Multi-space Probabilistic Sequence Modeling

Shuo Chen
Cornell University
Dept. of Computer Science
Ithaca, NY, USA
shuochen@cs.cornell.edu

Jiexun Xu
Cornell University
Dept. of Computer Science
Ithaca, NY, USA
jiexunxu@cs.cornell.edu

Thorsten Joachims
Cornell University
Dept. of Computer Science
Ithaca, NY, USA
tj@cs.cornell.edu

ABSTRACT

Learning algorithms that embed objects into Euclidean space have become the methods of choice for a wide range of problems, ranging from recommendation and image search to playlist prediction and language modeling. Probabilistic embedding methods provide elegant approaches to these problems, but can be expensive to train and store as a large monolithic model. In this paper, we propose a method that trains not one monolithic model, but multiple local embeddings for a class of pairwise conditional models especially suited for sequence and co-occurrence modeling. We show that computation and memory for training these multi-space models can be efficiently parallelized over many nodes of a cluster. Focusing on sequence modeling for music playlists, we show that the method substantially speeds up training while maintaining high model quality.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning; I.5.1 [**Pattern Recognition**]: Models

General Terms

Algorithms, Experimentation, Performance

Keywords

Music Playlists, Recommendation, Embedding, Sequences, Parallel Computing

1. INTRODUCTION

Learning methods that embed objects into Euclidean space have become the method of choice for a wide range of problems, ranging from recommendation and image search to playlist prediction and language modeling. Not only do they apply to modeling problems where a feature-vector representation of objects is not available (e.g., movies, users, songs), they actually compute a vectorial representation that can be used as the basis for subsequent modeling steps (e.g., semantic and syntactic language modeling).

While small to medium-scale models can be trained by standard methods, training large-scale models may not be feasible on a single machine. This is especially true for embedding problems that go beyond the Gaussian model of rating prediction. For example, when the embedding is used to model probability distributions over discrete objects like sequences (e.g., playlists [18], words in a sentence[10], purchases [16]) or complex preferences (e.g., as extension to [15]), computation time and memory for storing data and model become a bottleneck.

In this paper, we explore training algorithms for embedding models that execute a distributed fashion, especially for logistic embedding models of sequences and co-occurrences. We formulate an extended logistic model that directly exploits the properties of the data — namely that many dependencies are local even though we are training a global model. By uncovering the locality in the data, we partition the global embedding problem into multiple local embedding problems that are connected through narrow interfaces, which we call *portals*. We show that training this portal model in a distributed fashion can be decomposed into two steps, each of which performs maximum-likelihood optimization.

By deriving this portal model as an explicit probabilistic model for merging multiple local embeddings, the model not only allow more efficient training but also allows efficient prediction in a distributed fashion. Furthermore, the portal model provides understanding for why and when parallel training will be effective. We conduct extensive experiments on probabilistic sequence modeling for music playlist prediction, showing that we can train on hundreds of nodes in parallel without substantial reduction in model fidelity, but in orders of magnitude less time.

2. RELATED WORK

Embedding methods have been long studied and proved to be effective in capturing latent semantics of how items (e.g. words in sentences) interact with each other. These methods only have a linear blowup in parameters as the number of items goes up. For the purpose of this paper, this line of works can be categorized into two classes. The first class [17, 19, 20, 6, 21, 8, 16] defines a score to measure the intensity of any interaction, which is optimized while learning the embedding. The other class [1, 5, 11, 4, 10] explicitly reasons about the embedding under a probabilistic model of the data. Particularly relevant for this paper are those that normalize via a soft-max function to model distributions over discrete items, and that are trained via maximum likelihood.

There are at least two approaches to training these embedding models. First, one can formulate a relaxation of the training problem that can be optimized globally (e.g. a semidefinite programming or singular value decomposition [17, 19, 20]). Second, one can explicitly fix the dimensionality and solve the resulting non-convex objective function to a local optimum. Most popular and generally effective for this second approach are stochastic gradient method [18, 4, 16] that take one interaction at a time to compute and update with local gradients. However, for probabilistic models using the soft-max function computing gradients requires summation over all items. This can be troublesome when the number of total items scales up.

One way to address the growing computational needs arising from large datasets is the use of parallel computation. In particular, there are several works that aim to parallelize training via stochastic gradient methods for both shared-memory [14] and distributed-memory settings [22, 2]. The extension to multi-space embedding models we propose is different from these works. The embedding problem is divided into subproblems in multiple spaces that are only losely coupled, so that they can be solved in an embarrassingly parallel fashion. Furthermore, the model we introduce not only distributes computation, but also reduces that overall amount of computation that is necessary.

3. PROBABILISTIC EMBEDDING MODELS

We would like to first introduce a general family of models that can benefit from the techniques we propose in this paper. Suppose we have n types of items $\mathcal{X} = \bigcup_{i=1}^{n} \mathcal{X}_i$, with all $\mathcal{X}_i$ being disjoint. Each type $\mathcal{X}_i$ contains $|\mathcal{X}_i|$ distinct items $x_1^{(i)}, x_2^{(i)}, \ldots, x_{|\mathcal{X}_i|}^{(i)}$. The training dataset D consists of directional pairwise observations in the form of $(y|x)$, where $x, y \in \mathcal{X}$. For each $x \in \mathcal{X}$, we associate it with a d-dimensional (d is predefined) vector $X(x)$, so that the conditional probability of the pair $(y|x)$ can be modeled as

$$\Pr(y|x) = \prod_{i=1}^{n} \left(\frac{e^{I(X(y),X(x))}}{\sum_{y' \in \mathcal{X}_i} e^{I(X(y'),X(x))}} \right)^{\mathbb{1}^{\{y \in \mathcal{X}_i\}}}. \quad (1)$$

Here $\mathbb{1}^{\{\cdot\}}$ is the indicator function and $I(\cdot,\cdot)$ is the interaction function between two vectors in the d-dimensional space. Common choices include negative Euclidean distance and the inner product. The goal is to learn the collection of all the d-dimensional vectors (or a $\sum_{i=1}^{n} |\mathcal{X}_i|$ by d matrix), which can be done by maximizing the likelihood

$$X = \underset{X \in \Re^{(\sum_{i=1}^{n} |\mathcal{X}_i|) \times d}}{\operatorname{argmax}} \prod_{(y|x) \in D} \Pr(y|x). \quad (2)$$

Several existing works fall into this class of models:

- Logistic Markov Embedding (LME) [18] aims to model sequences for music playlist generation. x and y are consecutive songs in a playlist, and Euclidian distance in the embedding space reflects the probability of a transition from x to y.
- The sphere embedding [10] is proposed for modeling sentence structure in natural language. Such language models are important components in systems for machine translation, speech recognition, etc. Here $(y|x)$ means that word y follows word x.
- Stochastic Neighbor Embedding (SNE) [5] embeds general vectorial data points into low-dimensional space. x and y are data points that belong to the same type, and a directional pair $(y|x)$ exists if x and y are neighbors.
- The conditional model of Co-occurrence Data Embedding (CODE) [4] deals with two types of items, and the interaction is the co-occurrence of two items from different types (e.g. a word appears in a document). $(y|x)$ exists if x and y co-occur, and x is manually designated as the item being conditioned on.

