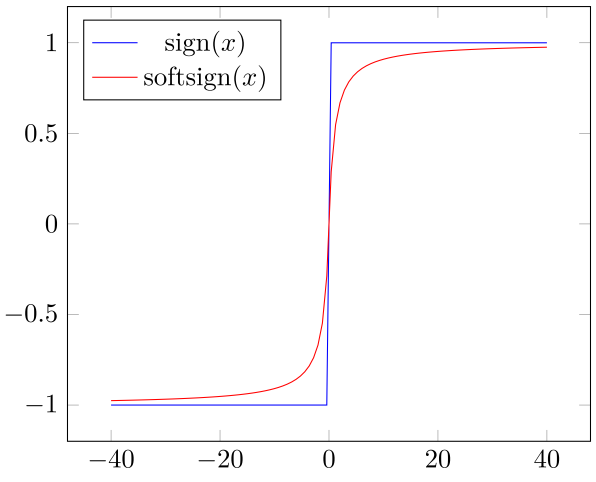

Figure 8: Comparison of sign and softsign.

An issue with this optimization problem is that it is highly discontinuous and with no gradients. This makes it difficult to utilize gradient descent methods to find the minimum. Therefore, we instead use the soft sign function defined by Bergstra et al. [7]:

$$\text{softsign}(x) = \frac{x}{|x| + 1}$$

This function is originally designed as an activation function in neural networks and is a continuous analog of the sign function, meaning that it has smooth gradients and approximates the function sign (see Figure 8).

Replacing sign with softsign, we now have the following loss function:

$$\sum_i \text{softsign}(\max(x_i \cdot \beta', 0) - \max(x'_i \cdot \beta', 0))$$

We use gradient descent to iteratively approach a minimum of this loss function. Once the loss function converges on a minimum, we have a new preference vector β''. But the path p' does not necessarily correspond to the shortest path with the new preference vector β''. Thus, we must find new shortest path p'' based on β'' and reiterate until the result converges.

We refer to the gradient descent iterations as *micro iterations*. Every n micro iteration we perform a new shortest path search and find new alternate paths. We refer to n micro iterations and a shortest path search as a "macro iteration".

6.3. TensorFlow

This optimization problem (described in Section 6.2) can easily be expressed using TensorFlow, as seen in the first function in Code Listing 2, where the parameters for the loss function are `c_pred` (the cost of the predicted route), `c_real` (the cost of the route the driver has actually taken), and `pref` (the current preference vector).

Code Listing 2 TensorFlow loss function in python.

```python
import tensorflow as tf

def loss(c_real, c_pred, pref):
  return tf.reduce_sum(tf.nn.softsign(
    tf.reduce_sum((c_real - c_pred) * pref, 1)))

def loss_multi(c_real, c_pred, pref, pref_idx):
  p = tf.gather(pref, pref_idx)
  return tf.reduce_sum(tf.nn.softsign(
    tf.reduce_sum((c_real - c_pred) * p, 1)))
```

For training several preference vectors, we can repeat the optimization for each user. Instead of doing this iteratively on a single preference vector, we chose to perform the optimization of each user in parallel. In order to do this, `pref` becomes a matrix with each row being a preference vector for a single user. In addition to the other parameters of the loss function, we now also need a vector parameter which selects which row in `pref` to use for each example. The resulting loss function is defined as `loss_multi` in Code Listing 2.

7. Framework Architecture

The architecture of our framework is centered around five main tasks: trip splitting, map matching, routing using CCH, routing using pgRouting, and CCH iterative learning using TensorFlow. Figure 9 shows how these tasks (diamonds) depend on each other and how the data (boxes) flows (arrows) through the framework. The framework consists of multiple open-source projects: OSRM (map matching), TensorFlow, CCH, and pgRouting. We have implemented TensorFlow and CCH using our personalized cost function and loss function.

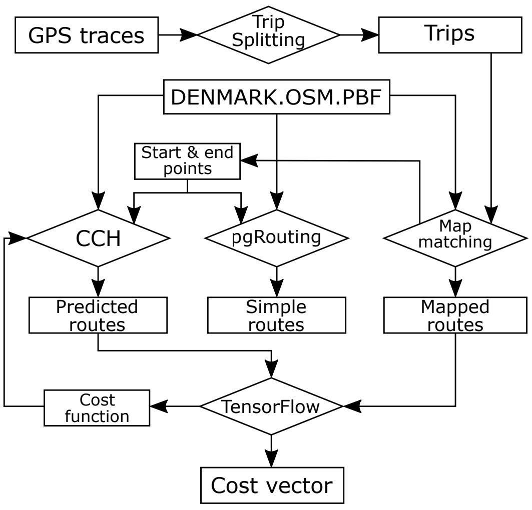

Figure 9: Architecture of our framework. The boxes represent data and the diamonds represent processes.

The input of our framework is the GPS traces of one or more user's for whom a preference vector should be created. These GPS traces are split into trips as described in Section 3.2. As mentioned in Section 3.2 a trip is split if there has been a stop, or no records have been observed, for more than 3 minutes. This interval might not be suitable for all parts of the world, due to differences in traffic patterns, e.g. length of red lights at intersections, so the framework would benefit from further analysis on this matter. The trips are then map matched using the method described in Section 4 resulting in map matched routes, also called matchings.

From OSM, the road network is extracted, as described in Section 3.1.5, to be used for map matching, CCH, and pgRouting. In this paper the Danish part of OSM was used, however it is possible to expand the road network by including OSM data from other road networks as well.

CCH and pgRouting both produce a route for each start and end point of the mapped routes (matchings) determined by the map matching algorithm. The route from CCH is found using our personalized cost function while pgRouting uses a static distance based cost function.

The predicted routes from CCH and the mapped routes from the map matching is given to TensorFlow which updates the personalized cost function as explained in Section 6.2. The new cost function is then provided to CCH resulting in new routes to be evaluated by TensorFlow. This process continues until the cost function converges, ideally when the personalization vector is representing the user's preferences exactly. The result is then a personalization vector that describes the routing preferences according to the provided trips. The personalization vector should use only GPS traces from a single user to best meet that user's preferences. The framework has a limitation in that it needs existing GPS data for a user to generate a preference vector and thereby be able to propose personalized routes. To help solve this problem we could provide several baseline preference vectors and then select the most suitable preference vector for a user based on their answers on an interview or survey, e.g. by showing several routes and ask them which they prefer.

Implementation problems

Trying to incorporate CRP into our architecture, we utilized the open-source C++ implementation developed by Wegner and Wolf [27]. Unfortunately, we found some problems with the implementation, as it relies on the patented graph partitioning software Buffoon, preventing us from using CRP and comparing it to CCH. Other graph partitioning methods, such as METIS[3] and KaHIP[4], were tried, however we found that the current implementation was not compatible with these methods. As such we had to exclude CRP from the architecture and instead focus on using CCH for personalization. Furthermore, we found that the CRP implementation was made with the intention of investigating results of routing and execution times for different routing queries, meaning important functions such as interpreting different turns was greatly simplified.

8. Experiments

We implemented our framework in C++, TensorFlow (using the Python API), and Node.JS. We utilize Cython to make C++ function calls from Python. We used PostgreSQL 9.6 with the PostGIS 2.3 extension as our database system. To learn the preference vectors we used TensorFlow r0.11, which enables us to utilize the GPU. The amount of code developed for this framework is listed in Table 3.

