Revisiting revisits in trajectory recommendation

Aditya Krishna Menon, Dawei Chen, Lexing Xie, Cheng Soon Ong
The Australian National University   Data61/CSIRO
{aditya.menon,u5708856,lexing.xie,chengsoon.ong}@anu.edu.au

ABSTRACT

Trajectory recommendation is the problem of recommending a sequence of places in a city for a tourist to visit. It is strongly desirable for the recommended sequence to avoid loops, as tourists typically would not wish to revisit the same location. Given some learned model that scores sequences, how can we then find the highest-scoring sequence that is loop-free?

This paper studies this problem, with three contributions. First, we detail three distinct approaches to the problem – graph-based heuristics, integer linear programming, and list extensions of the Viterbi algorithm – and qualitatively summarise their strengths and weaknesses. Second, we explicate how two ostensibly different approaches to the list Viterbi algorithm are in fact fundamentally identical. Third, we conduct experiments on real-world trajectory recommendation datasets to identify the tradeoffs imposed by each of the three approaches.

Overall, our results indicate that a greedy graph-based heuristic offer excellent performance and runtime, leading us to recommend its use for removing loops at prediction time.

CCS CONCEPTS

• **Information systems** → **Recommender systems**;

1 INTRODUCTION

A burgeoning sub-field of citizen-centric recommendation focusses on suggesting travel routes in a city that a tourist might enjoy. This goal encompasses at least three distinct problems:

(1) ranking *all* points of interest (POIs) in a city in a manner personalised to a tourist (e.g. a tourist to Sydney interested in scenic views might have `Opera House` ≻ `Darling Harbour` ≻ `Chinatown`) [9, 10, 20, 21];
(2) recommending the *next* location a tourist might enjoy, given the sequence of places they have visited thus far (e.g. given `Darling Harbour`→`Botanic Gardens`, we might recommend `Quay Café`) [5, 18, 22];
(3) recommending an *entire sequence* of POIs for a tourist, effectively giving them a travel itinerary (e.g. `Opera House`→`Quay Café`→`Darling Harbour`) [2, 8, 11–13].

Our focus in the present paper is problem setting (3), which we dub "trajectory recommendation".

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CitRec, August 27, 2017, Como, Italy
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5370-0/17/08…$15.00
https://doi.org/10.1145/3127325.3127326

Effectively tackling trajectory recommendation poses a challenge: at training time, how can one design a model that can recommend sequences of POIs which are coherent as a *whole*? Merely concatenating a tourists' personalised top ranking POIs into a sequence might result in prohibitive travel (e.g. `Opera House`→`Royal National Park`), or unacceptably homogeneous results (e.g. we might recommend three restaurants in a row). This motivates approaches that ensure *global cohesion* of the predicted sequence. In recent work, Chen et al. [4] showed that structured SVMs are one such viable approach which outperform POI ranking approaches.

In this paper, we focus on a distinct but related challenge: at prediction time, how can one recommend a sequence that does not have *loops*? This is desirable because tourists would typically wish to avoid revisiting a POI that has already been visited before. In principle, this problem will not exist if one employs a suitably rich model which learns to suppress sequences with loops. In practice, one is often forced to compromise on model richness owing to computational and sample complexity considerations. We thus study this challenge, with the following contributions:

(C1) We detail three different approaches to the problem – graph-based heuristics, integer linear programming, and list extensions of the Viterbi algorithm – and qualitatively summarise their strengths and weaknesses.
(C2) In the course of our analysis, we explicate how two ostensibly different approaches to the list Viterbi algorithm [15, 19] are in fact fundamentally identical.
(C3) We conduct experiments on real-world trajectory recommendation datasets to identify the tradeoffs imposed by each of the three approaches.

Overall, we find that all methods offer performance improvements over naïvely predicting a sequence with loops, but that a greedy graph-based heuristic offers excellent performance and runtime. We thus recommend its use for removing loops at prediction time over the more computationally demanding integer programming and list Viterbi algorithms.

2 TRAJECTORY & PATH RECOMMENDATION

We now formalise the problem of interest and outline its challenges.

2.1 Trajectory recommendation

Fix some set $\mathcal{P}$ of points-of-interest (POIs) in a city. A *trajectory*[1] is any sequence of POIs, possibly containing loops (repeated POIs). In the *trajectory recommendation* problem, we are given as input a training set of historical tourists' trajectories. From this, we wish to design a *trajectory recommender*, which accepts a *trajectory query* $\mathbf{x} = (s, l)$, comprising a start POI $s \in \mathcal{P}$, and trip length $l > 1$, and produces one or more sequences of l POIs starting from s.

[1] In graph theory, this is also referred to as a walk.