One should note that it is also possible to model joint distributions instead of conditional ones. Following [4], we could let

$$\Pr(x, y) \approx \Pr(y|x)\overline{\Pr}(x), \quad (3)$$

where $\overline{\Pr}(x)$ is estimated empirically from the training set. If symmetry is important, one could also consider

$$\Pr(x, y) \approx \frac{1}{2}(\Pr(y|x)\overline{\Pr}(x) + \Pr(x|y)\overline{\Pr}(y)). \quad (4)$$

In the rest of the paper, we mainly focus on sequence modeling via LME. However, it is possible to extend the ideas to this more general family, as to be discussed in Section 4.5.

3.1 Logistic Markov Embedding

LME [18, 12] was introduced as a probablistic model for learning to generate playlists. Given a collection of songs $\mathcal{S} = \{s_1, ..., s_{|\mathcal{S}|}\}$, a playlist is a sequence of songs from $\mathcal{S}$. We use $p = (p^{[1]}, ..., p^{[k_p]})$ to denote a playlist p of length k_p. We use D to represent a collection of playlists. Given a training sample D of playlists, the goal is to learn a d-dimensional vector for each of the songs in $\mathcal{S}$.

Suppose the vector that represents song s is $X(s)$. The transition probability given the previous song in a playlist p to the next song is modeled as

$$\Pr(p^{[i]}|p^{[i-1]}) = \frac{e^{-||X(p^{[i]})-X(p^{[i-1]})||_2^2}}{\sum_{j=1}^{|\mathcal{S}|} e^{-||X(s_j)-X(p^{[i-1]})||_2^2}} = \frac{e^{-\Delta(p^{[i]},p^{[i-1]})^2}}{Z(p^{[i-1]})}, \quad (5)$$

where $Z(p^{[i-1]})$ denotes the partition function in the denominator, and the distance $||X(s) - X(s')||_2$ is abbreviated by $\Delta(s, s')$. Given this local transition probability, the probability of a playlist is modeled as

$$\Pr(p) = \prod_{i=1}^{k_p} \Pr(p^{[i]}|p^{[i-1]}) = \prod_{i=1}^{k_p} \frac{e^{-\Delta(p^{[i]},p^{[i-1]})^2}}{Z(p^{[i-1]})}. \quad (6)$$

The learning problem is to find the coordinates for each of the songs (they form a $|\mathcal{S}|$ by d matrix X) that maximize the likelihood on a training playlist collection D

$$X = \underset{X \in \Re^{|\mathcal{S}| \times d}}{\operatorname{argmax}} \prod_{p \in D} \prod_{i=1}^{k_p} \frac{e^{-\Delta(p^{[i]},p^{[i-1]})^2}}{Z(p^{[i-1]})}. \quad (7)$$

Since this model only uses order-one Markov dependency, it is convenient to use a transition matrix T to represent the collection of playlists D. The element at the ith row and jth column T_{ij} is the number of transitions from s_i to s_j (We will denote this transition pair as $(s_i \to s_j)$, with s_i called from-song and s_j called to-song) in the playlist collection. T is usually very sparse. Thus it can be stored efficiently via hashing. Using T, the optimization problem can be rewritten as

$$X = \underset{X \in \Re^{|\mathcal{S}| \times d}}{\operatorname{argmax}} \prod_{i=1}^{|\mathcal{S}|} \prod_{j=1}^{|\mathcal{S}|} \left(\frac{e^{-\Delta(s_j, s_i)^2}}{Z(s_i)} \right)^{T_{ij}}. \quad (8)$$

Following the empirical results in [18], it is beneficial to include a popularity boost term $b(s)$ for each of the songs, slightly modifying (5) into

$$\Pr(p^{[i]} | p^{[i-1]}) = \frac{e^{-\Delta(p^{[i]}, p^{[i-1]})^2 + b_{\mathrm{idx}(p^{[i]})}}}{\sum_j e^{-\Delta(s_j, p^{[i-1]})^2 + b_j}}, \quad (9)$$

where $\mathrm{idx}(s)$ returns the index of a song in the song collection (e.g. $\mathrm{idx}(s_j) = j$). Equation (6) - (8) need to be changed accordingly. We call this model boosted-LME and the original model unboosted-LME. For brevity, we use the unboosted-LME for all mathematical derivations in this paper, but use the boosted-LME in the experiments.

3.2 Training and Lack of Scalability

The LME is typically trained using stochastic gradient, with the gradient for the log likelihood of any transition pair $(s_a \to s_b)$ expressed as

$$\frac{\partial l(s_a, s_b)}{\partial X(s_i)} = \mathbb{1}^{\{i=a\}} 2 \left[\overrightarrow{\Delta}(s_a, s_b) - \frac{\sum_{j=1}^{|\mathcal{S}|} e^{-\Delta(s_a, s_j)^2} \overrightarrow{\Delta}(s_a, s_j)}{Z(s_a)} \right] - \mathbb{1}^{\{i=b\}} 2 \overrightarrow{\Delta}(s_a, s_b) + 2 \frac{e^{-\Delta(s_a, s_i)^2} \overrightarrow{\Delta}(s_a, s_i)}{Z(s_a)}, \quad (10)$$

where $\overrightarrow{\Delta}(s_a, s_b)$ denotes $X(s_b) - X(s_a)$. Note that the computation of $Z(s_a)$ involves summing over $|\mathcal{S}|$ terms.

In each iteration of stochastic gradient descent, the algorithm sequentially traverse all the transition pairs $(s_i \to s_j)$ in D. To be more specific, it first picks a from-song s_i, computes and accumulates gradients for the transition pairs that have s_i as from-song, then adds it back to update the embedding and move on to the next from-song. This grouping by from-song allows that gradients for pairs $(s_i \to s_j)$ and $(s_i \to s_{j'})$ can share the computation of $Z(s_i)$. There are $|\mathcal{S}|$ from-songs, and for each from-song, the complexity for computing the partition function is $O(|\mathcal{S}|)$. Thus, the time complexity for each iteration is $O(|\mathcal{S}|^2)$. This causes serious scalability issue, and training LME on dataset with around 75K songs and 15K playlists took more than two weeks even for dimensionality $d = 2$.

