Recommender Systems - Advanced Models

By: Alex, Anna, Paola, Navya

Objectives

01 **Experiment Setup: Data Preparation and Evaluation**

02 **Fundamental Models: Implement NMF model**

03 **Papers: Advanced Models**

Experiment Setup

Data Overview

MovieLens 1M Dataset

- 1 million ratings from 6000 users on 4000 movies
- Released Feb 2003
- Each user has at least 20 ratings
- Data already separated into movies, users, train, test, and validation
- Link

Data Preparation

Change from **explicit** to **implicit** ratings:

- Original ratings from **1-5**
- Change to **0** if <4 and **1** if >=4 (binary)

```
train.rating = (train.rating >= 4).astype(int)
test.rating = (test.rating >= 4).astype(int)
valid.rating = (valid.rating >= 4).astype(int)
```

Make a **utility matrix**

- Will be used to perform **NMF** (Nonnegative Matrix Factorization)

```
train = train.drop(columns = ['timestamp'])
train.pivot_table(index=['user'], columns=['movie'], values=['rating'], fill_value=0)
```

Data Preparation

Creation of utility matrix:

from

	user	movie	rating
0	U0001	I3186	1
1	U0001	I1270	1
2	U0001	I1721	1
3	U0001	I1022	1
4	U0001	I2340	0
...	...	...	...
696766	U6040	I0924	1
696767	U6040	I1172	1
696768	U6040	I0036	1
696769	U6040	I0720	1
696770	U6040	I1719	1

696771 rows × 3 columns

to

	rating																	
movie	**I0001**	**I0002**	**I0003**	**I0004**	**I0005**	**I0006**	**I0007**	**I0008**	**I0009**	**I0010**	**...**	**I3943**	**I3944**	**I3945**	**I3946**	**I3947**	**I3948**	**I394**
user																		
U0001	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
U0002	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
U0003	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
U0004	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
U0005	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
U6036	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
U6037	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
U6038	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
U6039	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0
U6040	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0

6040 rows × 3648 columns

Data Preparation

All NAs (no rating available) replaced with 0.

```
trna = train.pivot_table(index=['user'], columns=['movie'], values=['rating'])
round(sum(trna.isna().sum())/(6040*3648),5)
```

```
0.96838
```

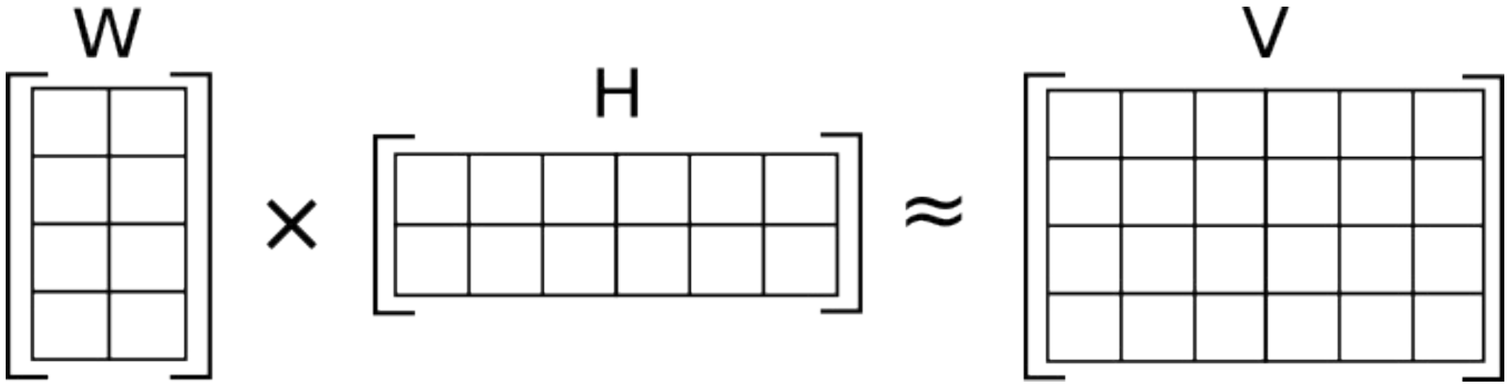
Despite appearances, not all entries are 0! The original data was just very sparse.

```
sum(tr['rating']['I0001'])
```

```
1352
```

NMF

(Non-Negative Matrix Factorization)



User Matrix X Item Matrix = Rating Matrix

```
model = skd.NMF(n_components = 10, init='random', random_state=0, max_iter=2000)
W = model.fit_transform(utilitym)
H = model.components_
WH = np.matmul(W,H)
```

```
WH.shape
```

```
(6040, 3648)
```

```
utilitym.shape
```

```
(6040, 3648)
```

Recommendation

(Non-Negative Matrix Factorization)

```
user_item = pd.DataFrame(V)
```

```
user_item['rec']= V.argmax(axis=1)
user_item
```

rec= index of the highest value for each user, corresponds to most recommended movie

```
for i in range(6040):
    user_item['movie_name'][i] = movies.loc[user_item['rec'][i]][1]
```

Provides movie name based on the recommended movie

	0	1	2	3	4	5	6	7	8	9	...	3640	3641	3642	3643	3644	3645	3646	3647	rec	movie_name
0	0.000000e+00	0.002009	0.006474	0.000109	0.001073	0.001794	0.000382	0.001777	0.000013	0.008379	...	2.643828e-03	0.000010	0.000011	0.009234	0.000349	0.000486	0.004719	0.005810	1751	Two Girls and a Guy (1997)
1	0.000000e+00	0.026695	0.007124	0.008752	0.016247	0.076412	0.003098	0.000000	0.000331	0.000419	...	3.814269e-04	0.000000	0.000034	0.000100	0.002783	0.001209	0.003476	0.021916	1113	Escape from New York (1981)
2	7.022739e-05	0.086977	0.009691	0.006656	0.006745	0.007505	0.004948	0.006912	0.006832	0.181017	...	6.241351e-04	0.001820	0.000060	0.002064	0.000235	0.000655	0.000333	0.000835	571	Little Rascals, The (1994)
3	0.000000e+00	0.015199	0.000000	0.000000	0.000082	0.006984	0.000006	0.001440	0.000027	0.008210	...	7.137590e-08	0.000000	0.000016	0.000002	0.000371	0.000202	0.001752	0.000000	1818	Almost Heroes (1998)
4	1.009084e-03	0.007337	0.008353	0.007860	0.004927	0.012103	0.001245	0.001854	0.000013	0.017574	...	0.000000e+00	0.001200	0.000000	0.011886	0.004063	0.001681	0.009033	0.011712	2362	Patch Adams (1998)