Formally, let $\mathcal{X} \doteq \mathcal{P} \times \{2, 3, \ldots\}$ be the set of possible queries, $\mathcal{Y} \doteq \bigcup_{l=2}^{\infty} \mathcal{P}^l$ be the set of all possible trajectories, and for fixed $\mathbf{x} \in \mathcal{X}$, $\mathcal{Y}_\mathbf{x} \subset \mathcal{Y}$ be the set of trajectories that conform to the constraints imposed by $\mathbf{x}$, i.e. if $\mathbf{x} = (s, l)$ then $\mathcal{Y}_\mathbf{x} = \{\mathbf{y} \in \mathcal{P}^l \mid y_1 = s\}$. Then, the trajectory recommendation problem has:

INPUT: training set $\left\{\left(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\right)\right\}_{i=1}^{n} \in (\mathcal{X} \times \mathcal{Y})^n$
OUTPUT: a trajectory recommender $r \colon \mathcal{X} \to \mathcal{Y}$

One way to design a trajectory recommender is to find a (query, trajectory) affinity function $f \colon \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$, and let

$$r(x) \doteq \operatorname*{argmax}_{\mathbf{y} \in \mathcal{Y}_x} \; f(\mathbf{x}, \mathbf{y}). \tag{1}$$

Several choices of f are possible. Chen et al. [3] proposed to use f given by a RankSVM model. While offering strong performance, this has a conceptual disadvantage highlighted in the previous section: it does not model global cohesion, and could result in solutions such as recommending three restaurants in a row.

To overcome this, Chen et al. [4] proposed to use f given by a structured SVM (SSVM), wherein $f(\mathbf{x}, \mathbf{y}) = \mathbf{w}^T \Phi(\mathbf{x}, \mathbf{y})$ for a suitable feature mapping Φ. When this feature mapping decomposes into terms that depend only on adjacent elements in the sequence $\mathbf{y}$ (akin to a linear-chain conditional random field), the optimisation in Equation 1 can be solved with the classic Viterbi algorithm.

2.2 Path recommendation

We argue that the definition of trajectory recommendation is incomplete for a simple reason: in most cases, a tourist will not want to revisit the same POI. Instead, what is needed is to recommend a *path*, i.e. a trajectory that does not have any repeated POIs. Let $\overline{\mathcal{Y}} \subset \mathcal{Y}$ be the set of all possible paths, and for fixed $\mathbf{x} \in \mathcal{X}$, let $\overline{\mathcal{Y}}_\mathbf{x} \subset \overline{\mathcal{Y}}$ be the set of paths that conform to the constraints imposed by $\mathbf{x}$. We now wish to construct a *path recommender* $r \colon \mathcal{X} \to \overline{\mathcal{Y}}$ via

$$r(x) \doteq \operatorname*{argmax}_{\mathbf{y} \in \overline{\mathcal{Y}}_x} \; f(\mathbf{x}, \mathbf{y}). \tag{2}$$

For f given by an SSVM, Equation 2 requires we depart from the standard Viterbi algorithm, as the sequence in Equation 1 may well have a loop.[2] There are two distinct modes of attack available:

(1) seek an approximate solution to the problem, via heuristics that exploit a graph view of Equation 2.
(2) seek an exact solution to the problem, via integer linear programming, or top-K extensions of the Viterbi algorithm.

While Chen et al. [4] suggested the latter exact approaches, they did not formally compare their performance either qualitatively or quantitatively; they did not detail the different top-K extensions of the Viterbi algorithm and the connections thereof; and they did not consider approximate approaches.

In the sequel, we thus detail the above approaches in more detail. Figure 1 gives a schematic overview of these algorithms.

[2]In SSVMs, this issue also arises during training [4], but we focus here only on the prediction problem. See §7 for additional comments.

3 GRAPH-BASED HEURISTICS

To design approximations, it will be useful to view Equation 2 as a graph optimisation problem for the case of structured SVMs with pairwise potentials. Here, the score $f(\mathbf{x}, \mathbf{y})$ can be decomposed as

$$f(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^{|\mathbf{y}|} \alpha(y_k) + \sum_{k=1}^{|\mathbf{y}|-1} \beta(y_k, y_{k+1}) \tag{3}$$

for suitable unary and pairwise potentials α, β [4]. Consider then a complete graph where the nodes are POIs, each node $p \in \mathcal{P}$ has a score $\alpha(p)$, and each edge $(p, p') \in \mathcal{P}^2$ has a score $\beta(p, p')$. Then, Equation 2 is equivalently a problem of selecting l nodes whose total sum of node and edge scores maximises Equation 3. We now look at ways of approximately solving this selection problem.

3.1 Heuristic loop elimination

Perhaps the simplest approximate solution to Equation 2 is to merely remove the first loop occurring in the standard Viterbi solution (Equation 1). That is, if the Viterbi solution is $(y_1, \ldots, y_l)$, return the sub-sequence $(y_1, \ldots, y_{i-1})$ for the first index i where y_i appears in this sub-sequence. This has complexity dominated by the Viterbi algorithm, *viz.* $\mathcal{O}(l \cdot |\mathcal{P}|^2)$ for an input query $\mathbf{x} = (s, l)$.

From a graph perspective, this approach makes the directed subgraph induced by the Viterbi solution acyclic by breaking the first cycle-inducing edge. This is sensible if sequences never escape the first cycle, i.e. after the first repeated POI, there is no new POI. We have indeed found this to be the case in our problem; more generally, the problem of removing cycles is a special case of the (NP-hard) minimum feedback arc-set problem [7, pg. 192].