The hardware used for our experiments are two Intel Xeon Quad-Core E5620 (2.40 GHz, 4 Cores, 8 Threads, 12M Cache) processors, 20 GB DDR3 RAM, and a NVIDIA GeForce GTX 1070 OC (1.822 GHz core clock rate, 1,920 CUDA cores, and 8 GB of GDDR5 memory).

8.1. Data foundation

The GPS data used in this paper is collected during an insurance company campaign in Denmark [2]. As incentive to participate, the insurance company would give insurance premium discounts to participants staying within the speed limits while driving. The total

[3]http://glaros.dtc.umn.edu/gkhome/metis/metis/overview
[4]http://algo2.iti.kit.edu/kahip/

Language	Files	Comment	Code
SQL	42	74	1497
C++	6	60	1167
Python	2	26	173
Cython	1	2	118
JavaScript	1	0	86
Shell	2	16	52
C++ Header	1	0	25

Table 3: Lines of source codes used for developing this framework.

data set (before filtration) was collected throughout 2007 and 2008, and consists of observations from 191 drivers with a data density distribution as seen in Figure 10. In total 274 million raw GPS records were obtained, collected at a rate of 1 Hz and amounting to about 3 million kilometers driven in total. A more detailed data description can be seen in Appendix A.

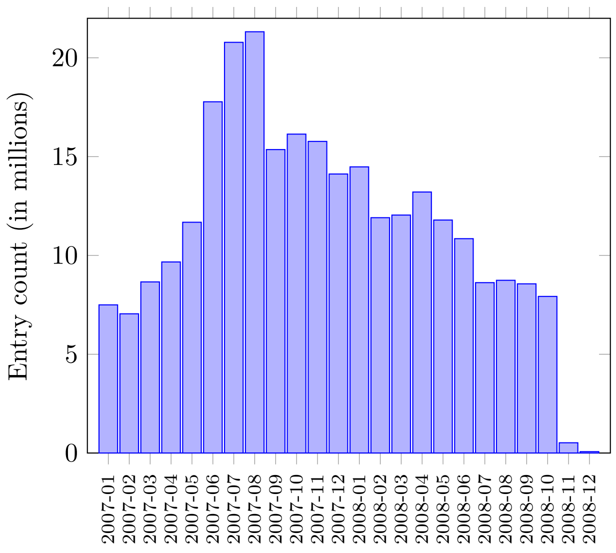

Figure 10: Data density across monthly intervals between 2007 and 2008.

8.1.1. Trip splitting

The collected traces of each driver is split into trips as described in Section 3.2. This resulted in 304,989 distinct trips with an average duration of 11 minutes and 26 seconds and an average traveled distance of 10.4 km. Some of the longest trips observed were more than 4 hours long and had distances above 450 km. Figure 11 shows the trip distribution across duration.

To reduce some of the noise in the data, trips with a duration of less than 30 seconds is ignored, leaving a total of 285,521 trips (203 million GPS records) with an average duration of 12 minutes and 13 seconds.

8.1.2. Map matching

After splitting the traces into trips, we found the actual routes taken using the OSRM map matching approach described in Section 4. As discussed in Section 4.1 the search radius used for map matching affects both execution time and accuracy. We tested 4 methods:

LowRadius	Static search radius of 15 meters. Default value when no radius input is given to the map matching algorithm.
HighRadius	Static search radius of 60 meter.
OSRM	Dynamic search radius defined as radius $= \min(200, 3.5 \cdot \text{HDOP} + 45)$. Default behavior when given an accuracy (here HDOP) for each GPS record.
CustomSR	Dynamic search radius defined in Section 4.1 equation (3).

Each method was tested and the results are compared with respect to run time for all trips, run time per trip, percentage of matched GPS records, percentage of trips not matched at all, and the average matchings per trip, where values closer to one indicates high cohesion and thus quality. The results for each method, along with the search radius range they use for our data set, are shown in Table 4.

Even though HighRadius seems to be slightly better than the CustomSR method, we choose to use CustomSR as it strikes a better balance between performance, accuracy and average matchings per trip than the other methods. Instead of using the raw GPS coordinates from the trips we will use the partially map matched trips (the matchings) to calculate and verify the preference vector for each driver. To make sure that the matchings we use contribute positively to the preference vector construction, we make one last filtration step. We ignore all matchings that are shorter than 500 meters as they are usually the result of data noise and contain few road features. We also ignore matchings that has a matching confidence less than 10 %. The effect of having this filtration is seen in Table 5.

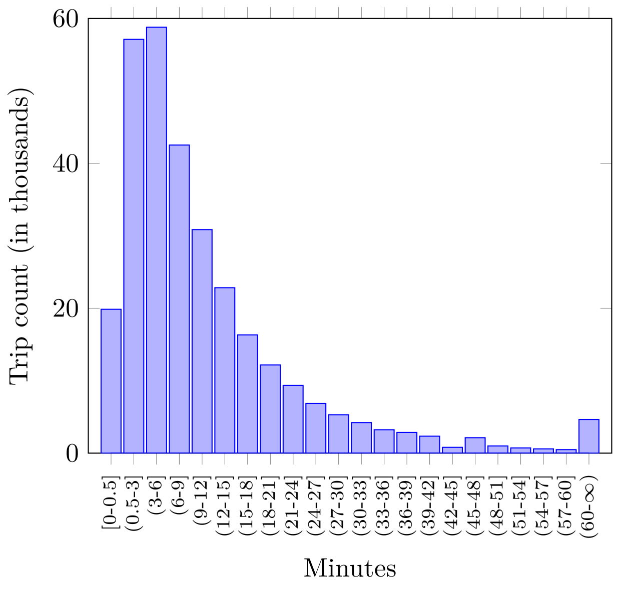

Figure 11: Trip distribution across duration.

Method: Search radius:	LowRadius (15 m)	HighRadius (60 m)	OSRM (45 - 200 m)	CustomSR (21 - 60 m)
Total run time (hh:mm:ss)	13:43:58	27:24:21	15:25:14	15:00:11
Run time/trip	173 ms	345 ms	195 ms	189 ms
Matched records	95.93 %	99.32 %	99.09 %	98.4 %
Non-matched trips	1.02 %	0.09 %	0.36 %	0.33 %
Average matchings/trip	3.99	2.05	3.64	2.65

Table 4: Map matching performance on 285,521 trips (203 million GPS records) for 4 different search radius methods.

	CustomSR	Filtered
Total trips	284,575	252,818
Avg. matchings/trip	2.65	1.42
Avg. matching confidence	52.16 %	92.97%
Avg. matching duration	358 s	719 s
Avg. matching distance	3.9 km	8.1 km

Table 5: Analysis of data before and after filtration.

The thresholds, 500 meters and 10 % confidence, are set relatively low to filter the worst matchings away.

The final data set used for learning and evaluation contains 1,963 matchings and 15,875 kilometers of data per driver on average, across 183 drivers.

8.2. pgRouting

To have a comparable baseline, we utilize routing algorithms in the PostGIS extension pgRouting, where appropriate spatial indexes have been made to ensure the comparison is fair. The algorithms performs routing based on a static cost function (using distance) provided by OSM. Changing it to suit our needs is not an issue, however making it dynamic is not possible.

