# Chapter 5
# Fast Online Recommendation

**Abstract** Based on the spatiotemporal recommender models developed in the previous chapters, the top-$k$ recommendation task can be reduced to an simple task of finding the top-$k$ items with the maximum dot-products for the query/user vector over the set of item vectors. In this chapter, we build effective multidimensional index structures metric-tree and Inverted Index to manage the item vectors, and present three efficient top-$k$ retrieval algorithms to speed up the online spatiotemporal recommendation. These three algorithms are metric-tree-based search algorithm (MT), threshold-based algorithm (TA), and attribute pruning-based algorithm (AP). MT and TA focus on pruning item search space, while AP aims to prune attribute space. To evaluate the performance of the developed techniques, we conduct extensive experiments on both real-world and large-scale synthetic datasets. The experimental results show that MT, TA, and AP can achieve superior performance under different data dimensionality.

**Keywords** Recommendation efficiency · Indexing structure · Top-$k$ query processing · Metric tree · TA

## 5.1 Introduction

The spatiotemporal recommender systems based on latent-class models such as topic models and matrix factorization have demonstrated better accuracy than other methods such as nearest-neighbor models and graph-based models in the previous chapters. However, the online computational cost of finding the top-ranked items for every query/user in the system, once the models have been trained, has been rarely discussed in the academic literature.

In the latent-class models, the predicted ranking score of an item w.r.t. a query/use can often boil down to a dot-product between two vectors representing the query/user and the item, as shown in the previous three chapters. Given a query/user, the straightforward method of generating top-$k$ recommendation needs to compute the ranking scores for all items and select the $k$ ones with highest ranking scores, which is computationally inefficient, especially when the number of items or items' latent attributes

© Springer Science+Business Media Singapore 2016

is large. Moreover, constructing the entire $\#QUERIES \times \#ITEMS$ preference matrix requires heavy computational (and space) resources. For example, the recently published Yelp's Challenge Dataset,[1] which contains 366,000 users and 61,000 business items. Generating the optimal recommendations in this dataset requires over $2.2 \times 10^{11}$ dot-products using a naive algorithm. A 100-dimensional model required 56 h to find optimal recommendations for all the queries. In terms of storage, saving the whole preference matrix requires over 1TB of disk-space. Moreover, this dataset is just a small sample of the actual Yelp dataset and the problem worsens with larger numbers.

The online top-$k$ recommendation can be formulated as: given a query $q$, we aim to find $k$ top-ranked items with highest ranking scores over a set of items $V$, and the ranking score is computed as follows:

$$S(q, v) = \sum_a W(q, a)F(v, a) = \mathbf{q}^T\mathbf{v} = \|\mathbf{q}\|\|\mathbf{v}\|\cos(\triangle_{\mathbf{q},\mathbf{v}}) \propto \|\mathbf{v}\|\cos(\triangle_{\mathbf{q},\mathbf{v}}) \tag{5.1}$$

where $a$ denotes a latent attribute of items, and $W(q, a)$ represents the weight of query $q$ on attribute $a$, and $F(v, a)$ represents the score of item $v$ on attribute $a$. We use $\mathbf{q}$ and $\mathbf{v}$ to denote the vectors of query $q$ and item $v$, respectively, and $\triangle_{\mathbf{q},\mathbf{v}}$ is the angle between the two vectors. Unfortunately, there is not any existing technique to efficiently solve this problem; a linear search over the set of points appears to be the state of the art.

### 5.1.1 Parallelization

The online computational cost of recommendation retrieval can be mitigated by parallelization. One possible way of parallelizing involves dividing the queries/users across cores/machines—each worker can compute the recommendations for a single query/user (or a small set of queries/users). Although the parallelization method can reduce the expensive time cost brought by multiple queries, this form of parallelization does not mitigate the high latency of computing recommendations for a single query/user. Our proposed techniques are orthogonal to parallelization, and can be parallelized to improve the scalability. The MT, TA, and AP techniques presented in this chapter aim reducing the single query latency.

### 5.1.2 Nearest-Neighbor Search

Efficiently finding the top-$k$ recommendation using the dot-product Eq. (5.1) appears to be very similar to the widely studied problem of $k$-nearest-neighbor search in

[1] http://www.yelp.com.sg/dataset_challenge/.

metric spaces [6]. The $k$-nearest-neighbor search problem (in metric space) can be solved approximately with the popular Locality-sensitive hashing (LSH) method [3]. LSH has been extended to other forms of similarity functions (as opposed to the distance as a dissimilarity function) like the cosine similarity [1]. In this section, we show that our problem is different from these existing problems.

The problem of $k$-nearest-neighbor search in metric space is defined as: given a query $q$, the aim is to find $k$ points $v \in V$ with least Euclidean distance to the query point $q$, and the Euclidean distance is computed as follows:

$$\| \mathbf{q} - \mathbf{v} \|^2 = \| \mathbf{q} \|^2 + \| \mathbf{v} \|^2 - 2\mathbf{q}^T\mathbf{v} \propto \| \mathbf{v} \|^2 /2 - \mathbf{q}^T\mathbf{v} \neq -\mathbf{q}^T\mathbf{v} \tag{5.2}$$

Obviously, if all the points in $V$ are normalized to the same length, then the problem of top-$k$ recommendation with respect to the dot-product is equivalent to the problem of $k$-nearest-neighbor search in any metric space. However, without this restriction, the two problems can yield very different answers.