3.3 Naive Parallelization

A first approach to speed up training is to parallelize the algorithm. The most natural approach is to perform the gradient computation in parallel. In each iteration, we assign each thread/process an approximately equal number of from-songs, and let it be responsible for the gradients that are associated with those from-songs. When it comes to the implementation of this method, there is a distinction between shared-memory setting and distributed-memory setting.

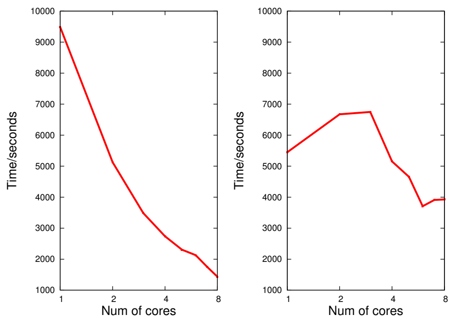

Figure 1: Time for computing the embedding on *yes_big* with respect to number of cores used for shared-memory (left) and distributed-memory (right) parallelization. The difference in runtime for 1 core result from different hardware used in the two experiments.

In the shared-memory parallelization, all threads share the same copy of the parameters (in our case, the embedding matrix X itself) to learn in main memory. A read-write lock ensures consistent access to X. We implemented the shared-memory parallelization with pthread, and tested it on an eight-core machine. A typical time-against-number-of-cores curve is shown in Fig. 1 (left), showing a close to linear speedup. However, the cost of a multi-core CPU goes up superlinearly with respect to the number of cores, and locking X will become a bottleneck as the number of cores increases.

In a distributed-memory architecture, where multiple machines connected via a network are used, it is easier to get a large number of processors. However, the communication overhead is typically larger. We implemented the algorithm using the Message Passing Interface (MPI), again letting each process be in charge of a subset of from-songs. Since each process now has its own copy of the embedding matrix X, we need to introduce checkpoints for communication over network to sync the matrices. At each checkpoint, one master process collects all the accumulated gradient from all the processes, adds them together to update the embedding matrix, and then redistributes the updated embedding matrix to each process. Each checkpoint for communication involved one MPI_Reduce and one MPI_Bcast call. A curve from one run is plotted in Fig. 1 right. The speedup is much less than for the shared-memory implementation, and it is not monotone, let alone linear. In general, network communication creates large overhead and large variability in the runtime.

We conclude that naive parallelization provides only limited benefits and has its own scalability limits. In particular, the naive parallelization does not reduce the total computation needed for training, meaning that even under perfect scaling through parallelization we need to grow the number of processors quadratically for this $O(|\mathcal{S}|^2)$ algorithm. In the next section, we therefore explore a new probabilistic embedding model that not only enables parallelization, but also addresses the $O(|\mathcal{S}|^2)$ scaling.

4. MULTI-SPACE EMBEDDING

The key insight motivating the Multi-space Logistic Markov Embedding (Multi-LME) proposed in the following lies in the locality of the data. Generally, songs only have transition connecting with a small subset of (similar) songs

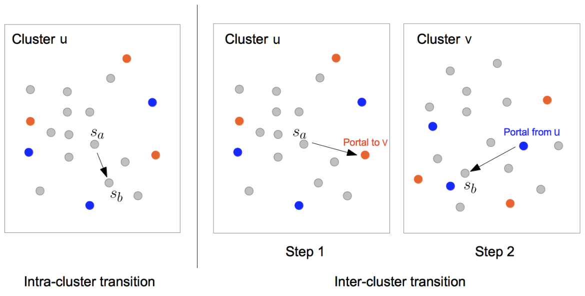

Figure 2: Illustration of transitions under Multi-LME. Gray, blue and orange circles represent real songs, entry portals and exit portals respectively. Left: intra-cluster transition from s_a to s_b in the same cluster C_u. Right: inter-cluster transition from s_a in C_u to s_b in C_v. Two portals $o^{\text{exit}}_{u,v}$ and $o^{\text{entry}}_{v,u}$ are used in the process.

(e.g. songs of the same genre). While we would still like to learn a global transition model, we only need to model these local transitions in detail, while a coarser model suffices for songs that are far away. The Multi-LME exploits this locality and abstracts far-away song songs behind a narrow interface that allows them to be processed largely independently on different compute nodes.

The Multi-LME model itself is a probabilistic embedding model that is trained via maximum likelihood in two steps. First, we formulate the problem of coarsely partitioning the songs as a maximum likelihood problem. Second, models for local sets of songs and their interfaces, called *portals*, to remote songs can be solved as largely independent maximum likelihood problems.

4.1 LME in Multiple Spaces

Suppose, for now, we already have a partition of all songs $\mathcal{S}$ into c clusters $\{C_1, C_2, \dots, C_c\}$. How to get this partition will be explained later. Naively, we could train a separate LME on each cluster in fully parallel fashion to embed its songs in an individual space. This offers a probabilistic model for the intra-cluster transitions. However, we still need to account for the inter-cluster transitions, since there is typically still a substantial portion of inter-cluster transitions.

We model these inter-cluster transitions via what we call *portals*[1]. Portals can be thought of as virtual songs added to each cluster to connect them. Each cluster has $2(c-1)$ portals, half of which are entry portals from the other $c-1$ clusters and the other half are exit portals to the other $c-1$ clusters. We use $o^{\text{entry}}_{u,v}/o^{\text{exit}}_{u,v}$ to denote the entry/exit portal in cluster C_u from/to cluster C_v. We also use $\mathcal{O}_u$ to denote the set of portals in C_u.

With the help of portals, it it now possible to model inter-cluster transition in a two-step fashion: suppose we want to do the transition $(s_a \rightarrow s_b)$, with $s_a \in C_u$ and $s_b \in C_v$ as shown in Fig. 2 (right). The Markov chain first transitions from s_a to the exit portal $o^{\text{exit}}_{u,v}$ to cluster v in cluster u (colored orange). It then transitions with probability 1 to the entry portal $o^{\text{entry}}_{v,u}$ from cluster u in cluster v (colored blue). From there, the chain takes a second step to go to s_b.

[1]The name portal is inspired by the video game Portal by Valve Corporation. A illuminating video that explains the idea can be found on their website `http://store.steampowered.com/video/400`.