In addition to the algorithms in pgRouting, we also made two additional modifications of Dijkstra's algorithm, that both work with a limited graph. The first, that we call `boxroute`, only looks at roads within a bounding box enclosing the source and target. The box has a padding, making it larger than just enclosing the source and target node. The second, named `polsroute`, takes the geodesic line (shortest Euclidean distance) between the source and target, and only considers roads within a certain distance of this straight line. The trade-off is that they may exclude the preferred route, or even make routing impossible, e.g. where crossing a bridge outside the padding is inevitable to get from source to target. In the tests, we try `boxroute` and `polsroute` each with both a 10 km and 70 km padding. Especially the 10 km padding suffers from failures because necessary roads are left out from the limited graph. The 70 km padding partially mitigates this problem for the Danish road graph, even though a path is not guaranteed to be found (the 10 km padding versions was unable to find a route in 3.55 % of the tested matchings, whereas the 70 km padding versions only failed to find a route for 1.32 %).

The data set used for testing is a representative subset of our map-matched GPS data. The use of a smaller subset is primarily due to run time concerns. From the map-matched GPS data, ignoring matchings shorter than 10 minutes, we choose 10 evenly distributed (over duration) matchings from each driver having at least 100 matchings. For each driver we sorted the matchings by duration, picked the first, the last, and 8 evenly spaced matchings in between. This gives us a selection of matchings of different durations without being biased towards any particular driving style. We also avoid that a large percentage of the matchings for any particular driver are the trips he takes every day, e.g. when commuting. In total, the subset consists of 1660 matchings.

The results of our test with the different algorithms can be seen in Table 6, and run time compared to distance of the points can be seen in Figure 12. bidirectional A* is omitted from the graph because, as seen in Table 6, it has a few outliers that are way off. Even though bidirectional A* has these outliers, it is still performing better than A* on average.

Algorithm	Min	Avg	Max
A*	3,628	3,770	4,525
A* (bidirectional)	2,556	3,005	133,012
Boxroute 10km	9	91	1,566
Boxroute 70km	179	886	3,619
Dijkstra	3,483	3,638	4,574
Dijkstra (bidirectional)	2,513	2,602	3,178
Polsroute 10km	8	89	1,560
Polsroute 70km	180	961	3,910

Table 6: Query run time (in ms) of the various pgRouting algorithms.

Comparing the pgRouting routes to our personalized routes can be seen in Figure 1, where the red route is chosen by pgRouting and the blue route is a personalized route suggestion based on a person who prefers faster road types. A comparable list of features from the two routes can be seen in Table 1.

8.3. RoutingKIT performance

To test the efficiency of RoutingKIT's CCH implementation we used the same 1660 matchings mentioned in Section 8.2. All three phases of CCH (preprocessing, customization, and querying) were tested. However, preprocessing of our graph (Denmark) takes less than 15 minutes, and only needs to be done once for every new version of the map, hence test results from the

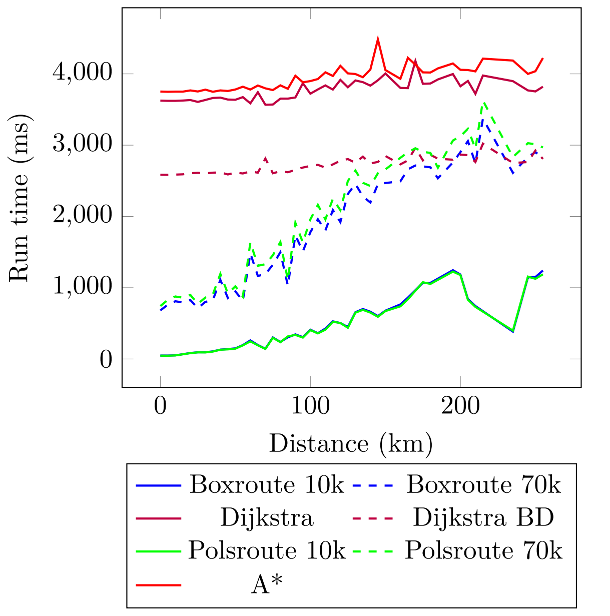

Figure 12: Average run time compared to distance of route for various pgRouting algorithms.

preprocessing phase is left out. The run times of the customization and query tests can be seen in Table 7 and Figure 13.

Algorithm	Customize	Query run time		
		Min	Avg	Max
CCH	1,001	14	15	20

Table 7: Run time (in ms) of the CCH algorithm.

We observe that CCH achieves outstanding results compared to the traditional routing algorithms used in pgRouting. When comparing CCH to the best performing pgRouting algorithm, CCH is 6 times faster on average, and the maximum query time is 78 times faster. Noting that only the pgRouting algorithms which limit the padding to 10 km are able to outperform CCH, and only on very short routes (longest matching in the test set, where CCH is not fastest has a traveling distance of 12,435 m). If we limit the comparison to the other algorithms, due to the 10 km padded algorithms not always finding a path or finding a sub-optimal path, the CCH algorithm is always faster, achieving a speedup of 173 times faster on average (59 times for the 70 km padding) and a maximum speedup of 159 times faster. Table 8 shows a summary of our findings when comparing CCH and pgRouting.

Algorithm	Avg (ms)	Max (ms)	Relative run time	
			Avg	Max
CCH	15	20	1	1
Best 10km	89	1,560	6	78
Best 70km	886	3,619	59	181
Best non-optimized	2,602	3,178	173	159

Table 8: Comparing query times for CCH and pgRouting.

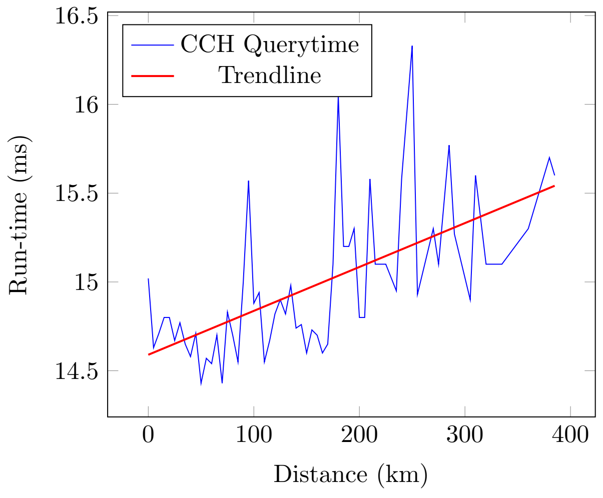

Figure 13: Average run time compared to distance for CCH.

When considering the response time experienced by the user, we have to consider the time it takes to customize the graph, and whether this adds to the response time. When running on a personal device the customization only needs to be performed when the cost function changes, but when used on a server shared by multiple users, this may have to be done every time. Even so, a response time of approximately 1 second, according to our tests, is comparable to the response time of off-the-shelf GPS navigation devices.

8.4. Preference vectors

In this section, we validate our personalization method by dividing our data set into a 80/20 training/test split. The loss function of both is presented, but only the training set is used for optimization. Before optimization begins, we normalize all features by their maximum observed value such that common features (e.g. meters driven) are weighed less and uncommon features (e.g. turns) are weighed more. We then perform personalization in a two-step process:

1. Starting from a shortest-distance preference vector, find a common baseline preference vector based on all matchings.

2. Starting from the baseline preference vector, find a personal preference vector for each driver.

Step 1 is done by first initializing the preference vector to have 1 in the distance entry and 0 in all other entries. Finding the baseline is done by ignoring the driver id and performing personalization as described in Section 6.2. The result is a single preference vector describing the average driver preference. Hence performing 20 macro iterations (finding new alternative routes and shortest paths) with 20 micro iterations (gradient descent iterations) each. We chose 20 macro iterations, as we experienced high memory usage beyond that. As the single-vector personalization converges very quickly, only 20 micro iterations per macro iteration are necessary.