Similarly, the problem of $k$-nearest-neighbor search w.r.t. cosine similarity is defined as: given a query $q$, the aim is to find $k$ points $v \in V$ with maximum cosine similarity for the query $q$, and the cosine similarity is computed as follows:

$$\cos(\triangle_{\mathbf{q},\mathbf{v}}) = \frac{\mathbf{q}^T\mathbf{v}}{\| \mathbf{q} \| \| \mathbf{v} \|} \propto \frac{\mathbf{q}^T\mathbf{v}}{\| \mathbf{v} \|} \neq \mathbf{q}^T\mathbf{v} \tag{5.3}$$

From the above equation, we can see that our problem of top-$k$ recommendation w.r.t. the dot-product is equivalent to the problem of $k$-nearest-neighbor search w.r.t. cosine similarity, if all the points in the set $V$ are normalized to the same length. Under general conditions, the two problems can be very different.

As analyzed above, our problem is not equivalent to the classic $k$-nearest-neighbor search problem in metric space or cosine similarity, thus existing solutions (e.g., the LSH family) to the knn problem cannot be straightforwardly applied to our problem. Actually, our problem is much harder than the knn problem. Unlike the distance functions in metric space, dot-products do not induce any form of triangle inequality. Moreover, this lack of any induced triangle inequality causes the similarity function induced by the dot-products to have no admissible family of locality sensitive hashing functions. Any modification to the similarity function to conform to widely used similarity functions (like Euclidean distance or Cosine-similarity) will create inaccurate results.

Moreover, dot-products lack the basic property of coincidence the self similarity is highest. For example, the Euclidean distance of a point to itself is 0; the cosine-similarity of a point to itself is 1. The dot-product of a point $q$ to itself is $\| \mathbf{q} \|^2$. There can possibly be many other points $v_i$ ($i$ =1, 2,...) in the set $V$ such that $\mathbf{q}^T\mathbf{v_i} > \| \mathbf{q} \|^2$. Without any assumption, the problem of top-$k$ recommendation with respect to the dot-product is inherently harder than the previously addressed similar problems.

**Algorithm 5:** Make-Metric-Tree-Split

```
Input: Item Set $V$;
1 Pick a random item $v \in V$;
2 $\mathbf{a} \leftarrow arg\ \max_{v' \in V} \parallel \mathbf{v} - \mathbf{v'} \parallel$;
3 $\mathbf{c} \leftarrow arg\ \max_{v' \in V} \parallel \mathbf{a} - \mathbf{v'} \parallel$;
4 $\mathbf{w} \leftarrow \mathbf{c} - \mathbf{a}$;
5 $b \leftarrow \frac{-1}{2}(\parallel \mathbf{c} \parallel^2 - \parallel \mathbf{a} \parallel^2)$;
6 return $(\mathbf{w}, b)$;
```

**Algorithm 6:** Make-Metric-Tree

```
Input: Item Set $V$;
Output: Metric Tree $T$;
1 $T.V \leftarrow V$;
2 $T.center \leftarrow mean(V)$;
3 $T.radius \leftarrow arg\ \max_{v \in V} \parallel T.center - \mathbf{v} \parallel$;
4 if $|V| \leq N_0$ then
5     return $T$; //leaf node
6 end
7 else
8     //else split the set;
9     $(\mathbf{w}, \mathbf{b}) \leftarrow$ Make-Metric-Tree-Split(V);
10    $V_l \leftarrow \{v \in V : \mathbf{w}^T \mathbf{v} + b \leq 0\}$;
11    $V_r \leftarrow V - V_l$;
12    $T.left \leftarrow$ Make-Metric-Tree($V_l$);
13    $T.right \leftarrow$ Make-Metric-Tree($V_r$);
14    return $T$;
15 end
```

## 5.2 Metric Tree

In this section, we describe metric tree and develop a novel branch-and-bound algorithm to provide fast top-$k$ recommendations.

Metric trees [4] are binary space-partitioning trees that are widely used for the task of indexing datasets in Euclidean spaces. The space is partitioned into overlapping hyper-spheres (balls) containing the points. We use a simple metric tree construction heuristic that tries to approximately pick a pair of pivot points farthest apart from each other [4], and splits the data by assigning points to their closest pivot. The tree $T$ is built hierarchically and each node in the tree is defined by the mean of the data in that node ($T.center$) and the radius of the ball around the mean enclosing the points in the node ($T.radius$). The tree has leaves of size at most $N_0$. The splitting and the recursive tree construction algorithm is presented in Algorithms 5 and 6.

The tree is space efficient since every node only stores the indices of the item vectors instead of the item vectors themselves. Hence, the matrix for the items is never duplicated. Another implementation optimization is that the vectors in the items' matrix are sorted in place (during the tree construction) such that all the items in the same node are arranged serially in the matrix. This avoids random memory access while accessing all the items in the same leaf node.