Evaluation

recall@K: measures how many items in the top K positions are relevant

- Hit rate: share of users for which at least one relevant item is present
- Since the goal is to find whether the *single most recent* movie is ranked highly, hit rate is a better metric

NDCG@K: compares all possible rankings of items to the ideal order that contains relevant items at the top positions of the list

Hit Rate

Method:

1. For each user, randomly sample 100 movies they did not watch
2. Find the rankings (estimated with NMF) of those movies
3. Add the user's most recent rating to the list
4. Rank the list
    a. If newest rating is 1, rank from highest to lowest
    b. If the newest rating is 0, rank from lowest to highest
5. Find the top 10 rankings
6. If the new rating is in the top 10 list, the model performed well

A note on ratings and rankings:

Generally, the recommender system only wants to figure out which movies a user will like. However, some users in the dataset had their most recent rating as a 0, and some users never gave a positive rating. To avoid excluding those users, we say the model performed well if a movie they rated poorly was one of our *least* recommended movies, as an inversion.

Hit Rate - Code

```
rates = []
stamps = test.pivot_table(index='user', columns='movie',
                          values='timestamp').rename_axis(index=None, columns=None)
rate = test.pivot_table(index='user', columns='movie',
                        values='rating').rename_axis(index=None, columns=None)

for i in range(0,6040):
    s = stamps.iloc[i].idxmin()
    rates.append(rate[s][i])
```

Gets each user's most recent rating by finding the lowest timestamp

- Find the index of each minimum timestamp
- Find the user's rating of the movie for that timestamp
- Repeat for every user

imports

```
import numpy as np
import pandas as pd
import sklearn as sk
import sklearn.decomposition as skd
import sklearn.metrics as skm
```

Hit Rate - Code

```
hr = 0
for i in range(0,6040):

    li = []
    curr = um.iloc[i]
    for j in range(0,3648):
        if np.isnan(curr[j]):
            li.append(j)

    dcurr = pd.DataFrame(WH).iloc[i]
    dcurr = dcurr[li]
    rank = pd.DataFrame(dcurr.sample(101)).reset_index()
    rank[100:] = rates[i]

    if rates[i] == 1:
        rank = rank.sort_values(by=[i], ascending = False)[:10]
        if 1 in rank['index']:
            hr = hr+1
    else:
        rank = rank.sort_values(by=[i], ascending = True)[:10]
        if 0 in rank['index']:
            hr = hr+1
```

for each user

for each user, finds the list of unrated movies

finds the unrated movies in the prediction matrix, adds the newest rating, and samples

Sorts the ratings, finds the top or bottom 10, and checks if the new rating is present

Result: 13%

NDCG

Method:

1. For each user, randomly sample 100 movies they did not watch
2. Find the rankings (estimated with NMF) of those movies
3. Add the user's most recent rating to the list
4. Rank the list
    a. If newest rating is 1, rank from highest to lowest
    b. If the newest rating is 0, rank from lowest to highest
5. Find the top 10 rankings
6. Find the ideal ranking (newest rating at the very top/bottom of list)
7. Compare using ndcg_score

NDCG - Code

rating = 1

```
if rank.iloc[0][i] == 1:
    true_rel = rank[i]
else:
    true_rel = pd.concat([pd.DataFrame([[0,1]], columns = ['index',i]), rank])
    true_rel = true_rel[:10][i]
ndcg = ndcg + skm.ndcg_score(np.asarray([true_rel]), np.asarray([rank[i]]))
```

rating = 0

```
if rank.iloc[9][i] == 0:
    true_rel = rank[i]
else:
    true_rel = pd.concat([rank, pd.DataFrame([[0,0]], columns = ['index',i])])
    true_rel = true_rel[1:][i]
ndcg = ndcg + skm.ndcg_score(np.asarray([true_rel]), np.asarray([rank[i]]))
```

result: 0.6523

Evaluation - Notes & Future

Notes

- Model performance increased with a higher maximum iteration value and a lower threshold value for NMF
- Replacing all NAs with 0s skewed our predicted values in that direction

Future

- In general, the model is not performing very well yet
- However, this is likely due to a number of factors:
  - Use of relatively simpler NMF model
  - Inexperience in the topic
  - Incorrect choice in evaluators
- For future endeavors, we will explore more modeling and pre-processing methods to improve performance

Fundamental Models

Papers

- "Real-time personalization using embeddings for search ranking at airbnb"
- "Self-Attentive Sequential Recommendation"

Airbnb | Vacation rentals, cab

airbnb.com

airbnb

Stays Experiences Online Experiences

Airbnb your home

Where
sav

Check in
Add dates

Check out
Add dates

Who
Add guests

Search

San Diego, CA

San Francisco, CA

Sacramento, CA

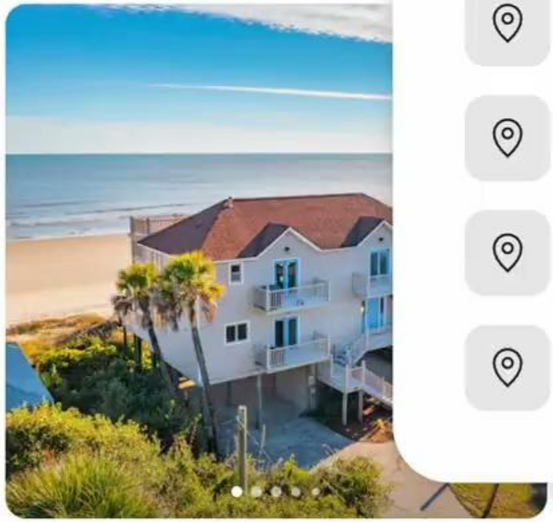
Santa Barbara, CA

San Jose, CA

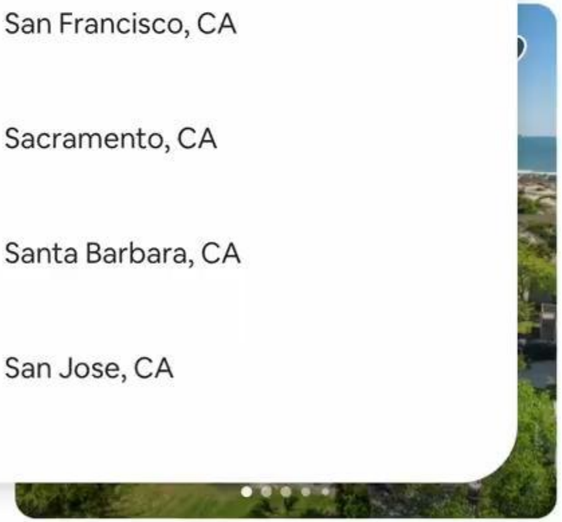
Beachfront Cabins Amaz OMG! Design Treehouses Tiny h