This algorithm is appealing in its simplicity, but has at least two drawbacks. First, it makes the questionable assumption that Equation 2 is solvable from the standard Viterbi solution alone. Second, the solution violates the length constraint of the path recommendation problem. As a remedy, we can request the Viterbi algorithm to return a sequence of longer length $l' \in \{l, l+1, \ldots, |\mathcal{P}|\}$, and pick the (smallest) l' for which the predicted length is closest to l.

3.2 Greedy path discovery

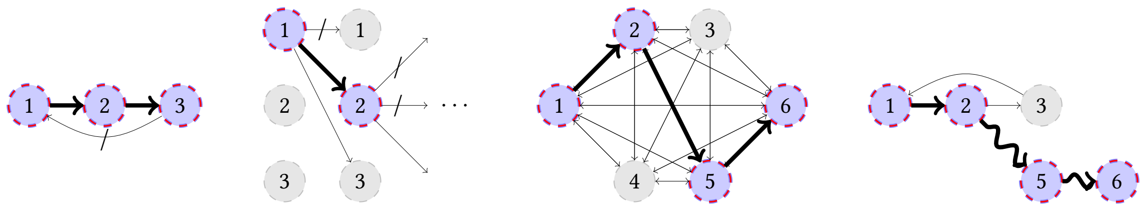
In light of the graph-based view, a natural approach to approximately solve Equation 2 is a greedy algorithm. Suppose we have already determined a partial path comprising distinct POIs $y_1, \ldots, y_k$. Then, we can select the next candidate POI y_{k+1} as the node that iteratively optimises Equation 3, subject to the constraint that it is distinct from all other nodes in the current path; formally, we pick

$$y_{k+1} = \operatorname*{argmax}_{p \in \mathcal{P} - \{y_1, \ldots, y_k\}} \; \alpha(p) + \beta(y_k, p).$$

This algorithm is faster than the above heuristic, with $\mathcal{O}(l \cdot |\mathcal{P}|)$ complexity, but similarly has an unclear performance guarantee.

4 INTEGER LINEAR PROGRAMMING & TSP

One way to exactly solve Equation 2 is to observe its similarity to the travelling salesman problem (TSP). Indeed, if the length $l = |\mathcal{P}|$, so that every POI is restricted to be visited once, then we exactly get the TSP problem; however, for smaller l, we require visiting only a *subset* of POIs, which is not a vanilla instance of TSP.



(a) LoopElim (§3.1). (b) Greedy (§3.2). (c) ILP (§4). (d) List Viterbi (§5).

Figure 1: Schematics of different algorithms to return a loop-free prediction. Nodes such as 1 are selected by the algorithm, with thick edges such as $\longrightarrow$ denoting the sequence ordering. LoopElim removes the loop from the Viterbi solution (here the POI sequence $(1, 2, 3, 1)$), possibly returning a path of shorter length than requested; Greedy incrementally selects POIs which have not been selected before, and locally maximises the sub-path score; ILP solves an integer linear program to find the optimal length l path in a complete graph over POIs; ListViterbi finds where the second-best sequence diverges from the standard Viterbi sequence ($(1, 2, 3, 1)$ as before); if not loop-free, it finds where the third-best diverges from the second-best, *etc.*

Nonetheless, we may take inspiration from methods to solve the TSP to attack our problem. In particular, we can formulate the trip recommendation problem as an *integer linear program* (*ILP*), as often done for the TSP [16]. Formally, given a starting location s and the required trip length l, we find the best possible path via [4]

$$\begin{aligned}
&\max_{\mathbf{u},\mathbf{v}} \sum_{k=1}^{m} \alpha(p_k) \cdot \sum_{j=1}^{m} u_{jk} + \sum_{j,k=1}^{m} u_{jk} \cdot \beta(p_j, p_k) \\
&s.t. \sum_{k=2}^{m} u_{1k} = 1,\ z(\mathbf{u})_1 = 0,\ z(\mathbf{u})_i \in \{0,1\} \qquad (\forall i \in \{2, \cdots, m\}) \\
&\quad \sum_{j,k=1}^{m} u_{jk} = l - 1,\ \sum_{j=1}^{m} u_{jj} = 0,\ \sum_{j=1}^{m} u_{ji} \le 1 \qquad (\forall i \in \{2, \cdots, m\}) \\
&\quad v_j - v_k + 1 \le (m-1)(1 - u_{jk}) \qquad (\forall j,k \in \{2, \cdots, m\}).
\end{aligned}$$

Here, we index POIs such that $s = p_1$ for brevity. The binary u_{jk} are true iff we visit p_k immediately after visiting p_j; the integer v_j track the rank of p_j; and $z(\mathbf{u})_i \doteq \sum_{j=1}^{m} u_{ji} - \sum_{k=1}^{m} u_{ik}$ indicates whether we end up at p_i. The constraints ensure we output a path of exactly l POIs; the last constraint in particular ensures we do not have (disjoint) cycles, as per Miller et al. [14]. By reading off the values of u_{jk}, we can determine the predicted sequence.