### *5.2.1 Branch-and-Bound Algorithm*

Metric trees are used for efficient nearest-neighbor search and are fairly scalable [5]. The search employs a depth-first branch-and-bound algorithm. A nearest-neighbor query is answered by traversing the tree in a depth-first manner—going down the node closer to the query first and bounding the minimum possible distance to items in other branches with the triangle-inequality. If this branch is farther away than the current neighbor candidate, the branch is removed from computation. Since the triangle inequality does not hold for the dot-product, we present a novel analytical upper bound for the maximum possible dot-product of a query vector with points (in this case, items) in a ball. We then employ a similar branch-and-bound algorithm for the purposes of searching for the $k$-highest dot-products (as opposed to the minimum pairwise distance in $k$-nearest-neighbor search).

**Deriving the Upper Bound**. Let $B^r_{v_c}$ be the ball of items centered around $v_c$ with radius $r$. Suppose that $v^*$ is the best possible recommendation in the ball $B^r_{v_c}$ for the query represented by the vector $\mathbf{q}$, and $r^*$ be the Euclidean distance between the ball center $v_c$ and the best possible recommendation $v^*$ (by definition, $r^* \leq r$). Let $\triangle$ be the angle between the vector $\mathbf{v_c}$ and the vector $\mathbf{v^*} - \mathbf{v_c}$, $\triangle_{q,v_c}$ and $\triangle_{v^*,v_c}$ be the angles between the vector $\mathbf{v_c}$ and vectors $\mathbf{q}$ and $\mathbf{v^*}$, respectively, as shown in Fig. 5.1. The distance of $v^*$ from $v_c$ is $(r^* sin\triangle)$ and the length of the projection of $v^*$ onto $v_c$ is $\| \mathbf{v_c} \| + r^* cos\triangle$. Therefore, we have:

$$\| \mathbf{v^*} \| = \sqrt{(\| \mathbf{v_c} \| + r^* \cos \triangle)^2 + (r^* \sin \triangle)^2} \tag{5.4}$$

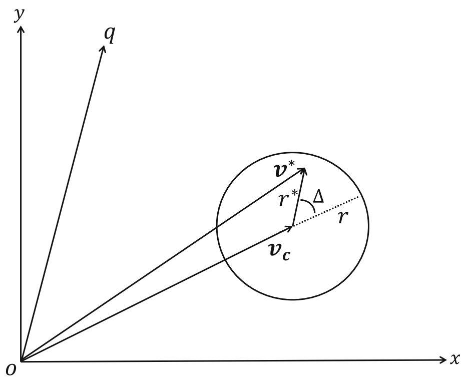

**Fig. 5.1** Bounding with a ball

**Algorithm 7:** Search-Metric-Tree

```
Input: Item Tree Node $T$, query $q$;
if $q.ub < \mathbf{q}^T\mathbf{T.center} + T.radius\cdot \| \mathbf{q} \|$ then
    if $isLeaf(T)$ then
        for $v \in T.V$ do
            if $\mathbf{q}^T\mathbf{v} > q.ub$ then
                $v' \leftarrow arg\ \min_{v_i \in q.candidates} \mathbf{q}^T\mathbf{v_i}$;
                $q.candidates \leftarrow (q.candidates - \{v'\}) \bigcup \{v\}$;
                $q.ub \leftarrow min_{v_i \in q.candidates} \mathbf{q}^T\mathbf{v_i}$;
            end
        end
    end
    else
        //best depth first traversal
        if $\mathbf{q}^T\mathbf{T.left.center} < \mathbf{q}^T\mathbf{T.right.center}$ then
            Search-Metric-Tree($q$, $T.right$);
            Search-Metric-Tree($q$, $T.left$);
        end
        else
            Search-Metric-Tree($q$, $T.left$);
            Search-Metric-Tree($q$, $T.right$);
        end
    end
end
```

$$\cos \triangle_{v^*,v_c} = \frac{\| \mathbf{v_c} \| + r^* \cos \triangle}{\| \mathbf{v^*} \|}, \quad \sin \triangle_{v^*,v_c} = \frac{r^* \sin \triangle}{\| \mathbf{v^*} \|}. \tag{5.5}$$

Let $\triangle_{q,v^*}$ be the angle between the vectors $\mathbf{q}$ and $\mathbf{v^*}$. This gives the following inequality regarding the angle between the query and the best possible recommendation (we assume that the angles lie in the range of $[-\pi, +\pi]$ instead of the usual range $[0, 2\pi]$):

$$|\triangle_{q,v^*}| \geq |\triangle_{q,v_c} - \triangle_{v^*,v_c}|,$$

which implies

$$\cos \triangle_{q,v^*} \leq \cos(\triangle_{q,v_c} - \triangle_{v^*,v_c}) \tag{5.6}$$

since $\cos(\cdot)$ is monotonically decreasing in the range $[0, \pi]$. Using this equality we obtain the following bound for the highest possible affinity between the user and any item within that ball:

$$\max_{v \in B_{v_c}^r} \mathbf{q}^T \mathbf{v} = \mathbf{q}^T \mathbf{v}^* = \| \mathbf{q} \| \| \mathbf{v}^* \| \cos \triangle_{q,v^*} \le \| \mathbf{q} \| \| \mathbf{v}^* \| \cos(\triangle_{q,v_c} - \triangle_{v^*,v_c})$$

where the last inequality follows from Eq. (5.6). Substituting Eqs. (5.4) and (5.5) in the above inequality, we have:

$$\begin{aligned}
\max_{v \in B_{v_c}^r} \mathbf{q}^T \mathbf{v} &\le \| \mathbf{q} \| (\cos \triangle_{q,v_c} (\| \mathbf{v_c} \| + r^* \cos \triangle) + \sin \triangle_{q,v_c} (r^* \sin \triangle)) \\
&\le \| \mathbf{q} \| \max_{\triangle} (\cos \triangle_{q,v_c} (\| \mathbf{v_c} \| + r^* \cos \triangle) + \sin \triangle_{q,v_c} (r^* \sin \triangle)) \\
&= \| \mathbf{q} \| (\cos \triangle_{q,v_c} (\| \mathbf{v_c} \| + r^* \cos \triangle_{q,v_c}) + \sin \triangle_{q,v_c} (r^* \sin \triangle_{q,v_c})) \\
&\le \| \mathbf{q} \| (\cos \triangle_{q,v_c} (\| \mathbf{v_c} \| + r \cos \triangle_{q,v_c}) + \sin \triangle_{q,v_c} (r \sin \triangle_{q,v_c}))
\end{aligned}$$

The second inequality comes from the definition of maximum, and the next equality comes from maximizing over $\triangle$ giving us the optimal value for $\triangle = \triangle_{q,v_c}$. The last inequality follows the $r^* \le r$. Simplifying the final inequality gives us the following upper bound:

$$\max_{v \in B_{v_c}^r} \mathbf{q}^T \mathbf{v} \le \mathbf{q}^T \mathbf{v_c} + r \| \mathbf{q} \| . \tag{5.7}$$

**The Retrieval Algorithm**. Using this upper bound in Eq. (5.7) for the maximum possible dot-product, we present the depth-first branch-and-bound algorithm to search for the $k$-highest dot-products in Algorithm 7. In the algorithm, the object $q.candidates$ contains the set of current best $k$ candidate items and $q.ub$ denotes the lowest affinity between the query and its current best candidates. The algorithm begins at the root of the tree of items. At each subsequent step, the algorithm is at a tree node. Using the bound in Eq. (5.7), the algorithm checks if the best possible item in this node is any better than the current best candidates for the query. If the check fails, this branch of the tree is not explored any more. Otherwise, the algorithm recursively traverses the tree, exploring the branch with the better potential candidates in a depth-first manner. If the node is a leaf, the algorithm just finds the best candidates within the leaf with the simple naive search. This algorithm ensures that the exact solution (i.e., the best candidates) is returned by the end of the algorithm.

## 5.3 TA-Based Algorithm

The straightforward method of generating the top-$k$ items needs to compute the ranking scores for all items according to Eq. (5.1), which is computationally inefficient, especially when the number of items becomes large. To speed up the process of producing recommendations, we extend the Threshold-based Algorithm (TA) [7, 8], which is capable of finding the top-$k$ results by examining the minimum number of items.

**Algorithm 8:** Threshold-based algorithm

```
Input: An item set $V$, a query $q$ and $A$ ranked lists $L_a$;
Output: List $L$ with all the $k$ highest ranked items;
1  Initialize priority lists $PQ$, $L$ and the threshold score $S_{Ta}$;
2  for $a = 1$ to $A$ do
3      $v = L_a.getfirst()$;
4      Compute $S(q, v)$ according to Eq. (5.1);
5      $PQ.insert(a, S(q, v))$;
6  end
7  Compute $S_{Ta}$ according to Eq. (5.8);
8  while true do
9      $nextListToCheck = PQ.getfirst()$;
10     $PQ.removefirst()$;
11     $v = L_{nextListToCheck}.getfirst()$;
12     $L_{nextListToCheck}.removefirst()$;
13     if $v \notin L$ then
14         if $L.size() < k$ then
15             $L.insert(v, S(q, v))$;
16         end
17         else
18             $v' = L.get(k)$;
19             if $S(q, v') > S_{Ta}$ then
20                 break;
21             end
22             if $S(q, v') < S(q, v)$ then
23                 $L.remove(k)$;
24                 $L.insert(v, S(q, v))$;
25             end
26         end
27     end
28     if $L_{nextListToCheck}.hasMore()$ then
29         $v = L_{nextListToCheck}.getfirst()$;
30         Compute $S(q, v)$ according to Eq. (5.1);
31         $PQ.insert(nextListToCheck, S(q, v))$;
32         Compute $S_{Ta}$ according to Eq. (5.8);
33     end
34     else
35         break;
36     end
37 end
```

We first precompute the ordered lists of items, where each list corresponds to a latent attribute learned by the latent-class models. For example, given $A$ latent attributes, we will compute $A$ lists of sorted items (i.e., inverted indices), $L_a$, $a \in \{1, 2, \dots, A\}$, where items in each list $L_a$ are sorted according to $F(v, a)$. Given a query $q$, we run Algorithm 8 to compute the top-$k$ items from the $A$ sorted lists and return them in the priority list $L$. As shown in Algorithm 8, we maintain a priority list $PL$ of the $A$ sorted lists where the priority of a list $L_a$ is determined by the ranking score (i.e., $S(q, v)$) of the first item $v$ in $L_a$ (Lines 2–6). In each iteration, we select the most promising item (i.e., the first item) from the list that has the highest priority in $PL$ and add it to the resulting list $L$ (Lines 9–16). When the size of $L$ is no less than $k$, we will examine the $k$th item in the resulting list $L$. If the ranking score of the $k$th item is higher than the *threshold score* (i.e., $S_{Ta}$), the algorithm terminates early without checking any subsequent items (Lines 18–21). Otherwise, the $k$th item $v'$ in $L$ is replaced by the current item $v$ if $v$'s ranking score is higher than that of $v'$ (Lines 22–25). At the end of each iteration, we update the priority of the current list as well as the threshold score (Lines 28–33).