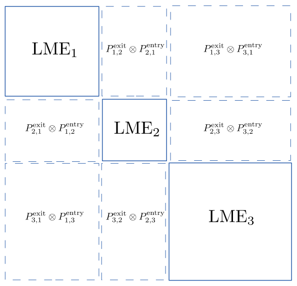
LME_1
$P^{\text{exit}}_{1,2} \otimes P^{\text{entry}}_{2,1}$
$P^{\text{exit}}_{1,3} \otimes P^{\text{entry}}_{3,1}$
$P^{\text{exit}}_{2,1} \otimes P^{\text{entry}}_{1,2}$
LME_2
$P^{\text{exit}}_{2,3} \otimes P^{\text{entry}}_{3,2}$
$P^{\text{exit}}_{3,1} \otimes P^{\text{entry}}_{1,3}$
$P^{\text{exit}}_{3,2} \otimes P^{\text{entry}}_{2,3}$
LME_3

Figure 3: Illustration of the effect of portal trick on the transition probability matrix for the case of three clusters. We assume the songs within the same cluster are grouped together. The intra-cluster transitions (diagonal blocks) are decided by the local LME. The inter-cluster transitions (off-diagonal blocks) are rank-one approximated by the outer product (denoted by $\otimes$) of an exit vector and an entry vector.

This means that $\Pr(p^{[i]}|p^{[i-1]})$ is modeled as the product of $\Pr(o^{\text{exit}}_{u,v}|p^{[i-1]})$ and $\Pr(p^{[i]}|o^{\text{entry}}_{v,u})$, each of which depends on its embedding in its own space. More specifically, we have

$$\Pr(p^{[i]}|p^{[i-1]}) = \frac{e^{-||X(o^{\text{exit}}_{u,v}) - X(p^{[i-1]}))||_2^2}}{\sum_{s \in C_u \cup \mathcal{O}_u} e^{-||X(s) - X(p^{[i-1]}))||_2^2}} \cdot \frac{e^{-||X(p^{[i]}) - X(o^{\text{entry}}_{v,u})||_2^2}}{\sum_{s \in C_v \cup \mathcal{O}_v} e^{-||X(s) - X(o^{\text{entry}}_{v,u})||_2^2}}, \quad \text{if } p^{[i-1]} \in C_u,\ p^{[i]} \in C_v \text{ and } u \neq v. \tag{11}$$

Adding portals does not change the representations of the intra-cluster transition by much. Intra-cluster transition still takes only one-step. As shown in Fig. 2 (left), we go directly from s_a to s_b in cluster u. A slight difference is that, when it comes to the partition function, we also need to consider the contribution of the portals. Formally, we have

$$\Pr(p^{[i]}|p^{[i-1]}) = \frac{e^{-||X(p^{[i]}) - X(p^{[i-1]}))||_2^2}}{\sum_{s \in C_u \cup \mathcal{O}_u} e^{-||X(s) - X(p^{[i-1]}))||_2^2}}, \quad \text{if } p^{[i-1]} \in C_u \text{ and } p^{[i]} \in C_u. \tag{12}$$

Adding popularity terms to equation (11) and (12) for both songs and portals is straightforward.

Figure 3 summarizes and further illustrates the structure of the portal model. Denote with $P^{\text{exit}}_{u,v}$ the length-$|C_u|$ exit vector that contains $\Pr(o^{\text{exit}}_{u,v}|s), \forall s \in C_u$, and with $P^{\text{entry}}_{v,u}$ the length-$|C_v|$ entry vector that contains $\Pr(s|o^{\text{entry}}_{v,u}), \forall s \in C_v$. Then the $|\mathcal{S}| \times |\mathcal{S}|$ transition probability matrix that contains all the $\Pr(s_j|s_i)$ is structured as illustrated in Fig. 3: diagonal blocks that govern the intra-cluster transitions are represented by their local LMEs; any off-diagonal block that stands for inter-cluster transitions can be seen as a rank-1 approximation by the outer product of an exit vector P^{exit} and an entry vector P^{entry}.

Finally, to verify that the Multi-LME is a valid probabilistic model, the sum of transition probabilities to all the songs in the collection must always be upper-bounded by 1. Assuming $p^{[i-1]} \in C_u$,

$$\sum_{s \in \mathcal{S}} \Pr(s|p^{[i-1]}) = \sum_{s \in C_u} \Pr(s|p^{[i-1]}) + \sum_{s \notin C_u} \Pr(s|p^{[i-1]})$$

$$
\begin{aligned}
&= \sum_{s \in C_u} \Pr(s|p^{[i-1]}) + \sum_{C_v, v \neq u} \sum_{s \in C_v} \Pr(s|p^{[i-1]}) \\
&= \sum_{s \in C_u} \Pr(s|p^{[i-1]}) + \sum_{C_v, v \neq u} \sum_{s \in C_v} \Pr(o_{u,v}^{\text{exit}}|p^{[i-1]}) \Pr(s|o_{v,u}^{\text{entry}}) \\
&= \sum_{s \in C_u} \Pr(s|p^{[i-1]}) + \sum_{C_v, v \neq u} \Pr(o_{u,v}^{\text{exit}}|p^{[i-1]}) \sum_{s \in C_v} \Pr(s|o_{v,u}^{\text{entry}}) \\
&\leq \sum_{s \in C_u} \Pr(s|p^{[i-1]}) + \sum_{C_v, v \neq u} \Pr(o_{u,v}^{\text{exit}}|p^{[i-1]}) \sum_{s \in C_v \cup \mathcal{O}_v} \Pr(s|o_{v,u}^{\text{entry}}) \\
&= \sum_{s \in C_u} \Pr(s|p^{[i-1]}) + \sum_{C_v, v \neq u} \Pr(o_{u,v}^{\text{exit}}|p^{[i-1]}) \\
&\leq \sum_{s \in C_u \cup \mathcal{O}_u} \Pr(s|p^{[i-1]}) \\
&= 1. \qquad (13)
\end{aligned}
$$

The fact that it is not equal to 1 is because our model assigns some probability to transitions from real songs to entry portals and transitions that make more than one pass through portals. We allowed them for the convenience of implementation. Note that this approximation is conservative, since any probability or likelihood we compute is actually a lower bound of the true value. We argue that the impact is minimal as long as the number of portals/clusters is small compared to the number of songs in each cluster. Also, when it comes to playlist generation, we can always renormalize the transition probabilities to make them sum to 1.