Filters

Display total before taxes



Folly Beach, South Carolina
Folly Beach
Mar 17 – 22
$427 night



Tybee Island, Georgia
Mid Beach
Mar 16 – 21
$422 night

Folly Beach, South Carolina ★ New
Folly Beach
Apr 15 – 20
night

Folly Beach, South Carolina ★ 4.7
Folly Beach
Mar 6 – 11
$232 night

Show map

Paper: Airbnb Search Ranking Algorithm

Challenge: Optimize host and guest preferences

- Less likely to rent same place > 1 time
- Only 1 guest per listing for certain time frame
- Understand guests' short-term and long-term interests

Real-time Personalization using Embeddings for Search Ranking at Airbnb

Mihajlo Grbovic
Airbnb, Inc.
San Francisco, California, USA
mihajlo.grbovic@airbnb.com

Haibin Cheng
Airbnb, Inc.
San Francisco, California, USA
haibin.cheng@airbnb.com

ABSTRACT

Search Ranking and Recommendations are fundamental problems of crucial interest to major Internet companies, including web search engines, content publishing websites and marketplaces. However, despite sharing some common characteristics a one-size-fits-all solution does not exist in this space. Given a large difference in content that needs to be ranked, personalized and recommended, each marketplace has a somewhat unique challenge. Correspondingly, at Airbnb, a short-term rental marketplace, search and recommendation problems are quite unique, being a two-sided marketplace in which one needs to optimize for host and guest preferences, in a world where a user rarely consumes the same item twice and one listing can accept only one guest for a certain set of dates. In this paper we describe Listing and User Embedding techniques we developed and deployed for purposes of Real-time Personalization in Search Ranking and Similar Listing Recommendations, two channels that drive 99% of conversions. The embedding models were specifically tailored for Airbnb marketplace, and are able to capture guest's short-term and long-term interests, delivering effective home listing recommendations. We conducted rigorous offline testing of the embedding models, followed by successful online tests before fully deploying them into production.

1 INTRODUCTION

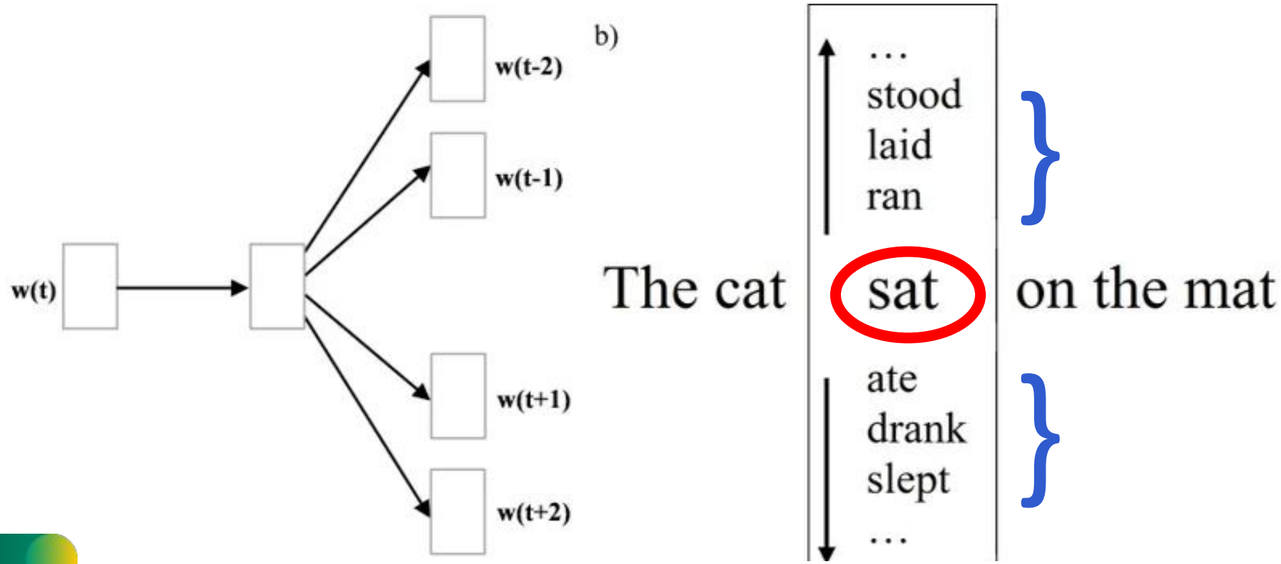
During last decade Search architectures, which were typically based on classic Information Retrieval, have seen an increased presence of Machine Learning in its various components [2], especially in Search Ranking which often has challenging objectives depending on the type of content that is being searched over. The main reason behind this trend is the rise in the amount of search data that can be collected and analyzed. The large amounts of collected data open up possibilities for using Machine Learning to personalize search results for a particular user based on previous searches and recommend similar content to recently consumed one.

The objective of any search algorithm can vary depending on the platform at hand. While some platforms aim at increasing website engagement (e.g. clicks and time spent on news articles that are being searched), others aim at maximizing conversions (e.g. purchases of goods or services that are being searched over), and in the case of two sided marketplaces we often need to optimize the search results for both sides of the marketplace, i.e. sellers and buyers. The two sided marketplaces have emerged as a viable business model in many real world applications. In particular, we have moved from the social network paradigm to a network with two distinct types of participants representing supply and demand. Example industries include accommodation (Airbnb), ride sharing (Uber, Lyft), online