Equation (5.8) illustrates the computation of the threshold score, which is obtained by aggregating the maximum $F(v, a)$ represented by the first item in each list $L_a$ (i.e., $\max_{v \in L_a} F(v, a)$). Consequently, it is the maximum possible ranking score that can be achieved by the remaining unexamined items. Hence, if the ranking score of the $k$th item in the resulting list $L$ is higher than the threshold score, $L$ can be returned immediately because no remaining item will have a higher ranking score than the $k$th item.

$$S_{Ta} = \sum_{a=1}^{A} W(q, a) \max_{v \in L_a} F(v, a) \tag{5.8}$$

### 5.3.1 Discussion

Being different from the metric tree-based algorithm, the TA-based algorithm requires the ranking function defined in Eq. (5.1) to be monotone given a query. Both the ranking functions in the nonnegative matrix factorization models and the probabilistic generative models developed in the previous three chapters meet this requirement, since the query weight on each attribute $W(q, a)$ is nonnegative in these models. It is easy to understand that Algorithm 8 is able to correctly find the top-$k$ items if the ranking function $S(q, v)$ defined in Eq. (5.1) is monotone. Below, we will prove it formally.

**Theorem 5.1** *Algorithm 8 is able to correctly find the top-k items if the ranking function $S(q, v)$ defined in Eq. (5.1) is monotone.*

*Proof* Let $L$ be a ranked list returned by Algorithms 8 which contains the $k$ spatial items that have been seen with the highest ranking scores. We only need to show that every item in $L$ has a ranking score at least as high as any other item $v$ not in $L$. By definition of $L$, this is the case for each item $v$ that has been seen in running Algorithm 8. So assume that $v$ was not seen, and the score of $v$ in each attribute $a$ is $F(v, a)$. For each ranked list $L_a$, let $\widetilde{v}_a$ be the last item seen in the list $L_a$. Therefore, $F(v, a) \leq F(\widetilde{v}_a, a)$, for every $a$. Hence, $S(q, v) \leq S_{Ta}$ where $S_{Ta}$ is the threshold score. The inequality $S(q, v) \leq S_{Ta}$ holds because of the monotonicity of the ranking function $S(q, v)$ defined in Eq. (5.1). But by definition of $L$, for every $v'$ in $L$ we have $S(q, v') \geq S_{Ta}$. Therefore, for every $v'$ in $L$ we have $S(q, v') \geq S_{Ta} \geq S(q, v)$, as desired.

Besides, Algorithm 8 has another nice property that it is instance optimal with accessing the minimum number of items, and no deterministic algorithm has a lower optimality ratio [2]. We use the word "optimal" to reflect the fact that Algorithm 8 is best deterministic algorithm. Intuitively, instance optimality corresponds to optimality in every instance, as opposed to just worst case or the average case. There are many algorithms that are optimal in a worst-case sense, but are not instance optimal.

Below, we will investigate the instance optimality of Algorithm 8 by an intuitive argument. If $P$ is an algorithm that stops earlier than Algorithm 8 in a certain case, before $P$ finds $k$ items whose ranking score is at least equal to the threshold score $S_{Ta}$, then $P$ must make a mistake, since the next unseen item $v$ might have a ranking score equal to $F(\widetilde{v}_a, a)$ in each attribute $a$, and hence have ranking score $S(q, v) = S_{Ta}$. This new item, which $P$ has not even seen, has a higher ranking score than some item in the top-$k$ list that was output by $P$, and so $P$ erred by stopping too soon.

## 5.4 Attribute-Pruning Algorithm

Both metric-tree and TA algorithms focus on pruning item search space, and they cannot reduce the time cost of computing the ranking score for a single item. Moreover, they cannot adapt to the high-dimension data, i.e., the number of latent attributes is large. When the dimensionality of items is high (e.g., $A > 500$), the tree index structures and tree-based search algorithms (e.g., metric-tree and R-tree) will lose their pruning ability, as analyzed in [5], which is also validated in our experiments. As for TA, it needs to frequently update the threshold for each access of sorted lists and to maintain the dynamic priority queue of sorted lists. These extra computations reduce down the efficiency of TA when the dimensionality is high.

To overcome the curse of dimensionality and speed up the online recommendation, we propose an efficient algorithm to prune the attribute space and facilitate fast computation of the ranking score for a single POI, inspired by TA algorithm [8] and Region Pruning strategy [9]. Our algorithm is based on three observations that (1) a query $q$ only prefers a small number of attributes (i.e., the sparsity of query preferences) and the query weights on most attributes are extremely small; (2) items with high values on these preferred attributes are more likely to become the top-$k$ recommendation results; and (3) the attribute values of most items also exhibit sparsity, i.e., each POI has significant values for only a handful of attributes.