As illustrated in Figure 14, the loss starts at 0.69 and initially descends quickly until it flattens out as the training progresses and the result finally converges at -0.56. The small spikes at $x = 20$, $x = 40$, and so on, are the results of each macro iteration where we add new alternative routes. The resulting vector can be seen in Appendix C. The baseline unexpectedly shows that right turns are 62% more expensive than left turns, opposite our intuition that right turns would be less expensive than left turns. We hypothesize on the reason for these results in Section 8.6.

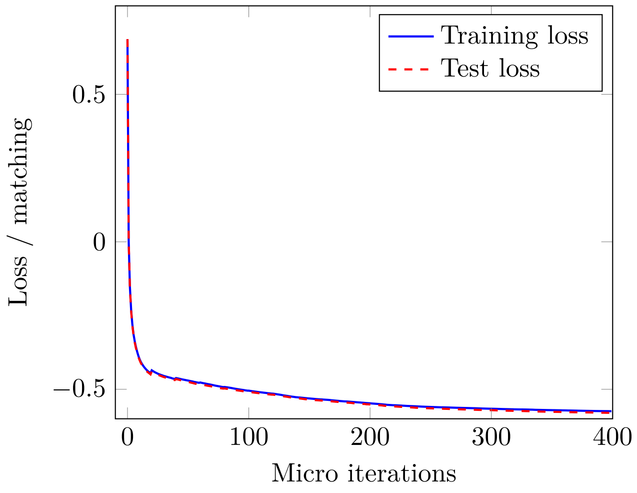

Figure 14: Loss across single-vector training.

After we find this baseline vector, we perform step 2. Here we repeat the same process, but with a separate vector for each driver and using the baseline from step 1 as the initial preference vector. This converges much more slowly compared to the former step, and finding alternative paths takes significantly longer because CCH needs to be customized to each user. For this reason, we increase the micro iterations per macro iteration to 1000. The result of step two is seen in Figure 15.

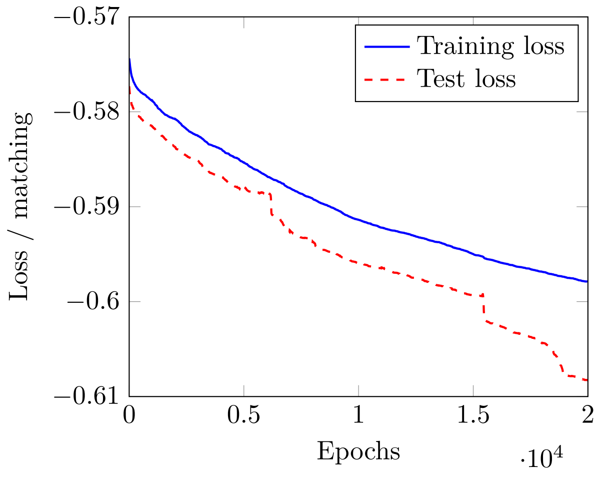

Figure 15: Loss across multi-vector training.

While the test set generally has a lower loss than the training set, we believe this to be primarily noise. The fact that the test set is not performing significantly worse than the training set indicates that overfitting is not a problem and that the method successfully generalizes to future matchings and not just the ones it has been trained with. The output of this is a vector for each driver. A summary of this data can be seen in Appendix C.

The preference vectors that we learn cannot easily be interpreted just by examining the preference weights. From the weights found, it is possible to determine whether a driver likes left turns more than right turns, but it is not possible to infer if the driver likes left turns in general. The weight of left turns is only meaningful in relation to all the other feature weights, and thus a single weight in isolation says nothing about the preferences of a driver.

We can visualize these vectors by reducing the dimensions to 2 using t-Distributed Stochastic Neighbor Embedding (t-SNE) [21] as seen in Figure 16. t-SNE has the property that points close to each other in the original space are likely to be close in the embedded space [21], and it thus gives us a good overview over how close each pair of points are.

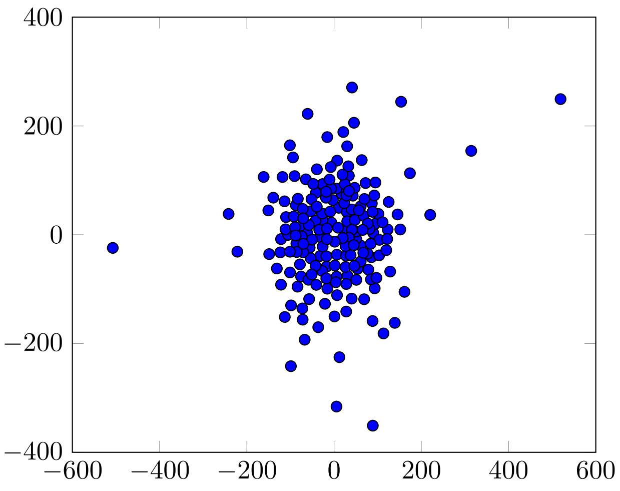

Figure 16: t-SNE reduced preference vectors.

We can see that generally the vectors are collected around a central point with some deviation from the mean. This seems to indicate that people generally value features the same, with few minor variations. This indicates that it might be possible to train a lower number of vectors than one for each driver; if two drivers have sufficiently similar driving preferences, they might be able to use the same preference vector. To see the potential of using a fewer number of preference vectors, we perform K-means clustering with a varying number of clusters and examine the sum of the squared distance to the nearest cluster in Figure 17.

We can see that already at 50 clusters, the squared distance is highly reduced. At 100, it is practically indistinguishable from its final value at 184. Some kind of clustering, instead of finding a preference vector for each driver, could reduce overhead significantly by not learning redundant vectors. We leave exploring these possible optimizations for future work.

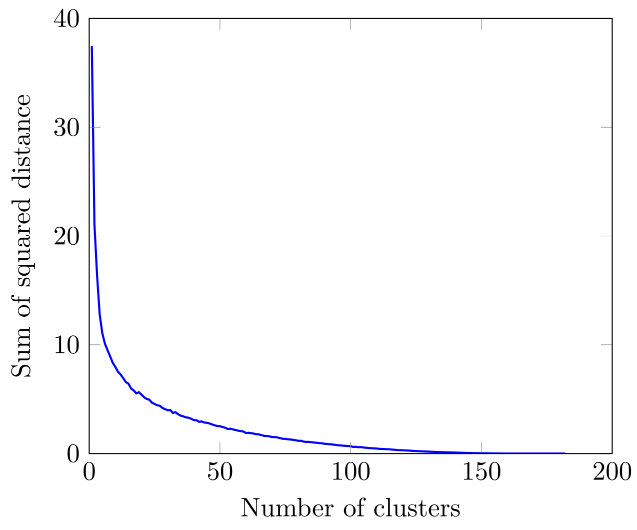

Figure 17: Effect of clustering.