An off-the-shelf solver (e.g. `Gurobi`) may be used on the ILP. While ILPs have worst case exponential complexity, such solvers are highly optimised and thus make many problem instances tractable.

The above ILP implicitly solves both the problem of selecting the set of l POIs to recommend, and the problem of ordering them. We could fix the set of POIs to recommend, e.g. by using the POIs in the Viterbi solution, and then find the optimal ordering of these nodes. However, we have found this to have only modest improvement over the loop elimination heuristic of the previous section.

5 TOP-K SEQUENCES USING LIST VITERBI

The Viterbi algorithm finds the best scoring sequence in Equation 1, which may have loops. To find the best sequence without loops in Equation 2, one can apply a *list Viterbi algorithm* to find not just the single best sequence, but rather the top K best sequences. By definition, the first such sequence that is loop free must be the highest scoring sequence that does not have loops. We detail existing approaches to solve the list Viterbi problem.

5.1 Parallel and serial list Viterbi algorithms

At a high level, there are two approaches to extend the Viterbi algorithm to the top-K setting. The first approach is to keep track, at each state, of the top K sub-sequences that end at this state; these are known as *parallel list Viterbi* algorithms. Such algorithms date back to at least Forney [6], but have the disadvantage of imposing a non-trivial memory burden. Further, they are not applicable as-is in our setting: we do *not* know in advance what value of K is suitable, since we do not know the position of the best loop-free sequence.

The second approach is to more fundamentally modify how one selects paths; these are known as *serial list Viterbi* algorithms. There are at least two such well-known proposals in the context of hidden Markov models (HMMs). In the signal processing community, Seshadri and Sundberg [19] proposed an algorithm that keeps track of the "next-best" sequence terminating at each state in the current list of best sequences. In the AI community, Nilsson and Goldberger [15] proposed an algorithm that cleverly partitions the search space into subsets of sequences that share a prefix with the current list of best sequences. While derived in different communities, these two algorithms are in fact only superficially different, as we now see.

5.2 Relating serial list Viterbi algorithms

The connection between the two list Viterbi algorithms is easiest to see when finding the second-best sequence for an HMM. Suppose we have an HMM with states S_t, observations O_t, transitions $a(i,j) = \Pr(\mathrm{S}_{t+1} = j \mid \mathrm{S}_t = i)$, and emissions $b(i,k) = \Pr(\mathrm{O}_t = k \mid \mathrm{S}_t = i)$. Suppose $s^*_{1:T}$ is the most likely length T sequence given observations $\mathrm{O}_{1:T}$, and $\delta(j,t)$ is the value of the best sequence up till position t ending at state j as computed by the Viterbi algorithm.

Our interest is in finding the second-best sequence with value $M \doteq \max_{\mathrm{S}_{1:T} \neq s^*_{1:T}} \Pr(\mathrm{S}_{1:T}, \mathrm{O}_{1:T})$. Seshadri and Sundberg [19] observed that $M = \bar{\delta}_{T+1}$, where $\bar{\delta}_t$ has a Viterbi-like recurrence

$$\bar{\delta}_t \doteq [\![t > 0]\!] \cdot \max \begin{cases} \max_{i \neq s^*_{t-1}} \delta(i, t-1) \cdot a(i, s^*_t) \cdot b(s^*_t, \mathrm{O}_t) \\ \bar{\delta}_{t-1} \cdot a(s^*_{t-1}, s^*_t) \cdot b(s^*_t, \mathrm{O}_t). \end{cases} \tag{4}$$

Intuitively, $\bar{\delta}_t$ finds the value of the second-best sequence that merges with the best sequence by at least time t.

Dataset	# Traj	# POIs	# Queries
Glasgow	351	25	64
Osaka	186	26	47
Toronto	977	27	99

Table 1: Statistics of trajectory datasets: the number of trajectories (# Traj), POIs (# POIs), queries (# Queries). Note that a distinct query may be associated with multiple trajectories.

Nilsson and Goldberger [15] observed that $M = \max_t \widehat{\rho}_t$, where

$$\widehat{\rho}_t \doteq \max_{i \neq s_t^*} \max_{S_{t+1:T}} \Pr(S_{1:t-1} = s^*_{1:t-1}, S_t = i, S_{t+1:T}, O_{1:T}).$$

Intuitively, $\widehat{\rho}_t$ finds the value of the second-best sequence that first deviates from the best sequence exactly at time t. One can compute $\widehat{\rho}_t$ using $\eta_{i,j,t} \doteq \max_{S\colon S_{t-1}=i, S_t=j} \Pr(S_{1:T}, O_{1:T})$ [15], which in turn can be computed from the "backward" analogue of δ.

To connect the two approaches, by unrolling the recurrence in Equation 4, and by definition of δ, we have $M = \max_t \widehat{\mu}_t$ where

$$\begin{aligned}\widehat{\mu}_t &\doteq \left[\prod_{k=t+2}^{T} a(s^*_{k-1}, s^*_k) \cdot b(s^*_k, O_k)\right] \cdot \max_{i \neq s_t^*} \delta(i,t) \cdot a(i, s^*_{t+1}) \cdot b(s^*_{t+1}, O_{t+1}) \\ &= \max_{i \neq s_t^*} \max_{S_{1:t-1}} \Pr(S_{1:t-1}, S_t = i, S_{t+1:T} = s^*_{t+1:T}, O_{1:T});\end{aligned}$$

i.e., the same quantities are computed, except that the former fixes the suffix of the candidate sequence, while the latter fixes the prefix. A similar analysis holds in the case of finding the Kth best sequence.