The above three observations indicate that only when a query prefers an attribute and the item has a high value on that attribute, will the score $W(q, a)F(v, a)$ contribute significantly to the final ranking score. Thus, we first pre-compute ordered lists of items, where each list corresponds to a latent attribute. For example, given $A$ latent attributes, we will compute $A$ lists of sorted items, $L_{a_j}$, $1 \leq j \leq A$, where items in each list $L_{a_j}$ are sorted according to $F(v, a_j)$. Different from the threshold algorithm (TA) developed in [8], each sorted list $L_{a_j}$ only stores $k$ items with highest $F(v, a_j)$ values instead of all items. Hence, it is space-saving. Besides, for each item $v$, its attributes are preranked offline according to the value of $F(v, a_j)$. Given an

**Algorithm 9:** Attribute Pruning Algorithm

```
Input: An item set $V$, a query $q$ and $A$ ranked lists $L_a$;
Output: Result list $L$ with $k$ highest ranking scores;
1  Initialize $L$ as $\emptyset$;
2  Sort the query attributes by $W(q, a)$;
3  Choose top $m$ attributes satisfying: $\sum_{j=1}^{m} W(q, a_j) > 0.9 \sum_{j=1}^{A} W(q, a_j)$;
4  for $j = 1$ to $m$ do
5      for $v \in L_{a_j}$ and $v \notin L$ do
6          Compute $S(q, v)$ according to Eq. (5.1);
7          if $L.size() < k$ then
8              $L.add(< v, S(q, v) >)$;
9          end
10         else
11             $v' = L.top()$;
12             if $S(q, v) > S(q, v')$ then
13                 $L.removeTop()$;
14                 $L.add(v, S(q, v))$;
15             end
16         end
17     end
18 end
19 for $v \in V$ and $v \notin L$ do
20     $PS = 0$, $PW = 0$, $Skip = false$, and $v' = L.top()$;
21     while there exists attribute $a$ not examined for $v$ do
22         $a = v.nextAttribute()$;
23         $PS = PS + W(q, a)F(v, a)$;
24         $PW = PW + W(q, a)$;
25         if $PS + (\sum_{j=1}^{A} W(q, a_j) - PW)F(v, a) \leq S(q, v')$ then
26             $Skip = true$;
27             $break$;
28         end
29     end
30     if $Skip == false$ then
31         if $S(q, v) > S(q, v')$ then
32             $L.removeTop()$;
33             $L.add(< v, S(q, v) >)$;
34         end
35     end
36 end
37 $L.Reverse()$;
38 Return $L$;
```

online query $q$, we develop a branch and bound algorithm, as shown in Algorithm 9, to prune the search space of the attributes in the computation of the ranking score, i.e., after we have scanned a small number of significant attributes for an item, it may not be necessary to examine the remaining attributes. The algorithm is called AP (Attribute Pruning) and contains two components: *initialization* and *pruning*.

In the initialization component (Lines 1–18), we select $k$ candidate items that are potentially good for recommendation. Specifically, we pick top $m$ attributes which cover most the query's preferences with smallest $m$, i.e., $\sum_{j=1}^{m} W(q, a_j) > \rho \sum_{j=1}^{A} W(q, a_j)$, where $\rho$ is a predefined constant between 0 and 1 (Line 3). In our experiment, AP achieves its best performance for $\rho = 0.9$. For each of the top $m$ attributes $a$, we choose top ranked items from $L_a$ as candidates (Lines 4–18).

In the pruning component (Lines 19–36), we check whether we can avoid traversing unnecessary attributes for item $v$ according to the descending order of $F(v, a)$. Suppose we have traversed attributes $\{a_1, \ldots, a_{i-1}\}$. The partial score we have computed for the traversed attributes is

$$PS(q, v) = \sum_{j=1}^{i-1} W(q, a_j) F(v, a_j).$$

When we explore the $i$th attribute, we compute the upper bound of ranking score for the item $v$ as:

$$UB(q, v) = PS(q, v) + \sum_{j=i}^{A} W(q, a_j) F(v, a_i) \tag{5.9}$$

Because we check the attributes in the descending order of $F(v, a)$, the actual value of $F(v, a)$ for the remaining attributes should be less than the value for the current attribute, i.e., $F(v, a_i)$. Therefore, we have a partial ranking score for the rest of the attributes, which is at most

$$\sum_{j=i}^{A} W(q, a_j) F(v, a_i), \tag{5.10}$$

where $\sum_{j=i}^{A} W(q, a_j)$ is the portion of query preferences for the rest attributes. The upper bound of $\sum_a W(q, a)\ F(v, a)$ for all attributes is $PS(q, v) + \sum_{j=i}^{A} W(q, a_j)\ F(v, a_i)$, which results in Eq. (5.9).

We employ a binary min-heap to implement $L$ so that the top item $v'$ has the smallest ranking score in $L$ (Line 20). If the upper bound is smaller than the ranking score of $v'$, we skip the current item (no need to check the remaining attributes) (Lines 25–28). Otherwise, we continue to check the remaining attributes. If all attributes are examined for the item and the item is not pruned by the aforementioned upper bound, we obtain the full score of the item to compare with $v'$ (Lines 30–35). We remove the item $v'$ and add the current item to the list if its full score is larger than $v'$ (Lines 31–34).