8.5. GPU accelerated training

Optimizing the loss function is a computationally intensive task, which we need to repeat every time we change the cost function, e.g. when we change which road features from OSM to examine. We would also need to repeat it at regular intervals, if we were still collecting data from the drivers. It is thus desirable, to be able to perform this task in the shortest possible amount of time. Using the GPU for this can speed this up by more than one order of magnitude, which can be seen in Table 9. Note that finding shortest path in each macro-iteration is still CPU-bound, as pathfinding is a largely serial task and thus cannot be run efficiently on the GPU. The training however, can use TensorFlow to fully utilize the GPU.

	Run time (hh:mm:ss)
NVIDIA GPU	00:55:18
2×Intel Xeon CPUs	10:25:04

Table 9: Comparing run time for training using our NVIDIA GPU vs. our two Xeon CPUs.

8.6. Sources of errors

In this section, we identify potential sources of errors that one should be aware of.

Firstly, as described in Section 4, we utilize the map matching from OSRM which is implemented using a Hidden Markov Model (HMM). This HMM is built on some prior knowledge and assumptions on how people drive and as such might skew how a trip is matched in certain scenarios, e.g. when a motorway and smaller road run in parallel and the GPS points indicate that the car is somewhere in between the two roads, the algorithm might deem it more probable that the car drives on the motorway, even though the closest road might be the smaller road.

Secondly, the map used in the map matching is extracted from OSM data from 2016, whereas the GPS data is collected in 2007 and 2008. Because there have been several changes to the road network in between data collection and map matching the result of the map matching might in some cases deviate from the actual path taken. In the cases of extreme changes to the road network, we hypothesize that the map matching algorithm have just failed, leaving out the trip from our testing and evaluation data set.

Thirdly, some of the GPS data collected might be the result of the driver following a GPS navigation system. In these cases, the driver's personal preference vector might instead capture the cost function used by the GPS navigation system instead of the driver's actual preferences, but for our data we assume that people mostly have not used GPS navigation software.

9. Conclusion

We present a framework to determine driving preferences using GPS traces. Using these preferences, we show how to provide personalized routes similar to the routes previously driven by the users. The framework relies on OSM to which we map match our GPS trajectories using OSRM. We use the result from the map matching to obtain road features of the driven routes and perform routing. Relying solely on data provided by OSM and other open-source routing and processing libraries allows the framework to be used in any region covered by OSM. Our framework projects the map matched data onto a line graph used by CCH to achieve efficient routing, and utilizes TensorFlow to learn the preference vector for a given user based on the observed routes. This vector is used by CCH to generate routes according to the user's preferences. By using CCH in favor of traditional methods such as Dijkstra and A*, we achieve a speedup of up to 173 times (on average) compared to pgRouting as seen in Section 8.2.

We also showed that a substantial increase in performance was possible when finding the preferences of users. This was achieved by utilizing the computational capabilities of GPUs in favor of traditional CPU based configurations. As seen in Section 8.5 performance is increased by one order of magnitude when using a single GPU compared to two CPU's.

Our experiments in Section 8.4 show that routes generated by simple cost functions such as shortest distance generally does not fit well with user preferences. We also see that a general preference vector actually fits all users quite well, indicating that the user preferences is in fact quite similar. We do, however, see that further improvement is still possible by creating personalized routes for the individual users.

One limitation of the framework is the inability to explicitly specify preferences for a given route, i.e. the preference vectors are not easily understandable and can as such not be explicitly defined as described in Section 8.4. Another limitation is that the current implementation assumes linear costs for all features and users, which means that we cannot model some user preferences, e.g. a cost caused by wind resistance, which is proportional to the speed squared.

Despite these limitations, the framework has a very general use, and could even be used without OSM, provided that another map is used along with some replacement for OSRM is used.

10. Future work

In this section, we will discuss various research and extensions that can improve this framework.

As explained in Section 8.4, user's are not that unique in their driving preferences. Using a kind of clustering to compute fewer preference vectors could improve performance dramatically, while still finding preference vectors close to the optimal.

A natural extension of shortest path calculations is to include a metric for travel time. Currently the features include distances driven on specific roads. These roads all have a speed class assigned and on this basis we could implement a naive travel time. However, a more precise travel time metric would be based on recorded travel time for each road segment determined by the observed GPS traces on that road segment. As this data is not available for all road segments, the travel time metric would only be on a small subset of all roads. An alternate way to evade that problem would be to predict the expected travel time based on similar road segments.

Another interesting feature to include in our framework is *time dependency*, which is the concept of predicting driving preferences for different times of a day, e.g. each hour of the day. From the current data foundation we can already see how the time of day (e.g. rush hour and weekends) affects the average speed, especially in and around larger cities. To include this in the framework we would have to develop a time dependent version of CCH, where the customization would be calculated for each preselected time-slot. Another aspect of time dependency is that a users preferences might also change depending on the time of day. A user might for example want to take the fastest route to work in the morning, and a slower route avoiding motorways when driving home in the afternoon.

It would be interesting to measure the extent of any performance improvement gained by utilizing the GPU for the CCH customization phase as Delling et al. [10] does for CRP. This could be compared to storing customized graphs on disk and reload the graphs as needed.

References

[1] Global positioning system standard positioning service performance standard, September 2008. URL http://www.gps.gov/technical/ps/2008-SPS-performance-standard.pdf.

[2] Niels Agerholm, Nerius Tradisauskas, Brith Klarborg, Lisbeth Harms, and Harry Lahrmann. SPAR PÅ FARTEN: de første resultater af et intelligent farttilpasnings-projekt i nordjylland baseret på incitament (forsikringsrabat). 2007. URL http://vbn.aau.dk/files/13658104/Adfaerd_agerholm_TD2007.pdf.

[3] J. Anez, T. De La Barra, and B. Pérez. Dual graph representation of transport networks. *Transportation Research Part B: Methodological*, 30(3):209 – 216, 1996. ISSN 0191-2615. doi: http://dx.doi.org/10.1016/0191-2615(95)00024-0. URL http://www.sciencedirect.com/science/article/pii/0191261595000240.

[4] Adrian Balteanu, Gregor Jossé, and Matthias Schubert. Mining driving preferences in multi-cost networks. In *Proceedings of the 13th International Conference on Advances in Spatial and Temporal Databases*, SSTD'13, pages 74–91, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-40234-0. doi: 10.1007/978-3-642-40235-7_5. URL http://dx.doi.org/10.1007/978-3-642-40235-7_5.

[5] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. *CoRR*, abs/1504.05140, 2015. URL http://arxiv.org/abs/1504.05140.

[6] Leonard E. Baum and Ted Petrie. Statistical inference for probabilistic functions of finite state markov chains. *Ann. Math. Statist.*, 37 (6):1554–1563, 12 1966. doi: 10.1214/aoms/1177699147. URL http://dx.doi.org/10.1214/aoms/1177699147.

[7] James Bergstra, Guillaume Desjardins, Pascal Lamblin, and Yoshua Bengio. Quadratic polynomials learn better image features. Technical report, Technical Report 1337, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, 2009.

[8] J. Dai, B. Yang, C. Guo, and Z. Ding. Personalized route recommendation using big trajectory data. In *2015 IEEE 31st International Conference on Data Engineering*, pages 543–554, April 2015. doi: 10.1109/ICDE.2015.7113313.

[9] Renato Werneck Daniel Delling, Moritz Kobitzsch. Customizing driving directions with gpus. In *Proceedings of the 20th International Conference on Parallel Processing (Euro-Par 2014)*. Springer, January 2014. URL https://www.microsoft.com/en-us/research/publication/customizing-driving-directions-with-gpus/.