Skip-gram Model

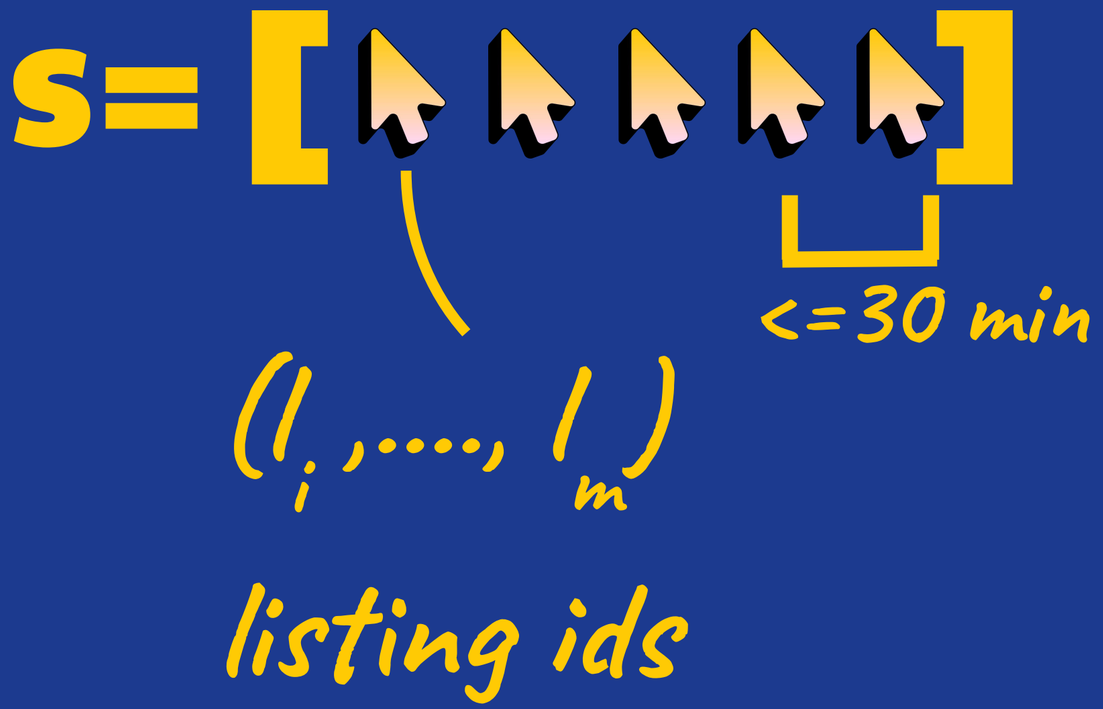
Raw text -> vector format

Input: target word “sat”
Output: context words

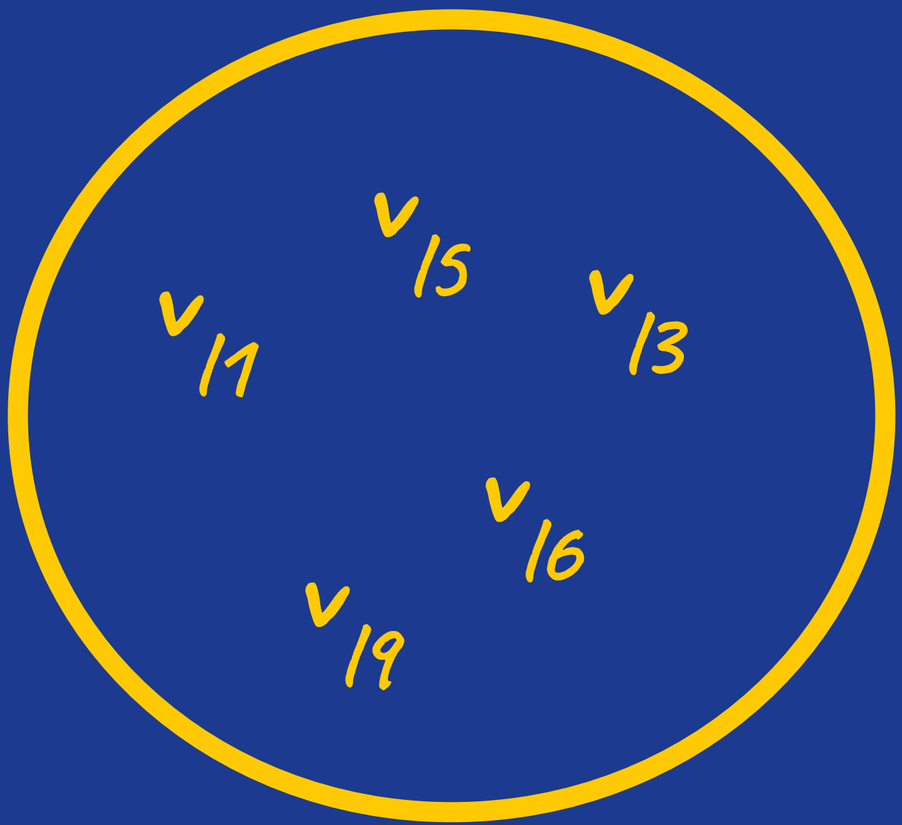


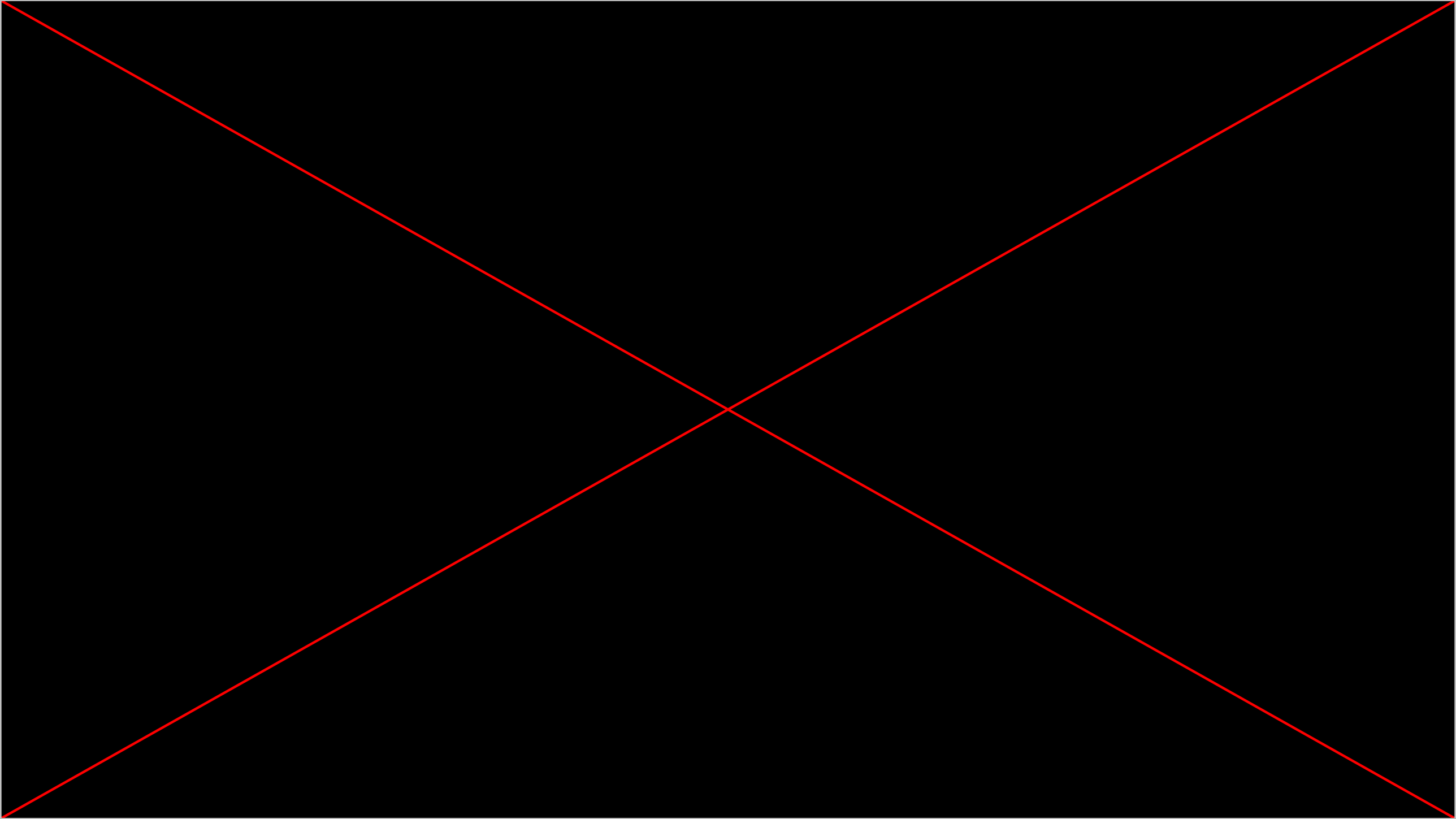
Short Term: Listing Embeddings

Set S of s click sessions:



***d*-dimensional embedding space: $\mathbb{R}^d$**





Short Term: Skip-gram Model

Objective function:

$$\mathcal{L} = \sum_{s \in \mathcal{S}} \sum_{l_i \in s} \Big(\sum_{-m \geq j \leq m, i \neq 0} \log \mathbb{P}(l_{i+j} | l_i) \Big),$$

length of neighborhood for clicked listing

Probability of observing a listing l_{i+j} from neighborhood of clicked listing l_i:

$$\mathbb{P}(l_{i+j} | l_i) = \frac{\exp(\mathbf{v}_{l_i}^\top \mathbf{v}'_{l_{i+j}})}{\sum_{l=1}^{|\mathcal{V}|} \exp(\mathbf{v}_{l_i}^\top \mathbf{v}'_l)},$$

input vector for listing l_i

output vector for listing l

set of unique listing ids

Problem: |V| too large!

v_{11} v_{15} v_{19} v_{113} v_{117} v_{121} v_{125} v_{129} v_{133} v_{137}

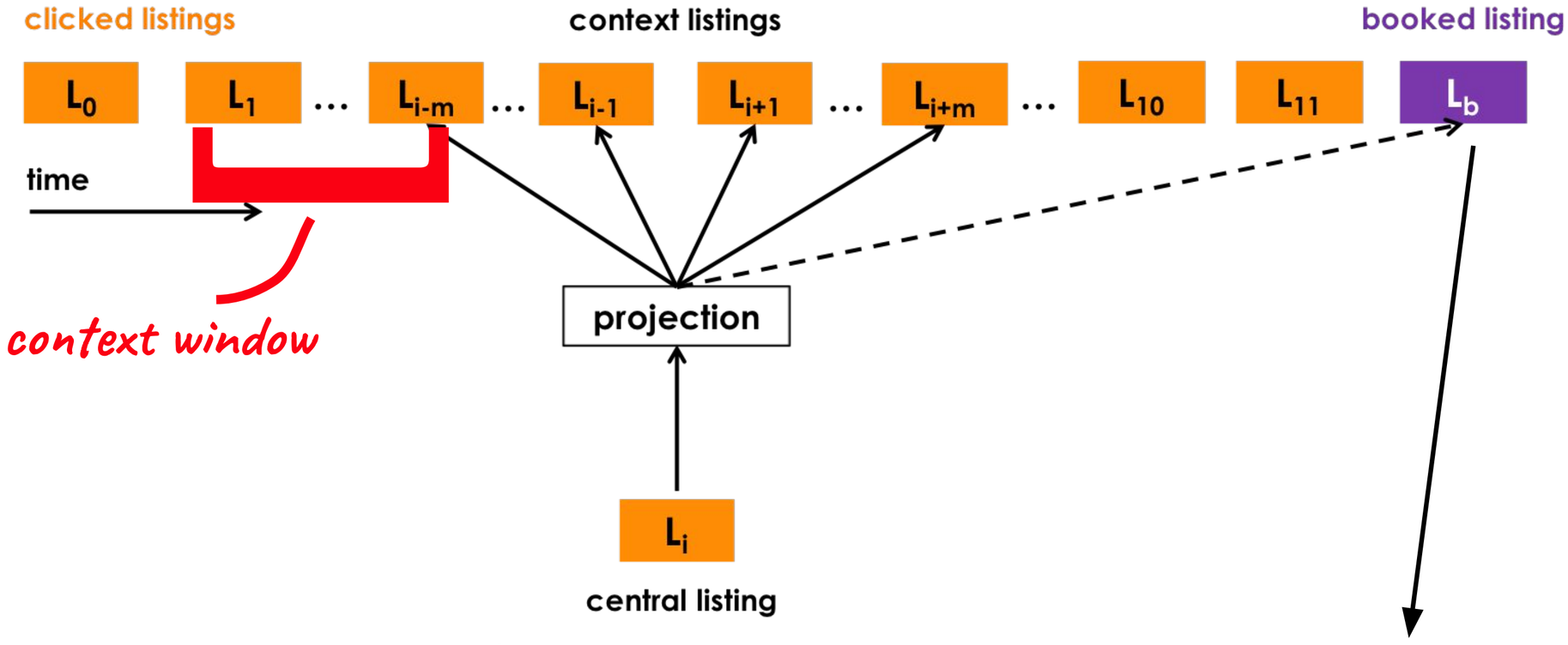
v_{12} v_{16} v_{110} v_{114} v_{118} v_{122} v_{126} v_{130} v_{134} v_{138}

v_{13} v_{17} v_{111} v_{115} v_{119} v_{123} v_{127} v_{131} v_{135} v_{139}

v_{14} v_{18} v_{112} v_{116} v_{120} v_{124} v_{128} v_{132} v_{136} v_{140}

Millions of listing ids....