The complexity of either of the above algorithms is $\mathcal{O}(l \cdot |\mathcal{P}|^2 + l \cdot |\mathcal{P}| \cdot K + l \cdot K \cdot \log(l \cdot K))$ [15]. This is tractable, but we emphasise that the smallest K guaranteeing we obtain a path is unknown *a-priori*.

6 EMPIRICAL COMPARISON

We now empirically compare the methods discussed above, to get a firmer sense of their tradeoffs. We focus on path recommendation tasks where an SSVM is learned to score sequences. (We refer the reader to Chen et al. [4] for a detailed comparison of SSVM to other baselines, e.g. RankSVM.) We then apply each of the above methods to (approximately) solve the inference problem of Equation 2.

6.1 Experimental setup

Following Chen et al. [3, 4], we work with data extracted from Flickr photos for the cities of Glasgow, Osaka and Toronto [3, 11]. Each dataset comprises of a list of trajectories as visited by various Flickr users. Table 1 summarises the statistics of each dataset.

To produce a recommendation, we use the aforementioned inference methods (named LoopElim(++), Greedy, ILP, ListViterbi) as well as standard inference (Viterbi). LoopElim processes the Viterbi solution for the query length l, while LoopElim++ processes the solutions for all longer lengths l', as described in §3.1.

We evaluate each algorithm using leave-one-query-out cross validation, i.e. in each round, we hold out all trajectories for a distinct query **x** in the dataset. To measure performance, we use the **F$_1$ score on points** [11], which computes F$_1$-score on the predicted versus seen points without considering their relative order, and the **F$_1$ score on pairs** [3], which computes the F$_1$-score on all ordered pairs in the predicted versus ground truth sequence.

6.2 Results and discussion

We now address several key questions relating to the tradeoffs of the various methods considered.

How often does the top-scoring sequence have loops? It is first of interest to confirm that even for the powerful SSVM model, the top-scoring sequence as found by the Viterbi algorithm often contain loops. Indeed, we find that on the (Osaka, Glasgow, Toronto) datasets, the top-scoring sequence for (23.9%, 31.2%, 48.5%) respectively of all queries have loops.

How important is it to remove loops? Having confirmed that loops in the top-scoring sequence are an issue, it is now of interest to establish that removing such loops during prediction is in fact important. This is confirmed in Tables 2 – 3, where we see that there can be as much as a **17%** improvement in performance over the Viterbi baseline. These improvements are over all queries, including those where the Viterbi algorithm does not have loops. Restricting to those queries where there are loops, Tables 4b – 5b show that the improvements are dramatic, being as high as **50%**.

How reliably can LoopElim(++) **get the desired length?** Recall that LoopElim and its variant LoopElim++ may result in a path of the wrong length. Figure 2 shows that for a significant fraction of queries, these algorithms will output a different length path to that specified in the query, and are thus not suitable if we strictly enforce a length constraint. Of the two, LoopElim++ outputs more trajectories of the correct length, as per design.

How reliably can LoopElim(++) **predict a good path?** Assuming one can overlook LoopElim(++) producing a path of possibly incorrect length, it is of interest as to how well they perform. Tables 2 – 3 show that the heuristic often grossly underperforms compared to the exact ILP and ListViterbi approaches. (Being exact, the latter methods have nearly identical accuracy, with occasional differences owing to ties.) Curiously, LoopElim++, while producing paths of length closer to the original, actually performs slightly *worse* than the naïve LoopElim on the larger Toronto dataset.

How reliably can Greedy **predict a good path?** Unlike LoopElim(++), the Greedy method is guaranteed to produce a path of the correct length. Surprisingly, it also performs very well compared to the ListViterbi and ILP methods, offering significant improvements over these methods on the larger Glasgow and Toronto datasets, while being competitive on the smaller Osaka dataset.

How does trajectory length influence accuracy? The above analyses the accuracy over all queries, and over queries where Viterbi outputs a loop. It is of interest to partition the set of queries based on the length of the requested trajectory. Intuitively, we expect that the longer the requested trajectory, the less accurate all methods will fare; this is because longer trajectories imply an exponentially large search space. (Indeed, on Toronto, for a query with length 13, the first **5 million** sequences have loops!)

Figure 3 confirms this intuition: on all datasets, and for all methods, a longer trajectory length implies significantly worse performance in absolute terms. Interestingly, the relative improvements of all methods over Viterbi are either consistent or actually *increase* with longer trajectories; this is reassuring, and justifies the effort spent in removing loops. Of further interest is that the Greedy heuristic remains dominant for longer trajectories on the Glasgow and Toronto datasets.

Table 2: Point F_1 cross-validation scores.

(a) Raw scores.