[10] Daniel Delling, Andrew V. Goldberg, Moises Goldszmidt, John Krumm, Kunal Talwar, and Renato F. Werneck. Navigation made personal: Inferring driving preferences from gps traces. In *Proceedings of the 23rd SIGSPATIAL International*

Conference on Advances in Geographic Information Systems, GIS '15, pages 31:1–31:9, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3967-4. doi: 10.1145/2820783.2820808. URL http://doi.acm.org/10.1145/2820783.2820808.

[11] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks. *Transportation Science*, 2015. URL http://dx.doi.org/10.1287/trsc.2014.0579.

[12] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. *J. Exp. Algorithmics*, 21(2):1.5:1–1.5:49, April 2016. ISSN 1084-6654. doi: 10.1145/2886843. URL http://doi.acm.org/10.1145/2886843.

[13] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[14] Jon Froehlich and John Krumm. Route prediction from trip observations. In *SAE Technical Paper*. SAE International, 04 2008. doi: 10.4271/2008-01-0201. URL http://dx.doi.org/10.4271/2008-01-0201.

[15] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. *Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks*, pages 319–333. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-68552-4. URL http://dx.doi.org/10.1007/978-3-540-68552-4_24.

[16] Robert Geisberger, Michael N Rice, Peter Sanders, and Vassilis J Tsotras. Route planning with flexible edge restrictions. *Journal of Experimental Algorithmics (JEA)*, 17:1–2, 2012. URL http://dl.acm.org/citation.cfm?id=2133805.

[17] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, August 2012. ISSN 1526-5447. doi: 10.1287/trsc.1110.0401. URL http://dx.doi.org/10.1287/trsc.1110.0401.

[18] Ruihong Huang and Christina Kennedy. Uncovering hidden spatial patterns by hidden markov model. In *Proceedings of the 5th International Conference on Geographic Information Science*, GIScience '08, pages 70–89, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87472-0. doi: 10.1007/978-3-540-87473-7_5. URL http://dx.doi.org/10.1007/978-3-540-87473-7_5.

[19] Julia Letchner, John Krumm, and Eric Horvitz. Trip router with individualized preferences (trip): Incorporating personalization into route planning. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 1795. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[20] Dennis Luxen and Christian Vetter. Real-time routing with openstreetmap data. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '11, pages 513–516, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1031-4. doi: 10.1145/2093973.2094062. URL http://doi.acm.org/10.1145/2093973.2094062.

[21] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[22] Paul Newson and John Krumm. Hidden Markov map matching through noise and sparseness. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '09, pages 336–343, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-649-6. doi: 10.1145/1653771.1653818. URL http://doi.acm.org/10.1145/1653771.1653818.

[23] Kayur Patel, Mike Y. Chen, Ian Smith, and James A. Landay. Personalizing routes. pages 187–190, 2006. doi: 10.1145/1166253.1166282. URL http://doi.acm.org/10.1145/1166253.1166282.

[24] Project-OSRM. Open Source Routing Machine - C++ backend. https://github.com/Project-OSRM/osrm-backend/blob/5.5/include/engine/routing_algorithms/map_matching.hpp#L431, 2016.

[25] Aaron Schild and Christian Sommer. On balanced separators in road networks. *J. Exp. Algorithmics*, 9125:286–297, 2015. doi: 10.1007/978-3-319-20086-6_22. URL http://dx.doi.org/10.1007/978-3-319-20086-6_22.

[26] Steven S. Skiena. *The Algorithm Design Manual*. Springer, second edition, 2008.

[27] Michael Wegner and Matthias Wolf. C++ implementation of customizable route planning (CRP). https://github.com/michaelwegner/CRP, 2016.

[28] Stephan Winter. Modeling costs of turns in route planning. *GeoInformatica*, 6(4):345–361, 2002. ISSN 1573-7624. doi: 10.1023/A:1020853410145. URL http://dx.doi.org/10.1023/A:1020853410145.

[29] Stephan Winter. *Route Specifications with a Linear Dual Graph*, pages 329–338. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-642-56094-1. doi:

10.1007/978-3-642-56094-1_24. URL http://dx.doi.org/10.1007/978-3-642-56094-1_24.

[30] Bin Yang, Chenjuan Guo, Yu Ma, and Christian S. Jensen. Toward personalized, context-aware routing. *The VLDB Journal*, 24(2):297–318, April 2015. ISSN 1066-8888. doi: 10.1007/s00778-015-0378-1. URL http://dx.doi.org/10.1007/s00778-015-0378-1.

Appendix

A. Basic data analysis

This section describes some basic data analysis methods that we used, and includes a heat map of our data as well as a description of how we identified urban areas needed for determining speed limits.

A.1. GPS records

Figure 18 shows the distribution of all GPS records throughout Denmark described in Section 8.1, before any filtration. The recordings are counted in squares of 1 km and the colors indicate the number of observed GPS records in this square. The intervals defined in the legend follow an exponential function. As seen in the figure, some records are placed in the sea, which is likely to be what we would classify as outliers and discard.