Skip-gram Model: Negative sampling

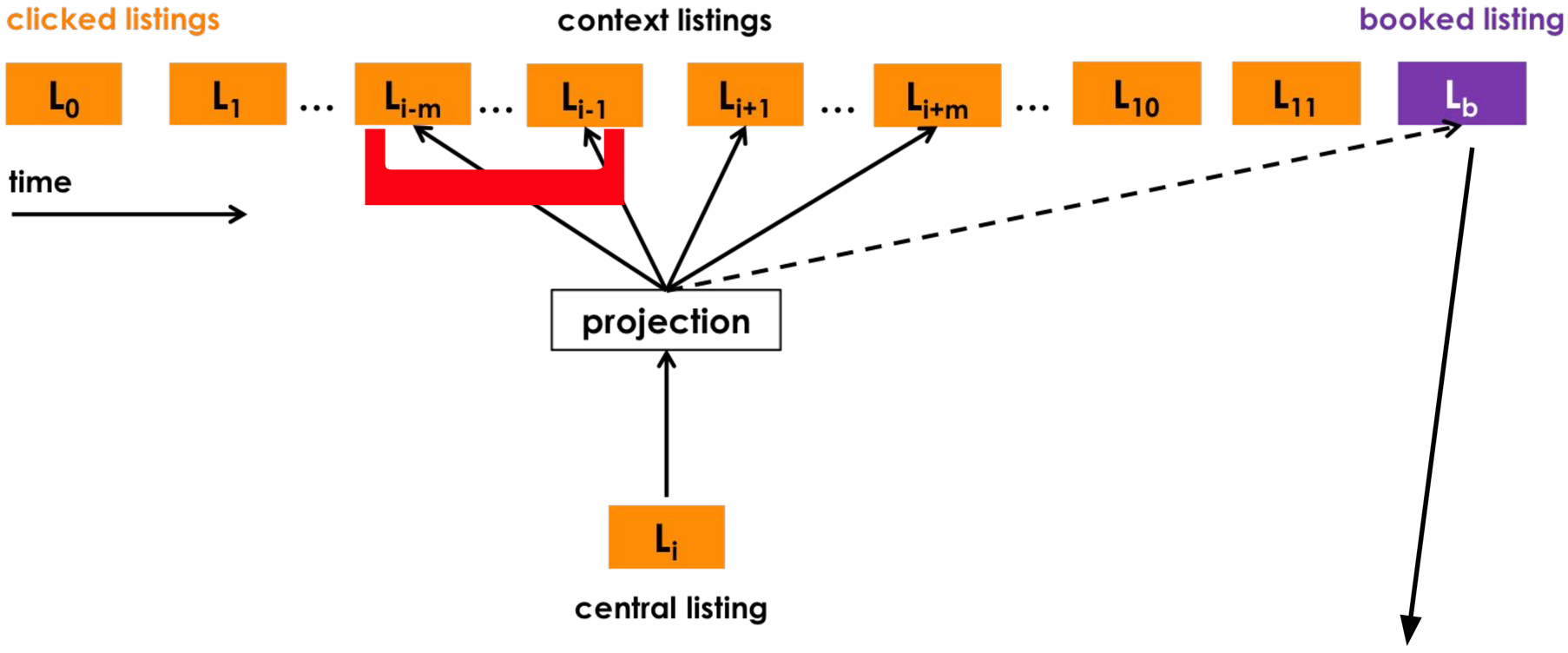


$$\underset{\theta}{\operatorname{argmax}} \sum_{(l,c)\in\mathcal{D}_p} \log \frac{1}{1+e^{-\mathbf{v}'_c\mathbf{v}_l}} + \sum_{(l,c)\in\mathcal{D}_n} \log \frac{1}{1+e^{\mathbf{v}'_c\mathbf{v}_l}} + \log \frac{1}{1+e^{-\mathbf{v}'_{l_b}\mathbf{v}_l}},$$

(clicked listings, their contexts)

(clicked listings, n randomly sampled listings)