4.2 Parallelization

The key advantage of the Multi-LME model is that training can be completely (with respect to both objective functions and parameters) decomposed for each cluster. To see it, we can write the likelihood on the training sample as

$$
\begin{aligned}
L(D|X) &= \prod_{i=1}^{|\mathcal{S}|} \prod_{j=1}^{|\mathcal{S}|} \Pr(s_j|s_i)^{T_{ij}} \\
&= \prod_{u=1}^{c} \Bigg[\prod_{i=1}^{|\mathcal{S}|} \prod_{j=1}^{|\mathcal{S}|} \Pr(s_j|s_i)^{T_{ij} \cdot \mathbb{1}^{\{s_i \in C_u \wedge s_j \in C_u\}}} \\
&\quad \cdot \Pr(o_{u,v}^{\text{exit}}|s_i)^{T_{ij} \cdot \mathbb{1}^{\{s_i \in C_u \wedge s_j \in C_v \wedge u \neq v\}}} \\
&\quad \cdot \Pr(s_j|o_{u,v}^{\text{entry}})^{T_{ij} \cdot \mathbb{1}^{\{s_i \in C_v \wedge s_j \in C_u \wedge u \neq v\}}} \Bigg] \\
&\triangleq \prod_{u=1}^{c} L(D_u|X_u). \qquad (14)
\end{aligned}
$$

D_u is the local subset of the training sample D restricted to the songs in cluster C_u, as computed by Alg. 1. Note that each $L(D_u|X_u)$ depends only on the parameters X_u, which is a $|C_u| + 2(c-1)$ by d matrix representing the coordinates of the songs and portals in the space of C_u. To maximize the entire likelihood $L(D|X)$, we can optimize each $L(D_u|X_u)$ independently.

In practice, one can solve all local LMEs in parallel, with each single LME running on one node without communicating with others over the network. If the number of local LMEs exceeds the number of processors, we find the following scheduling algorithm to be effective [9, p.600-606]. For each of the c LMEs, we associate it with the number of the nonzero elements in the transition matrix as a load factor. We then sort these LMEs in descending order of the load factors. Starting from the LME with biggest load, we sequentially assign them to the process with least total load.

Algorithm 1 Build training set for a cluster

```
Input: Training set $D$, partition $\{C_1, C_2, \ldots, C_c\}$
Output: Training set $D_u$ for $C_u$
Initialize $D_u$ as an empty set.
for $(s_i \to s_j) \in D$ do
  if $s_i \in C_u \wedge s_j \in C_u$ then
    Add $(s_i \to s_j)$ to $D_u$
  else if $s_i \in C_u \wedge s_j \in C_v \wedge u \neq v$ then
    Add $(s_i \to o_{u,v}^{\text{exit}})$ to $D_u$
  else if $s_i \in C_v \wedge s_j \in C_u \wedge u \neq v$ then
    Add $(o_{u,v}^{\text{entry}} \to s_j)$ to $D_u$
  else
    Do nothing
  end if
end for
```

If we sequentially solve all local LMEs on a single processor and assuming a fixed number of iterations, the overall complexity is $O(c(\max_i |C_i| + 2(c-1))^2)$ for an appropriate value of c, which is much better than the $O(|\mathcal{S}|^2)$ of the monolithic LME.

4.3 Multi-Space Partitioning

Finally, we need to resolve how to partition the collection of songs $\mathcal{S}$ into c disjoint clusters. To a first approximation, we want to create a partition of songs that gives us as many one-step intra-cluster transitions as possible on one hand. On the other hand, we would like to have the size of clusters to be as balanced as possible, so that the whole process would not be bottlenecked by one big local cluster. Overall, this preclustering step needs to be efficient and scale well, since it will be executed on a single machine.

As shown in [18], the embedding produced by LME forms meaningful clusters, with songs that have high transition frequencies being close to each other. This suggests that LME itself could be used for preclustering.

Consider a small subset of songs $\mathcal{S}_{\text{int}}$ from $\mathcal{S}$, which we call "internal songs". All other songs are called "external songs", denoted by $\mathcal{S}_{\text{ex}} = \mathcal{S} \backslash \mathcal{S}_{\text{int}}$. We propose a modified LME objective that models transitions between internal songs as usual, and aggregates transitions to external songs behind special objects called "medleys" $\mathcal{M} = \{m_1, m_2, \ldots, m_c\}$. We consider c medleys, one for each cluster, which are points in embedding space like portals. Suppose we have a mapping $f(\cdot)$ that maps a external song to a medley. Then our entire training playlist dataset or training transition pairs D can be rewritten into a new one D' by replacing any external songs s with its medley $f(s)$ and only keeping transitions that involve at least one internal songs.

We then train an LME on this new dataset with $|\mathcal{S}_{\text{int}}|$ real songs plus c medleys. Once the embedding is trained, we can reassign each external songs to a medley that further maximizes the training likelihood. For an external song $s \in \mathcal{S}_{\text{ex}}$,

$$
\begin{aligned}
f(s) = \operatorname*{argmax}_{m \in \mathcal{M}} &\prod_{\substack{(s \to s') \in D \\ s' \in \mathcal{S}_{\text{int}}}} \left[\Pr(s'|m)\right]^{T_{\text{idx}(s)\text{idx}(s')}} \\
&\cdot \prod_{\substack{(s' \to s) \in D \\ s' \in \mathcal{S}_{\text{int}}}} \left[\Pr(m|s')\right]^{T_{\text{idx}(s')\text{idx}(s)}}, \qquad (15)
\end{aligned}
$$

Algorithm 2 LME with medleys

```
Input: Training set $D$, internal and external song collection $\mathcal{S}_{\text{int}}$ and $\mathcal{S}_{\text{ex}}$.
Initialize $f(\cdot)$ as random mapping from external songs to medleys.
while have not converged do
    Use Alg. 3 to build training set with medleys $D'$.
    Train LME on $D'$.
    Use equation (15) to update mapping $f(\cdot)$.
end while
```

with transition probabilities given by the LME. This can be done by simply traversing all the medleys.

Training the LME and reassigning external songs to medleys can be alternated to greedily maximize likelihood. This algorithms is summarized in Algs. 2 and 3. Once the algorithm converges, we do the following to assign each song into a cluster:

- For an internal song, assign it to the cluster represented by its closest medley in the embedding space.
- For an external song s that has transitions with internal songs, assign it to the cluster represented by its medley $f(s)$.
- For an external song that does not have transitions with internal songs, we iterate through all clusters and add the external song with the closest connection to the current cluster. The iteration ends when all external songs have been assigned.