**Note that** our proposed AP algorithm in this section also requires the ranking function $S(q, v)$ to be monotone, e.g., the query weight on each attribute $W(q, a)$ is nonnegative.

## 5.5 Experiments

In this section, we evaluate the top-$k$ recommendation efficiency of metric tree (MT), TA, and attribute pruning (AP) algorithms on both real-world and large-scale synthetic datasets. There are 114,508 users, 62,462 spatial items and 1,434,668 check-ins in the real-world dataset Twitter. Every user has approximately 13 check-ins and every spatial item is associated with 23 check-ins on average. The users' check-ins display the power-law distribution. To keep the sparsity property on the synthetic dataset, we generate a dataset with 23 million check-ins, 1 million spatial items and 1.77 million users to simulate the distribution of the check-ins on the real-world dataset.

### *5.5.1 Experimental Results*

This experiment is to evaluate the efficiency of our proposed online recommendation algorithms MT, TA, and AP on both the real-life and large-scale synthetic datasets. We compare them with a baseline algorithms. The baseline is a brute-force algorithm (BF) that needs to compute a ranking score for each item and selects top-$k$ ones with highest ranking scores. All the online recommendation algorithms were implemented in Java (JDK 1.7) and run on a Windows Server 2008 with 256G RAM.

Figure 5.2 shows the time costs of producing top-10 recommendation on the large-scale synthetic dataset. We control the number of available items to vary from 0.1 million to 1 million to test the scalability, and the dimensionality is set to be 10, 50, 100, 500, 1000, and 1500. From the results, we can observe that these four algorithms have significant performance disparity when the dimensionality increases from 10 to 1500. Obviously, $AP$ exhibits highly desirable scaling characteristics—sub-linear time complexity to both data size and data dimensionality, while other competitor methods, MT and TA, are very sensitive to the data dimensionality, and they perform better than BF only for low-dimensionality setting (e.g., less than 100 dimensionality). The test results also provide important insights to choose online recommendation algorithms: when the data dimensionality is not larger than 10, MT is the best choice; when $10 < A \leq 50$, TA can achieve best performance; and when $A > 50$, we suggest to choose AP algorithm.

To further analyze these algorithms in the high-dimension setting, we test them on the Twitter dataset. Table 5.1 shows the performance with 1500 latent dimensions (i.e., $A = 1500$) and $k$ (i.e., the number of recommendations) set to 1, 5, 10, 15, and 20. A greater value of $k$ is not necessary for the top-$k$ recommendation task. Obviously, the AP algorithm outperforms others significantly and consistently for different number of recommendations. For example, on average the AP algorithm finds the top-10 recommendations from about 62,000 items in 67.57 ms, and achieves 1.65 times faster than the brute-force algorithm (BF).

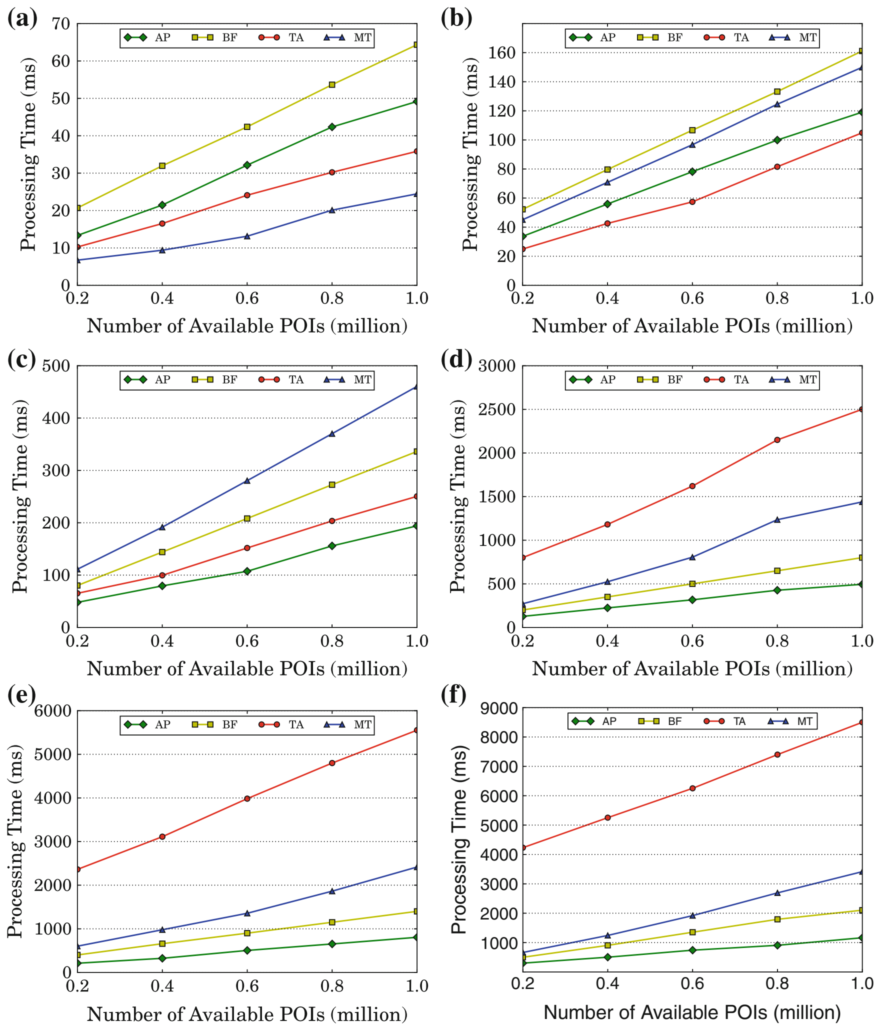

**Fig. 5.2** Recommendation Efficiency on the Synthetic Dataset with Varying Dimensionality (A). **a** A = 10. **b** A = 50. **c** A = 100. **d** A = 500. **e** A = 1000. **f** A = 1500