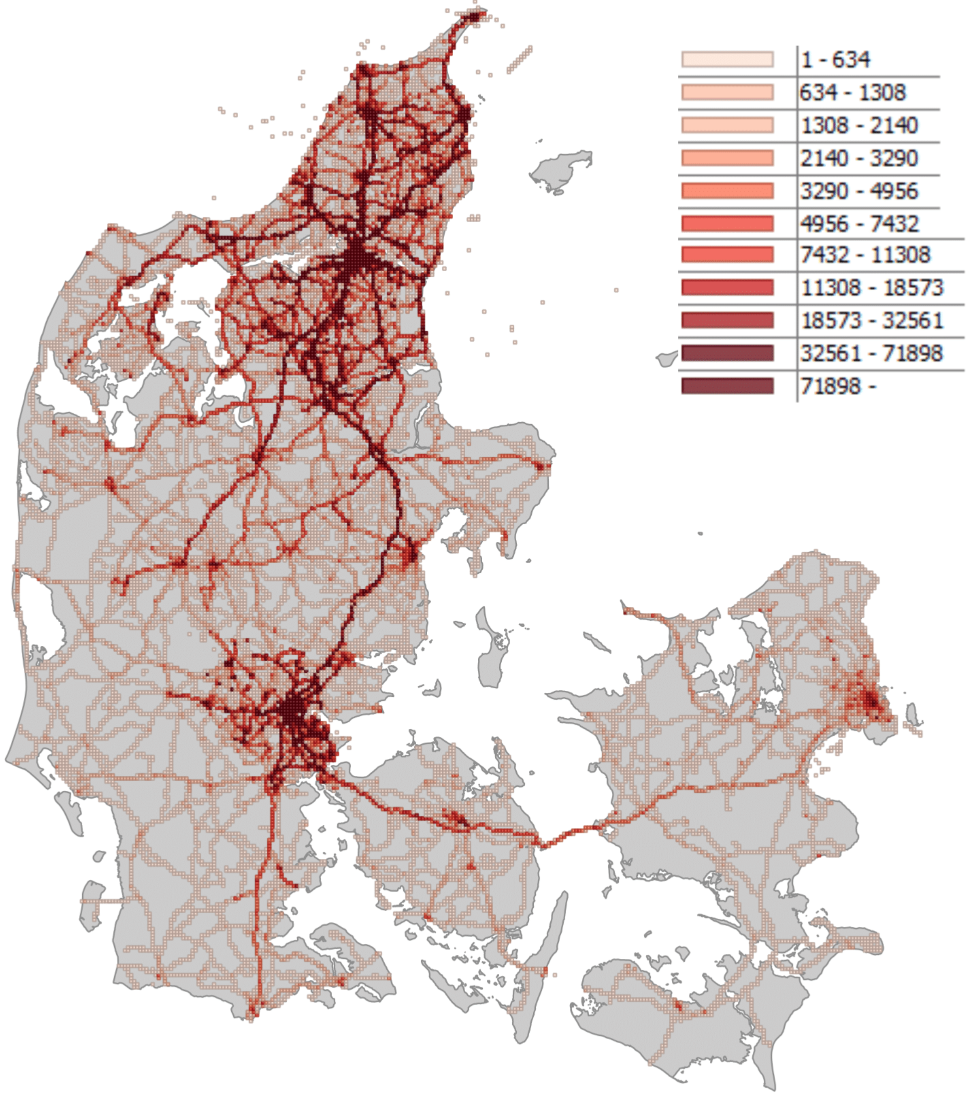

Figure 18: Distribution of GPS records, heatmap with 1 km squares.

A.2. Identifying urban areas in Denmark

We investigated whether it was possible to infer if an arcs is inside an urban area, based on spatial data in OSM. Some urban areas, such as *villages* and *towns*, is available but did not sufficiently covers all urban areas in Denmark. In order to expand the areas we included the *residential*, *industrial*, and *retail* areas. As these areas did not include the nearby roads we used the PostGIS `ST_Buffer` function to geometrically expand the areas 250 meters in all directions. Merging all overlapping areas (union) and taking the convex hull of each resulting area results in smoother areas with more resemblance to real urban boundaries. The PSQL statement to calculate these urban areas can be seen in Code Listing 3.

Code Listing 3 PSQL statement for finding urban areas.

```
CREATE TABLE urban_areas AS (
  SELECT ST_ConvexHull((ST_Dump(ST_Union(ST_Buffer(ST_MakePolygon(linestring),250)))).geom) as area
  FROM osm.ways
  WHERE ST_NPoints(linestring) > 3
  AND ST_IsClosed(linestring)
  AND (tags -> 'landuse' IN ('residential', 'industrial' , 'retail')
  OR tags -> 'place' IN ('village','town'))
);
```

With a slightly rough but clear definition of where the urban areas of Denmark lies we then classify a arcs to inside an urban area if the head or tail node of the arcs in contained inside any of the polygon in `urban_areas`. To decide if a point in inside a polygon we utilize `ST_Contains(polygon, point)`. All the urban in Denmark are outlined in Figure 19.

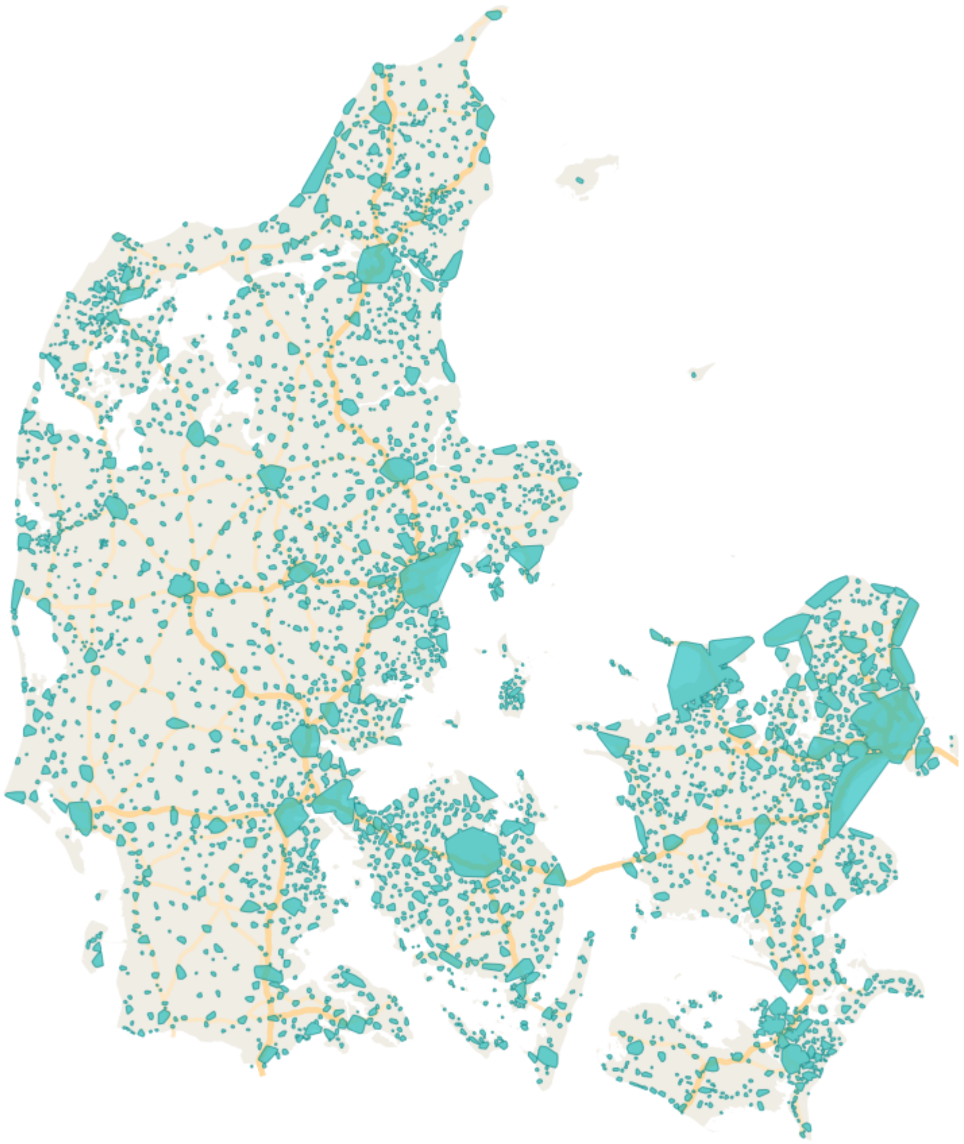

Figure 19: Approximate urban areas (turquoise) in Denmark.

B. Trip splitting

The SQL code written in Code Listing 4 is used to extract all trips from the table of traces, with the method described in Section 3.2.