There are a few tweaks we used in our implementation that help improve the performances. First, we select the songs with the largest number of appearances as internal songs. Second, we stop the two-step algorithm when less than 0.5% of the external songs that have transitions with internal songs change its medley compared to the last iteration. Third, we choose to train an unboosted LME as empirically it gives more balanced clustering results, which is good for parallelization to be discussed in later sections. Fourth, we fix the dimensionality to be 2 to make the preclustering phase as fast as possible. Fifth, each LME run except the first one is seeded with the embedding produced by the previous iteration.

The number of points that need to be embedded in each LME run is $|\mathcal{S}_{\text{int}}| + c$, so the complexity of each LME iteration is $O((|\mathcal{S}_{\text{int}}| + c)^2)$. As we control $|\mathcal{S}_{\text{int}}|$ to be less than 10% of $\mathcal{S}$, this is acceptable. Also, because of seeding, later LME runs take much less time than earlier runs, as most of the vectors are already near its optimal position at initialization. Usually it takes less than 20 LME runs to converge.

There are other algorithms/packages that could be used for preclustering, and we explore the following two in the following experiments:

1. Spectral Clustering [13] can be used to cluster an undirected graph. The main routine of spectral clustering is a eigenvalue decomposition on the graph Laplacian matrix. Our playlist data forms an undirected graph if we deem each song as a vertex, and transitions between songs as undirected edges associated with the frequency as the weight. We need to run the kmeans phase of the algorithm several times to pick the most balanced partitioning.

Algorithm 3 Build transition pairs with medleys

```
Input: Training set $D$, internal and external song collection $\mathcal{S}_{\text{int}}$ and $\mathcal{S}_{\text{ex}}$, external song to medley mapping $f(\cdot)$.
Output: New training set with medleys $D'$.
Initialize $D'$ as an empty set.
for $(s_i \to s_j) \in D$ do
    if $s_i \in \mathcal{S}_{\text{int}} \wedge s_j \in \mathcal{S}_{\text{int}}$ then
        Add $(s_i \to s_j)$ to $D'$
    else if $s_i \in \mathcal{S}_{\text{int}} \wedge s_j \in \mathcal{S}_{\text{ex}}$ then
        Add $(s_i \to f(s_j))$ to $D'$
    else if $s_i \in \mathcal{S}_{\text{ex}} \wedge s_j \in \mathcal{S}_{\text{int}}$ then
        Add $(f(s_i) \to s_j)$ to $D'$
    else
        Do nothing
    end if
end for
```

	yes_small	*yes_big*	*yes_complete*
Appearance Threshold	20	5	0
Num of Songs	3,168	9,775	75,262
Num of Train Trans	134,431	172,510	1,542,372
Num of Test Trans	1,191,279	1,602,079	1,298,181
Uniform baseline	-8.060856	-9.187583	-11.229025
Unigram baseline	-7.647614	-8.635369	-9.013904
Bigram baseline	-7.332255	-8.850634	-8.917027

Table 1: Statistics and baselines of the playlists datasets.

2. METIS [7] is a popular software package that does undirected graph partitioning by using a hierarchical coarsening and refining algorithm as well as some finely tuned heuristics. It is a very fast and highly optimized implementation.

4.4 Implementation

We have implemented two versions of Multi-LME. One is a single-process version, which is written in C and sequentially solves all local LMEs in the second phase. The other one is an MPI version, which is implemented in C with Open MPI [3]. It dispatches the LMEs in the second phase to different processes. MPI allows the program to be run on both shared-memory (a multi-core machine) and distributed-memory (cluster of machines) setting, and it handles communication transparently. For both versions, we use the LME-based algorithm as the default preclustering method. We also offer the option to take partitioning results produced by other programs as an input file. The source code is available at `http://lme.joachims.org`.

4.5 Extension and Generalization

The portal trick for parallelization can also be applied more generally to the family of models defined in Section 3. The key modification is to introduce portals (potentially different portals for different types of items), and rewrite the conditional probability $\Pr(y|x)$ as $\Pr(y|o^{\text{entry}}) \cdot \Pr(o^{\text{exit}}|x)$. Then the embedding problem breaks up into several independent and much smaller problems in separate spaces. For the preclustering phase, one can come up with a problem-specific algorithm, or just use the general purpose clustering methods discussed in the Section 4.3.

5. EXPERIMENTS

The following experiments analyze training efficiency and prediction performances of the Multi-LME on datasets of

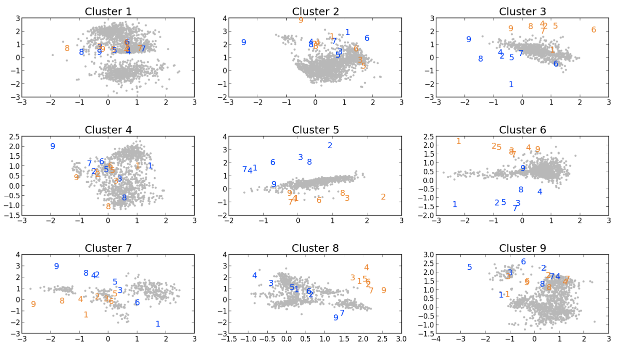

Figure 4: A plot of a multi-spaced embedding produced by Multi-LME with $d = 2$ and $c = 9$ for *yes_big*. Gray points represent songs. Blue and orange numbers represent entry and exit portals respectively, with the number itself denoting the cluster it is linked with.

different sizes. The monolithic LME will serve as the key baseline. We also explore the effect of different preclustering methods, and how robustly the model behaves under different parameter choices.

We evaluate on the playlists datasets collected from *Yes.com* that is described in [18]. Yes.com is a website that provides radio playlists of hundreds of stations in the United States. Its web-based API[2] allows user to retrieve the playlists played in last 7 days at any station in the database. Playlists were collected without taking any preferences on genres. Preprocessing that keeps songs whose number of appearance is above certain threshold offered three datasets with different number of songs, namely *yes_small* (thresholded by 20), *yes_big* (thresholded by 5) and *yes_complete* (thresholded by 0, everything is kept). Then each of the dataset was divided them into a training set and a testing set, with the training set containing as few playlists as possible to have all songs appear at least once. The key statistics about the datasets are given in Table 1, and the datasets are available at `http://lme.joachims.org`.

Unless specified otherwise, our experiments use the following setup. Any model is first trained on the training set and then tested on the testing set. The test performance is evaluated by using the average log-likelihood. It is defined as $\log(\Pr(D_{\text{test}}))/N_{\text{test}}$, where N_{test} is the number of transitions in testing set.