Skip-gram Model: Negative sampling

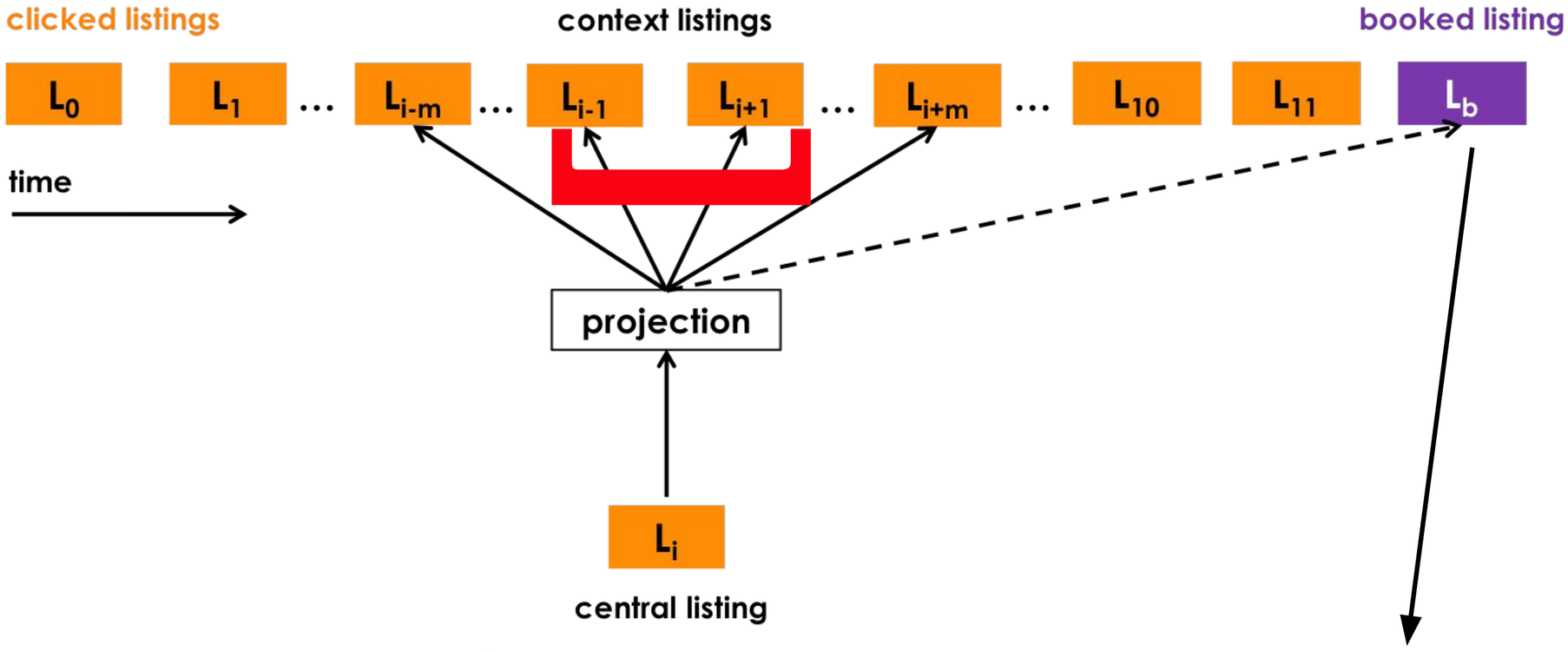


$$\underset{\theta}{\operatorname{argmax}} \sum_{(l,c)\in\mathcal{D}_p} \log \frac{1}{1+e^{-\mathbf{v}'_c\mathbf{v}_l}} + \sum_{(l,c)\in\mathcal{D}_n} \log \frac{1}{1+e^{\mathbf{v}'_c\mathbf{v}_l}} + \log \frac{1}{1+e^{-\mathbf{v}'_{l_b}\mathbf{v}_l}},$$

(clicked listings, their contexts)

(clicked listings, n randomly sampled listings)

Skip-gram Model: Negative sampling



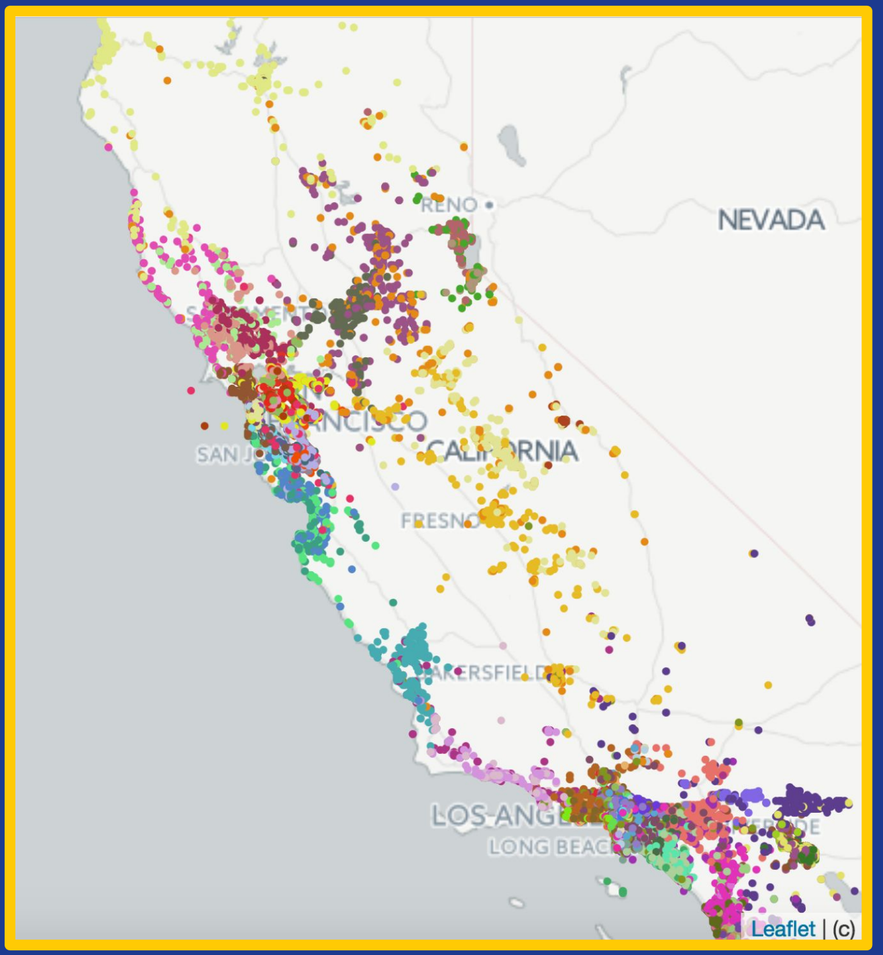
$$\underset{\theta}{\operatorname{argmax}} \sum_{(l,c)\in\mathcal{D}_p} \log \frac{1}{1+e^{-\mathbf{v}'_c\mathbf{v}_l}} + \sum_{(l,c)\in\mathcal{D}_n} \log \frac{1}{1+e^{\mathbf{v}'_c\mathbf{v}_l}} + \log \frac{1}{1+e^{-\mathbf{v}'_{l_b}\mathbf{v}_l}},$$

(clicked listings, their contexts)

(clicked listings, n randomly sampled listings)

Handling Cold Start

New listings do not have embeddings since they're not in click sessions training data.



Use metadata such as location, price, etc.