	LoopElim	LoopElim++	Greedy	ILP	ListViterbi	Viterbi
Osaka	0.635 ± 0.039	0.639 ± 0.038	0.626 ± 0.040	0.638 ± 0.039	0.638 ± 0.039	0.622 ± 0.040
Glasgow	0.718 ± 0.030	0.721 ± 0.030	0.751 ± 0.028	0.741 ± 0.028	0.741 ± 0.028	0.689 ± 0.032
Toronto	0.699 ± 0.027	0.696 ± 0.027	0.756 ± 0.024	0.754 ± 0.023	0.754 ± 0.023	0.651 ± 0.030

(b) Improvement over Viterbi.

	LoopElim	LoopElim++	Greedy	ILP	ListViterbi
Osaka	2.0%	2.7%	0.5%	2.6%	2.6%
Glasgow	4.2%	4.6%	9.0%	7.6%	7.6%
Toronto	7.3%	6.8%	16.0%	15.7%	15.7%

Table 3: Pair F_1 cross-validation scores.

(a) Raw scores.

	LoopElim	LoopElim++	Greedy	ILP	ListViterbi	Viterbi
Osaka	0.369 ± 0.059	0.373 ± 0.059	0.369 ± 0.059	0.375 ± 0.059	0.375 ± 0.059	0.364 ± 0.060
Glasgow	0.480 ± 0.049	0.485 ± 0.049	0.522 ± 0.048	0.506 ± 0.048	0.508 ± 0.048	0.461 ± 0.050
Toronto	0.490 ± 0.041	0.489 ± 0.041	0.543 ± 0.038	0.530 ± 0.037	0.529 ± 0.037	0.463 ± 0.041

(b) Improvement over Viterbi.

	LoopElim	LoopElim++	Greedy	ILP	ListViterbi
Osaka	1.4%	2.4%	1.4%	3.0%	3.0%
Glasgow	4.1%	5.1%	13.1%	9.7%	10.1%
Toronto	5.8%	5.7%	17.2%	14.5%	14.4%

Table 4: Point F_1 cross-validation scores of queries where Viterbi recommendation has a loop.

(a) Raw scores.

	LoopElim	LoopElim++	Greedy	ILP	ListViterbi	Viterbi
Osaka	0.389 ± 0.038	0.408 ± 0.038	0.374 ± 0.046	0.405 ± 0.042	0.405 ± 0.042	0.338 ± 0.034
Glasgow	0.569 ± 0.045	0.577 ± 0.045	0.658 ± 0.044	0.644 ± 0.040	0.644 ± 0.040	0.476 ± 0.040
Toronto	0.540 ± 0.030	0.534 ± 0.030	0.657 ± 0.027	0.654 ± 0.025	0.654 ± 0.025	0.443 ± 0.028

(b) Improvement over Viterbi.

	LoopElim	LoopElim++	Greedy	ILP	ListViterbi
Osaka	15.1%	20.9%	10.8%	19.7%	19.7%
Glasgow	19.4%	21.3%	38.2%	35.2%	35.2%
Toronto	22.1%	20.7%	48.5%	47.8%	47.7%

Table 5: Pair F_1 cross-validation scores of queries where Viterbi recommendation has a loop.

(a) Raw scores.

	LoopElim	LoopElim++	Greedy	ILP	ListViterbi	Viterbi
Osaka	0.088 ± 0.026	0.104 ± 0.026	0.103 ± 0.034	0.112 ± 0.030	0.112 ± 0.030	0.067 ± 0.020
Glasgow	0.261 ± 0.047	0.277 ± 0.052	0.378 ± 0.066	0.345 ± 0.055	0.350 ± 0.054	0.201 ± 0.039
Toronto	0.236 ± 0.031	0.235 ± 0.030	0.352 ± 0.032	0.318 ± 0.024	0.317 ± 0.023	0.180 ± 0.025

(b) Improvement over Viterbi.

	LoopElim	LoopElim++	Greedy	ILP	ListViterbi
Osaka	32.7%	55.4%	54.5%	68.2%	68.2%
Glasgow	29.9%	37.6%	88.1%	71.4%	73.9%
Toronto	30.9%	30.1%	95.0%	76.5%	76.0%

How fast are the various methods? As expected, the heuristic LoopElim and Greedy algorithms have the fastest runtime, being on the order of milliseconds per query even for long trajectories (Figure 4). The Greedy algorithm is the faster of the two, as it does not even require running the standard Viterbi algorithm. The LoopElim++ variant is much slower than LoopElim, as it needs to perform the Viterbi calculation multiple times.

The exact methods are by comparison slower, especially for medium length trajectories. Amongst these methods, for shorter trajectories, the ListViterbi approach is to be preferred; however, for longer trajectories, the ILP approach is faster. The reason for the ListViterbi to suffer at longer trajectories is simply because this creates an exponential increase in the number of available choices, which must be searched through serially. Of interest is that ILP approach has runtime largely independent of the trajectory length. This indicates the branch-and-bound as well as cutting plane underpinnings of these solvers are highly scalable.

Overall, the Greedy algorithm is at least competitive, and often more accurate than exact methods; it is also significantly faster. Thus, for recommending paths, we recommend this algorithm.