Code Listing 4 SQL code for selecting all trips from table `traces`.

```
SELECT q3.tripid,
       q3.driver_id,
       tsrange(MIN(q3.time_stamp), MAX(q3.time_stamp), '[]'::text) AS range,
       int4range(MIN(q3.id), MAX(q3.id) + 1) AS id_range
FROM
  ( SELECT MAX(q2.tripid_start) OVER (PARTITION BY q2.driver_id
           ORDER BY q2.time_stamp ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS tripid,
           q2.id,
           q2.driver_id,
           q2.time_stamp
    FROM
      ( SELECT q1.id,
               q1.driver_id,
               q1.time_stamp,
               CASE
                   WHEN q1.driver_id <> COALESCE(q1.prev_driver_id::integer, 0)
                        OR ( q1.prev_ts + '00:03:00'::interval SECOND) < q1.time_stamp
                           THEN nextval('trip_seq'::regclass)
                   ELSE NULL::bigint
               END AS tripid_start
        FROM
          ( SELECT q0.id,
                   q0.driver_id,
                   q0.time_stamp,
                   LAG(q0.driver_id) OVER (PARTITION BY q0.driver_id ORDER BY q0.time_stamp) AS
                        prev_driver_id,
                   LAG(q0.time_stamp) OVER (PARTITION BY q0.driver_id ORDER BY q0.time_stamp) AS prev_ts
            FROM
              ( SELECT traces.id::integer AS id,
                       traces.driver_id,
                       traces.time_stamp
                FROM traces
                WHERE traces.odospeed > 0) q0) q1) q2) q3
GROUP BY q3.tripid, q3.driver_id;
```

C. Preference vectors

In Table 10 we illustrate preference vectors obtained from our experiments in Section 8, plus an example of a personal preference vector from our result. A detailed analysis can be found in Section 8.4.

	Weight	Baseline	Personalized			
			Average	Median	Std. dev.	Example
Distance	1.000e+00	9.293e-01	5.357e-01	4.801e-01	3.284e-01	2.912e-01
Motorway	1.001e+00	-1.111e-02	4.290e-03	1.636e-03	3.621e-02	1.514e-02
Highway	2.335e+00	5.129e-02	7.412e-02	5.855e-02	5.776e-02	4.915e-02
Urban	3.514e+01	8.482e-01	1.411e+00	1.209e+00	9.437e-01	9.029e-01
Unclassified	9.119e+00	2.109e-01	4.351e-01	3.466e-01	3.331e-01	2.720e-01
Asphalt	1.000e+00	-6.744e-02	-6.872e-02	-6.983e-02	3.947e-02	-4.019e-02
Other paved	3.239e+01	1.068e-01	5.247e+00	3.454e+00	5.187e+00	1.454e+01
Unpaved	5.196e+01	4.911e-01	7.424e+00	6.072e+00	6.810e+00	1.147e+00
Dist. at urban speed	8.506e+00	2.767e-01	4.202e-01	3.369e-01	2.653e-01	2.578e-01
Dist. at rural speed	2.549e+00	5.917e-02	1.164e-01	1.036e-01	7.154e-02	7.237e-02
Dist. at motorway speed	1.009e+00	-1.003e-02	6.522e-02	4.394e-02	8.523e-02	1.753e-01
1 Lane	1.131e+00	1.356e-02	4.136e-02	3.235e-02	5.000e-02	4.024e-02
2 Lane	5.655e+00	7.136e-02	5.109e-01	3.894e-01	4.285e-01	2.702e-01
Roundabout	3.044e+02	5.248e+00	1.360e+01	1.032e+01	1.203e+01	5.829e+00
Tunnel	9.548e+01	8.198e-01	1.232e+01	9.137e+00	1.064e+01	3.456e+00
Bridge	1.868e+01	1.948e-01	1.250e+00	1.001e+00	1.033e+00	4.860e-01
Traffic signal	4.950e+03	1.145e+02	2.700e+02	1.977e+02	2.626e+02	1.439e+02
Crossing	6.033e+03	1.128e+02	4.006e+02	2.745e+02	4.006e+02	1.411e+02
Traffic calmings	5.850e+03	8.627e+01	3.952e+02	2.646e+02	4.148e+02	1.523e+02
Left Turn	5.218e+03	2.023e+02	2.936e+02	2.309e+02	2.021e+02	1.774e+02
Right Turn	1.044e+04	3.280e+02	4.892e+02	3.745e+02	3.404e+02	2.793e+02

Table 10: Overview of preference vectors found by our method.

D. Handling large data sets using a small server

When handing large amount of data on a relatively small server with inferior hardware (specification described in Section 8) compared to large server racks, performance often gets affected, to overcome some of these problems we used varies techniques, mainly on the database system. This appendix contains two examples, of how we have improved the performance of our data flows in this semester.

D.1. Avoiding delays when map matching

The input to map matching is a join between the GPS data table and the trip table. Map matching needs to receive this data ordered by trip and timestamp. Our server can perform this join and ordering in around 2 hours, after which the actual map matching starts.

Because map matching spends a lot longer processing the data, than it takes the database to deliver them, we expected to be able to remove this 2 hour delay. By storing the join in a table, the situation is improved. However the delay was not gone, because the data still needs to be sorted, which takes about 15 minutes. Creating an index containing all the data made no difference, because PostgreSQL 9.6 calculated a lower cost for a sequential scan and sorting, than for an index only scan. In this case, we could set the PostgreSQL setting `enable_seqscan` to false to get the desired effect. After these improvements, there are no noticeable delay before the map matching starts.

D.2. Parsing JSON objects in small batches

The output from the map matching is returned as JSON objects, which we store in the database, and parse and copy it to relational tables when the map matching is complete. We use the JSON data as a reference in case we discover a field we did not parse, or if we need to be sure the parsed data are correct. The parsing is a slow process, especially if we try to parse all in the same statement execution. If we parse using one statement execution each, it is slower than parsing multiple objects together. This performance increases as the batch size increases until some optimal batch size, but then the process becomes progressively slower as we increase the number of objects in each batch. We did not investigate the optimal batch size, but found a batch size of 500 JSON objects (where an JSON object corresponds to a single trip) to be a good choice. The consequence of this is, that we need to perform 610 parsing statements. Fortunately PostgreSQL allows us to use a script to create a new SQL script, as seen in Code Listing 5.

Code Listing 5 SQL script for creating and executing a script, which will parse all JSON output from the map matching.

```
-- Fetch the last trip id, store in :tripend
select max(tripid) as tripend from :schema.trip_routes \gset

-- remove script from last run (if any)
\! rm -f call_parse_fnc.sql

-- Only selected data in output (no headers etc.)
\pset tuples_only on

-- 500 in each batch
\set jobsize 500

-- Create the script file
\o call_parse_fnc.sql
select 'select ' || :'schema' || '.mapmatch_parse_fnc(' || from_tripid::text || ', ' || to_tripid::text
    || ');' from (select d from_tripid, d+:jobsize-1 to_tripid from (select
    generate_series(1,:tripend-1,:jobsize) d) q2 order by 1 asc) q;
\o

-- Call the newly created script file
\i call_parse_fnc.sql

-- Cleanup
\! rm -f call_parse_fnc.sql
```

In case we would like to use the JSON data before it is stored in relational tables, we created two views for this purpose, as seen in Code Listing 6. Using the views is not as fast as using the relational tables, but as long as we limit the queries to a few trips, the difference is not noticeable by humans.

Code Listing 6 Example view over map matching JSON output.

```
-- create view with matchings including legs (json) for child views
create view :schema.v_matching_sub as
  select tripid, matchno::integer, confidence, duration, distance,
         ST_Transform(ST_LineFromEncodedPolyLine(geometry), 25832) as geom,
         legs as mleg
    from :schema.trip_routes,
         jsonb_array_elements(route->'matchings') with ordinality as t(matching, matchno),
         jsonb_to_record(matching) as x(confidence real, duration real, distance real, geometry text,
              legs jsonb);

-- create the view expeted to be used by end-user queries
create view :schema.v_matching as
  select tripid, matchno, confidence, duration, distance, geom from :schema.v_matching_sub;
```