Following [18], our baselines include uniform, unigram and bigram (with Witten-Bell smoothing) models. We altogether list the baselines on three datasets in Table 1. A detailed comparison against these baseline is not of great interest in this paper, since the test log-likelihood of all models is substantially above these baselines. The baselines were largely added to provide a meaningful scale for performance differences.

The experiments were run on a cluster of computers, with each single one having a dual-core Intel Xeon CPU 3.60GHz and 8Gb RAM.

5.1 What does the multi-space embedding with portals look like?

To get an idea of how the songs and portals distribute in the multiple spaces, we first provide the following qualitative

[2]`http://api.yes.com`

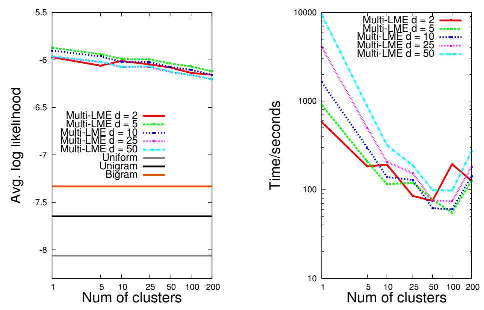

Figure 5: Test log-likelihood (left) and run time (right) for various settings of c and d on *yes_small*.

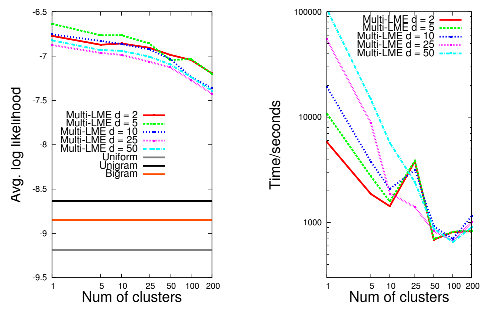

Figure 6: Test log-likelihood (left) and run time (right) for various settings of c and d on *yes_big*.

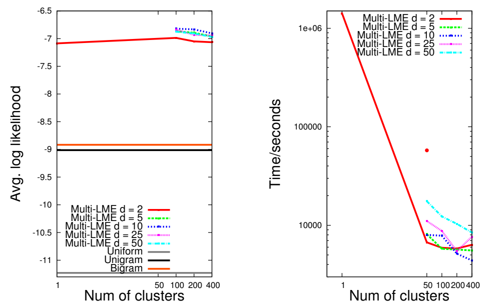

Figure 7: Test log-likelihood (left) and run time (right) for various settings of c and d on *yes_complete*.

analysis. We took the *yes_big* dataset, set dimensionality of embedding $d = 2$ and number of clusters $c = 9$, and then plotted all the embeddings in their own 2D plane. The plots can be seen in Fig. 4.

There are several interesting things worth pointing out. First, different spaces have different scales, as can be seen from the different granularities of x and y axes. This is the result of independent training, and it shows that by introducing portals, we add links between clusters without enforcing any constraints to coordinate them. Second, some clusters (e.g. Cluster 1 and 7) exhibit inner structure with subclusters, which suggest that it is possible to further partition into more clusters without hurting model fidelity. Finally, it can be observed that most of the portals are distributed in the peripheral areas of the mass of songs. This makes sense, as the portals represent songs that are outside the current cluster.

5.2 How does Multi-LME compare to the original LME?

As the first question in our quantitative analysis, we want to explore whether the decoupled training in multiple space

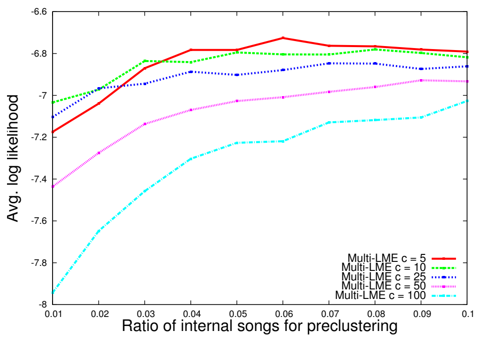

Figure 8: Test log-likelihood against the ratio of internal songs in preclustering phase. Tested on *yes_big*, with $d = 5$.

substantially degrades model fidelity. We trained the Multi-LME with various choice of d and c ($c = 1$ means the original LME with no partitioning), on both *yes_small* and *yes_big*. We used the LME-based preclustering method, with 8% of songs chosen as internal songs. The test log likelihoods are reported in Figs. 5 and 6 left. As can be seen, although the curves tend to go down as we increase the number of clusters, they are still high above the three baselines even for the worst case. Note that for *yes_small* the use of $c = 200$ clusters is excessive, and we even have more portals than real songs in some clusters. The small loss in model fidelity is well acceptable.

How does the runtime scale with the number of clusters? To avoid any outside influence on the runtime, each experiment was run sequentially on a single machine and process. The time reported here are the time spent on preclustering phase plus the average time for the local embeddings accross all clusters. This gives us an idea of how fast training can be done given enough machines so that all individual embeddings can be run at the same time. Figs. 5 and 6 right, show that a substantial speedup until a sweet spot at around 100 clusters is reached. After that runtime increases again, as preclustering and the number of added portals slows down training. The bumps are largely due to some runs taking a larger number of LME iterations than other due to numerical issues with the stopping criterion. As expected, the speedup is bigger the larger the dataset and the larger the dimensionality.

The Multi-LME can also handle the *yes_complete* dataset with 75K songs, which is largely intractable using conventional LME training. Here, we fix the ratio of internal songs for preclustering to be 0.03. Fig. 7 shows the results. Note that we are missing results for $c = 1$ and $d > 2$ — even for $d = 2$, training original LME with a single process already took us more than two weeks. For the test log-likelihood on the left, we can see that Multi-LME is even slightly better than the brute-force training. We conjecture that the added modeling flexibility of not having to fit all points into a single metric space can improve model fit. In terms of runtime, the Multi-LME improves over the LME from more than two weeks to just a few hours. There is a standalone single red point, which stands for the naive parallelization in the distributed memory setting on 50 cores for $d = 2$. The Multi-LME is substantially faster when using the same number of processors.

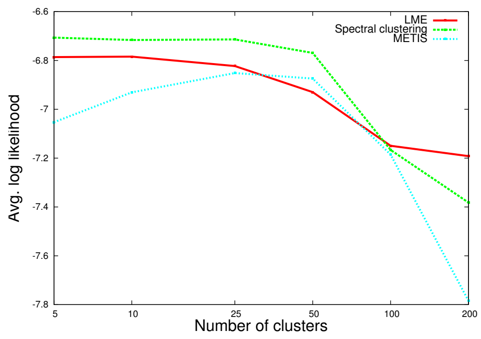

Figure 9: Test log-likelihood on *yes_big*, with $d = 5$ and the preclustering method varied. The ratio of internal songs is set to 0.03 for LME-based method.