7 DISCUSSION AND CONCLUSION

We formalised the problem of eliminating loops when recommending trajectories to visitors in a city, and surveyed three distinct approaches to the problem – graph-based heuristics, list extensions of the Viterbi algorithm, and integer linear programming. We explicated how two ostensibly different approaches to the list Viterbi algorithm [15, 19] are in fact fundamentally identical.

In experiments on real-world datasets, a greedy graph-based heuristic offered excellent performance and runtime. We thus recommend its use for removing loops at prediction time over the more involved integer programming and list Viterbi algorithms.

As a caveat on the applicability of the greedy algorithm, we note that the problem of removing loops also arises during training SSVMs in the loss-augmented inference step [4]. The list Viterbi algorithm has been demonstrated useful in this context; it is unclear whether the same will be true of the approximate greedy algorithm, as it will necessarily lead to sub-optimal solutions.

As future work, it is of interest to extend the greedy algorithm to the top-K evaluation setting of Chen et al. [4], wherein the recommender produces a *list* of paths to be considered. A natural strategy would be to augment the algorithm with a beam search.

Further, the idea of modifying the standard Viterbi inference problem (Equation 1) has other applications, such as ensuring diversity in the predicted ranking. Such problems have been studied in contexts such as information retrieval [1] and computer vision [17], and their study would be interesting in trajectory recommendation. More broadly, investigation of efficient means of ensuring global cohesion – e.g. preventing homogeneous results – is an important direction for the advancement of citizen-centric recommendation.

REFERENCES

[1] Jaime Carbonell and Jade Goldstein. 1998. The Use of MMR, Diversity-based Reranking for Reordering Documents and Producing Summaries *(SIGIR '98).*

[2] Chao Chen, Daqing Zhang, Bin Guo, Xiaojuan Ma, Gang Pan, and Zhaohui Wu. 2015. TripPlanner: Personalized trip planning leveraging heterogeneous crowdsourced digital footprints. *IEEE Transactions on Intelligent Transportation Systems* 16, 3 (2015), 1259–1273.

[3] Dawei Chen, Cheng Soon Ong, and Lexing Xie. 2016. Learning Points and Routes to Recommend Trajectories *(CIKM '16).*

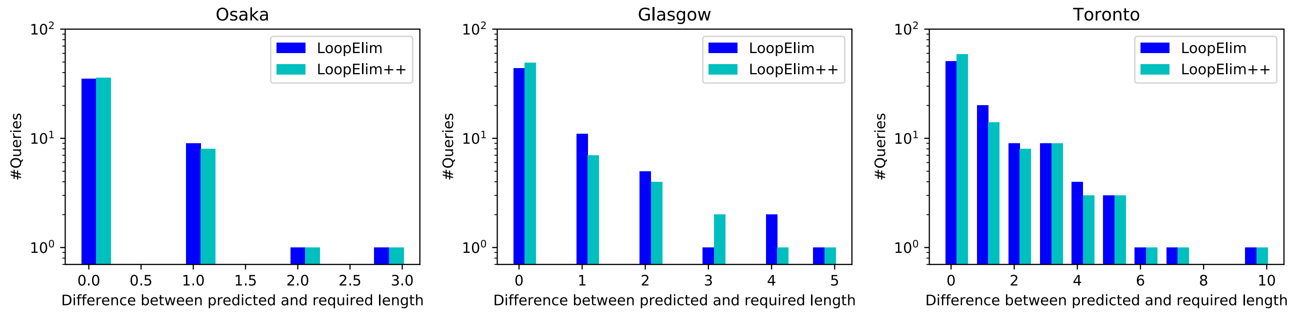


Figure 2: Absolute difference between recommended and required sequence length for LoopElim(++).

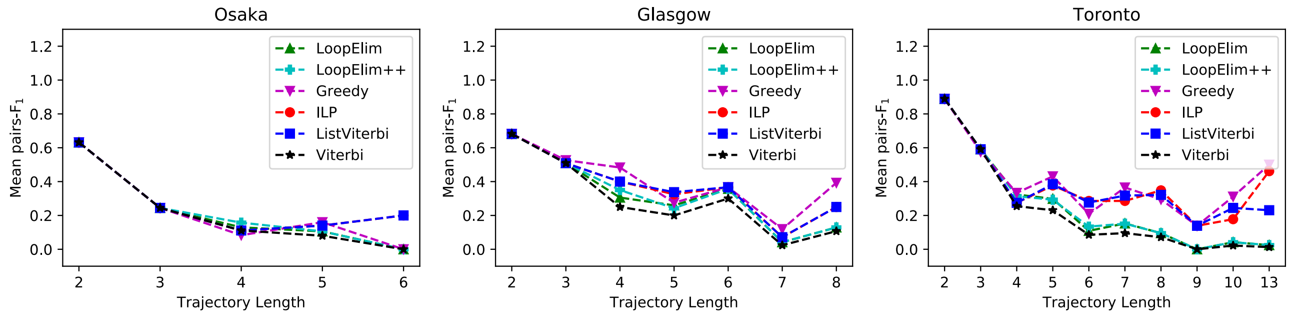


Figure 3: Accuracy versus trajectory length for all inference algorithms.