Specifically, from the results, we observe that: (1) AP outperforms BF significantly, justifying the benefits brought by pruning attribute space. It only needs to access very few attributes for each item to compute its partial score, about 145 attributes on average (that is less than 10 %) for $k = 10$, and 120 attributes for $k = 5$, since AP algorithm takes full advantage of the sparsity of both query and POI vectors. (2) Although the time cost of AP increases with the increasing number of recommendations (i.e., $k$), it is still much lower than that of BF in the recommendation task even when $k = 20$. (3) The time cost of MT is higher than that of the naive linear

**Table 5.1** Recommendation efficiency on twitter dataset

| Methods | Online recommendation time cost (ms) | | | | |
|---|---|---|---|---|---|
| | k=1 | k=5 | k=10 | k=15 | k=20 |
| AP | **50.91** | **61.75** | **67.57** | **72.13** | **75.32** |
| BF | 110.83 | 111.40 | 111.81 | 111.95 | 112.94 |
| MT | 130.90 | 131.46 | 131.87 | 132.39 | 132.99 |
| TA | 880.92 | 1055.68 | 1144.82 | 1221.49 | 1265.58 |

scan method (BF) in this task, although it achieves better performance than BF in the low-dimension setting. This is because MT loses its ability to prune item search space and needs to scan all items in the leaf nodes when the dimensionality is high. (4) The threshold algorithm (TA) performs worse than the brute-force algorithm, since it still needs to access many items (around 40 % of the items on average for $k = 10$ and 35 % for $k = 5$). Moreover, TA needs to frequently update the threshold for each access of sorted lists and to maintain the dynamic priority queue of sorted lists. These extra computations reduce down the efficiency of TA when the dimensionality is high. In summary, although both TA and MT can achieve better performance than BF due to their ability of pruning POI search space when the dimensionality is not very high, they cannot overcome the curse of dimensionality when the items have thousands of attributes. In contrast, the AP algorithm is designed for pruning attribute space, thus it can still achieve superior performance for the setting of high dimensionality.

## 5.6 Summary

In this chapter, we proposed three techniques for efficient spatiotemporal recommendation: (i) metric tree (MT). We index the item vectors in a binary spatial partitioning metric tree and used a simple branch-and-bound algorithm with a novel bounding scheme to prune the item search space. (ii) TA. We precomputes an inverted list for each latent attribute $a$ in which items are sorted according to their values on attribute $a$, and also maintains a priority queue of the inverted lists that controls which inverted list to access in the next. The algorithm has the nice property of terminating early without scanning all items. (iii) Attribute Pruning (AP). Being different from metric tree and TA which focus on pruning item search space, the AP aims to reduce the time cost of computing the ranking score for a single item by pruning the attribute space. We evaluated our algorithms on both real-world and large-scale synthetic datasets, demonstrating the superiority of MT, TA, and AP over the linear-scan algorithm. Moreover, we found that these three techniques show different performances with varying the data dimensionality (i.e., the number of items' latent attributes): when the data dimensionality is not larger than 10, MT is the best choice; when the number of latent attributes is between 10 and 50, TA can achieve best performance; and when the data dimensionality exceeds 50, we suggest to choose the AP.

## References

1. Charikar, M.S.: Similarity estimation techniques from rounding algorithms. In: STOC, pp. 380–388 (2002)
2. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. In: PODS, pp. 102–113 (2001)
3. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: STOC, pp. 604–613 (1998)
4. Koenigstein, N., Ram, P., Shavitt, Y.: Efficient retrieval of recommendations in a matrix factorization framework. In: CIKM, pp. 535–544 (2012)
5. Liu, T., Moore, A.W., Yang, K., Gray, A.G.: An investigation of practical approximate nearest neighbor algorithms. In: NIPS, pp. 825–832 (2004)
6. Shakhnarovich, G., Darrell, T., Indyk, P.L: Nearest-Neighbor Searching and Metric Space Dimensions, pp. 15–59. MIT Press, Cambridge (2006)
7. Yin, H., Cui, B., Chen, L., Hu, Z., Zhou, X.: Dynamic user modeling in social media systems. ACM Trans. Inf. Syst. **33**(3):10:1–10:44 (2015)
8. Yin, H., Cui, B., Sun, Y., Hu, Z., Chen, L.: Lcars: a spatial item recommender system. ACM Trans. Inf. Syst. **32**(3):11:1–11:37 (2014)
9. Zhao, K., Cong, G., Yuan, Q., Zhu, K.: Sar: a sentiment-aspect-region model for user preference analysis in geo-tagged reviews. In: ICDE (2015)