5.3 What are the effects of ratio of internal songs in preclustering phase?

We explore how many songs are needed in the preclustering stage. For a range of different numbers of clusters we vary the ratio of internal songs in the preclustering phase. The resulting curves are shown in Fig. 8. As expected, using more internal songs produces Multi-LME embeddings with better test log-likelihood. The models with bigger c tends to need to more internal songs. The curves gradually flatten once the ratio is above certain threshold, which should be considered the best ratio.

In practice, the best ratio needs to be tuned for different datasets. As in our experiments, 0.08 tends to work well for *yes_small* and *yes_big*, but for *yes_complete* 0.03 is enough. This may be due to the fact that a small set of popular songs are responsible for most of the plays in the playlists dataset.

5.4 What are the effects of different preclustering?

As mentioned in Section 4.3, the preclustering phase can be replaced by general graph partitioning methods. Here we investigate how different methods affect the final test log-likelihood. For different methods, we vary the number of clusters to draw the curves in Fig. 9. Spectral clustering tends to perform slightly better than the LME preclustering. METIS is the worst, but is still high above any of the three baselines. The differences between three methods fairly small.

The comparison does not tell much in some sense, since for all of the three methods, there are quite a few parameters that need tuning. The LME preclustering takes the ratio of internal songs; spectral clustering needs to choose the type of Laplacian and number of rounds kmeans needs to run; METIS has its balancing factor and type of algorithms to use. For the experiments in Fig. 9, these parameters were left at their default settings.

It is also difficult to quantitatively compare run time, as the three method are implemented quite differently. LME preclustering is implemented in C without use of any non-standard libraries; Spectral clustering is written in MATLAB, making use of the efficient eigs function to solve the eigenvalue decomposition problem; METIS is written in highly optimized C code. Also, run time varies when different parameters are applied. Qualitatively, we observed that METIS is almost always the fastest. Depending on the size of the dataset, the advantage our method and spectral clustering may shift.

6. CONCLUSIONS

This paper proposes a probabilistic embedding model that exploits locality in the data to reduce training complexity and to permit parallelization. The key idea is to model highly connected regions in detail, but connect remote region only through a narrow interface that largely decouples the training problems. The formulation as a single probabilistic model and its associated maximum likelihood training objective guides not only how each local model should be fit and connected to other local models, but also how the training problem should be split across multiple computer nodes. Empirical results show orders of magnitude reduced runtime with at worst slightly reduced, but sometimes even improved model fidelity.

7. ACKNOWLEDGMENTS

We thank Johannes Gehrke, Wenlei Xie and Guozhang Wang for their valuable suggestions and advice. This work was supported in part by NSF Awards IIS-1217686 and IIS-0905467.

8. REFERENCES

[1] Y. Bengio, H. Schwenk, J.-S. Senécal, F. Morin, and J.-L. Gauvain. Neural probabilistic language models. *Innovations in Machine Learning*, pages 137–186, 2006.

[2] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao. Optimal distributed online prediction using mini-batches. *The Journal of Machine Learning Research (JMLR)*, 13:165–202, 2012.

[3] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[4] A. Globerson, G. Chechik, F. Pereira, et al. Euclidean embedding of co-occurrence data. *Journal of Machine Learning Research (JMLR)*, 2007.

[5] G. Hinton and S. Roweis. Stochastic neighbor embedding. *Advances in Neural Information Processing Systems (NIPS)*, 15:833–840, 2002.

[6] E. H. Huang, R. Socher, C. D. Manning, and A. Y. Ng. Improving word representations via global context and multiple word prototypes. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (ACL): Long Papers-Volume 1*, pages 873–882. Association for Computational Linguistics, 2012.

[7] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[8] M. Khoshneshin and W. N. Street. Collaborative filtering via euclidean embedding. In *Proceedings of the fourth ACM conference on Recommender systems (RecSys)*, pages 87–94. ACM, 2010.

[9] J. Kleinberg and E. Tardos. *Algorithm design*. Pearson Education India, 2006.

[10] Y. Maron, M. Lamar, and E. Bienenstock. Sphere embedding: An application to part-of-speech induction. In *Neural Information Processing Systems Conference (NIPS)*, 2010.

[11] A. Mnih and G. Hinton. Three new graphical models for statistical language modelling. In *Proceedings of the 24th International Conference on Machine learning (ICML)*, pages 641–648. ACM, 2007.

[12] J. Moore, Shuo Chen, T. Joachims, and D. Turnbull. Learning to embed songs and tags for playlist prediction. In *International Conference on Music Information Retrieval (ISMIR)*, pages 349–354, 2012.

[13] A. Ng, M. Jordan, Y. Weiss, et al. On spectral clustering: Analysis and an algorithm. *Advances in Neural Information Processing Systems (NIPS)*, 2:849–856, 2002.

[14] F. Niu, B. Recht, C. Ré, and S. J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *arXiv preprint arXiv:1106.5730*, 2011.

[15] S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme. BPR: Bayesian personalized ranking from implicit feedback. In J. Bilmes and A. Y. Ng, editors, *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 452–461. AUAI Press, 2009.

[16] S. Rendle, C. Freudenthaler, and L. Schmidt-Thieme. Factorizing personalized markov chains for next-basket recommendation. In *Proceedings of the 19th international conference on World Wide Web (WWW)*, pages 811–820. ACM, 2010.

[17] S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, 2000.

[18] Shuo Chen, J. Moore, D. Turnbull, and T. Joachims. Playlist prediction via metric embedding. In *Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining (KDD)*, pages 714–722. ACM, 2012.

[19] J. B. Tenenbaum, V. De Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.

[20] K. Q. Weinberger, B. D. Packer, and L. K. Saul. Nonlinear dimensionality reduction by semidefinite programming and kernel matrix factorization. In *Proceedings of the tenth international workshop on artificial intelligence and statistics*, pages 381–388, 2005.

[21] D. Zhou, S. Zhu, K. Yu, X. Song, B. L. Tseng, H. Zha, and C. L. Giles. Learning multiple graphs for document recommendations. In *Proceedings of the 17th international conference on World Wide Web (WWW)*, pages 141–150. ACM, 2008.

[22] M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized stochastic gradient descent. *Advances in Neural Information Processing Systems (NIPS)*, 23(23):1–9, 2010.