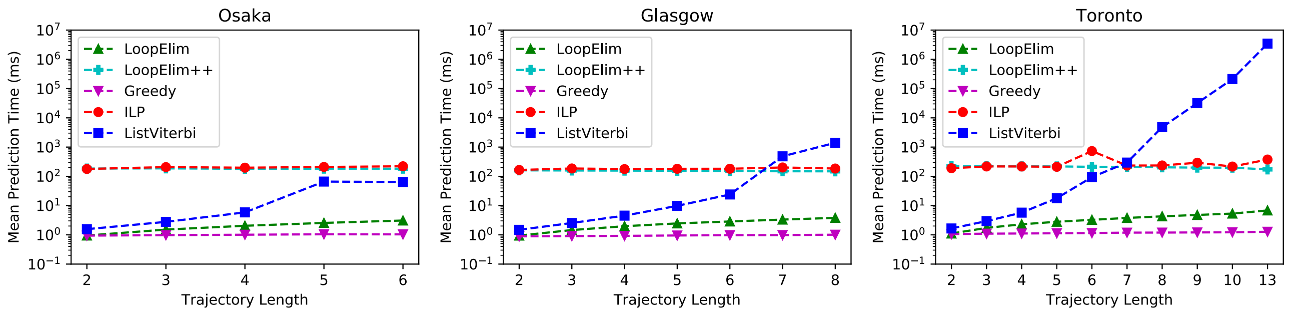


Figure 4: Prediction time versus trajectory length for all inference algorithms.

[4] Dawei Chen, Lexing Xie, Aditya Krishna Menon, and Cheng Soon Ong. 2017. Structured Recommendation. *CoRR* abs/1706.09067 (2017). https:arxiv.org/abs/1706.09067

[5] Chen Cheng, Haiqin Yang, Michael R Lyu, and Irwin King. 2013. Where you like to go next: Successive point-of-interest recommendation *(IJCAI '13)*. 2605–2611.

[6] G. D. Forney. 1973. The Viterbi algorithm. *IEEE* 61, 3 (March 1973), 268–278.

[7] Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.

[8] Aristides Gionis, Theodoros Lappas, Konstantinos Pelechrinis, and Evimaria Terzi. 2014. Customized tour recommendations in urban areas *(WSDM '14)*. 313–322.

[9] Hsun-Ping Hsieh and Cheng-Te Li. 2014. Mining and Planning Time-aware Routes from Check-in Data *(CIKM '14)*. 481–490.

[10] Defu Lian, Cong Zhao, Xing Xie, Guangzhong Sun, Enhong Chen, and Yong Rui. 2014. GeoMF: Joint geographical modeling and matrix factorization for point-of-interest recommendation *(KDD '14)*. 831–840.

[11] Kwan Hui Lim, Jeffrey Chan, Christopher Leckie, and Shanika Karunasekera. 2015. Personalized tour recommendation based on user interests and points of interest visit durations *(IJCAI '15)*.

[12] Eric Hsueh-Chan Lu, Ching-Yu Chen, and Vincent S Tseng. 2012. Personalized trip recommendation with multiple constraints by mining user check-in behaviors *(SIGSPATIAL '12)*. 209–218.

[13] X. Lu, C. Wang, J. M. Yang, Y. Pang, and L. Zhang. 2010. Photo2Trip: Generating Travel Routes from Geo-tagged Photos for Trip Planning *(MM '10)*.

[14] C. E. Miller, A. W. Tucker, and R. A. Zemlin. 1960. Integer Programming Formulation of Traveling Salesman Problems. *J. ACM* 7, 4 (Oct. 1960), 326–329.

[15] Dennis Nilsson and Jacob Goldberger. 2001. Sequentially finding the N-best list in hidden Markov models *(IJCAI '01)*.

[16] Christos H Papadimitriou and Kenneth Steiglitz. 1998. *Combinatorial optimization: algorithms and complexity*. Dover Publications. 308–309 pages.

[17] Dennis Park and Deva Ramanan. 2011. N-best Maximal Decoders for Part Models *(ICCV '11)*. IEEE, 2627–2634.

[18] S. Rendle, C. Freudenthaler, and L. Schmidt-Thieme. 2010. Factorizing Personalized Markov Chains for Next-basket Recommendation *(WWW '10)*. 811–820.

[19] Nambirajan Seshadri and Carl-Erik Sundberg. 1994. List Viterbi decoding algorithms with applications. *IEEE Transactions on Communications* 42, 234 (1994).

[20] Y. Shi, P. Serdyukov, A. Hanjalic, and M. Larson. 2011. Personalized Landmark Recommendation Based on Geotags from Photo Sharing Sites *(ICWSM '11)*.

[21] Quan Yuan, Gao Cong, and Aixin Sun. 2014. Graph-based point-of-interest recommendation with geographical and temporal influences *(CIKM '14)*.

[22] W. Zhang and J. Wang. 2015. Location and Time Aware Social Collaborative Retrieval for New Successive Point-of-Interest Recommendation *(CIKM '15)*. 1221–1230.