Geographic similarity (k-means)

Avg cosine similarities

Table 1: Cosine similarities between different Listing Types

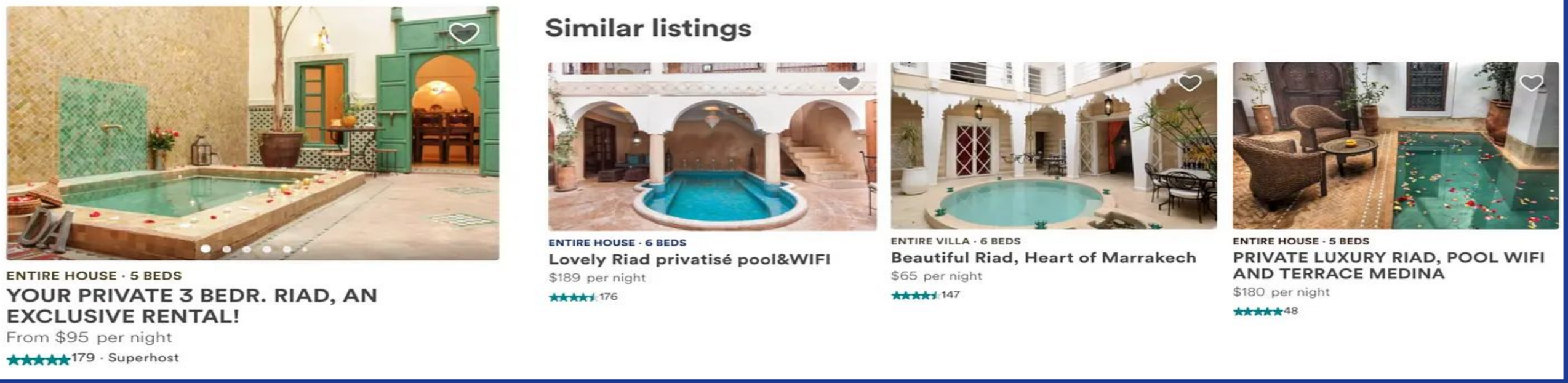
Room Type	Entire Home	Private Room	Shared Room
Entire Home	**0.895**	0.875	0.848
Private Room		**0.901**	0.865
Shared Room			**0.896**

Table 2: Cosine similarities between different Price Ranges

Price Range	<$30	$30-$60	$60-$90	$90-$120	$120+
<$30	**0.916**	0.887	0.882	0.871	0.854
$30-$60		**0.906**	0.889	0.876	0.865
$60-$90			**0.902**	0.883	0.880
$90-$120				**0.898**	0.890
$120+					**0.909**

Handling Cold Start

Some listings characteristics, however, are harder to learn (architecture):



Long Term: User-type & Listing-type Embeddings

We can use signals from a user's longer-term history to further personalize search, BUT:

$$s_b = [\ldots]$$

- ➔ **Bookings are less frequent events, so less data**
- ➔ **At least 5-10 occurrences of entity needed**
- ➔ **Time frame between consecutive bookings may be long. Preferences may change (price point)**

Long Term: User-type & Listing-type Embeddings

SOLUTION: Many-to-one mapping

listing_id

listing_type

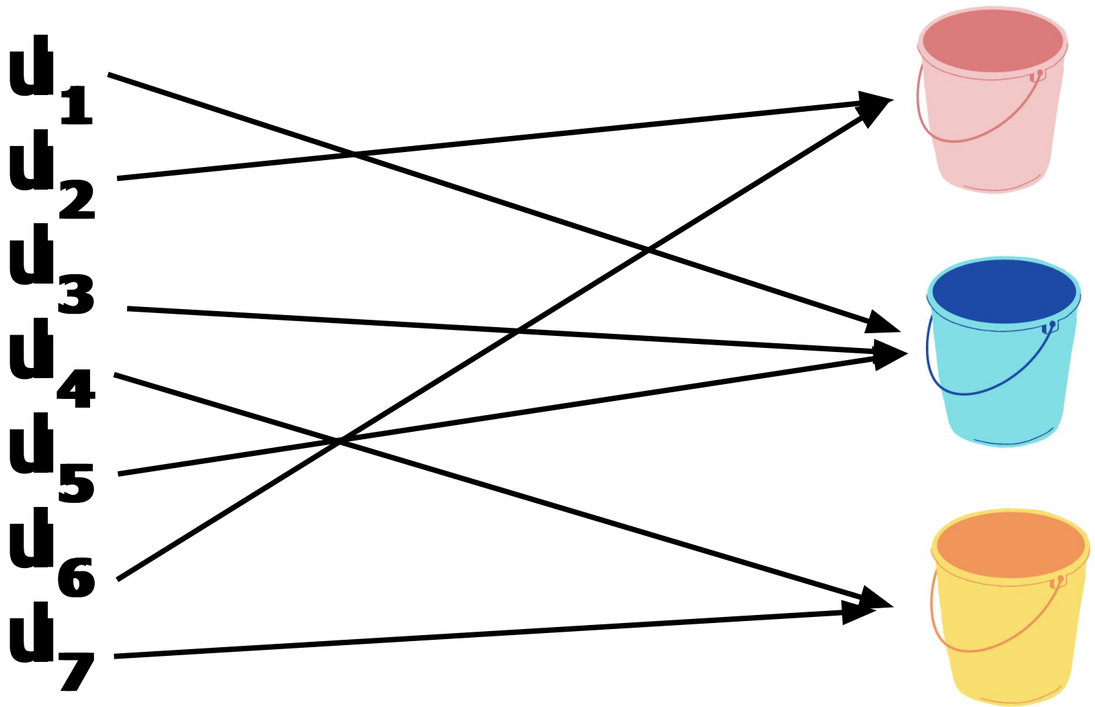


Table 4: Mappings of user meta data to user type buckets

Buckets	1	2	3	4	5	6	7	8
Market	SF	NYC	LA	HK	PHL	AUS	LV	...
Language	en	es	fr	jp	ru	ko	de	...
Device Type	Mac	Msft	Andr	Ipad	Tablet	Iphone	...	
Full Profile	Yes	No						
Profile Photo	Yes	No						
Num Bookings	0	1	2-7	8+				
$ per Night	<40	40-55	56-69	70-83	84-100	101-129	130-189	190+
$ per Guest	<21	21-27	28-34	35-42	43-52	53-75	76+	
Capacity	<2	2-2.6	2.7-3	3.1-4	4.1-6	6.1+		
Num Reviews	<1	1-3.5	3.6-10	> 10				
Listing 5 Star %	0-40	41-60	61-90	90+				
Guest 5 Star %	0-40	41-60	61-90	90+				

Long Term: Skip-gram Model

Table 5: Recommendations based on type embeddings

User Type	
$SF\_lg_1\_dt_1\_fp_1\_pp_1\_nb_3\_ppn_5\_ppg_5\_c_4\_nr_3\_l5s_3\_g5s_3$	
Listing Type	**Sim**
$US\_lt_1\_pn_4\_pg_5\_r_5\_5s_4\_c_2\_b_1\_bd_3\_bt_3\_nu_3$ (large, good reviews)	0.629
$US\_lt_1\_pn_3\_pg_3\_r_5\_5s_2\_c_3\_b_1\_bd_2\_bt_2\_nu_3$ (cheaper, bad reviews)	0.350
$US\_lt_2\_pn_3\_pg_3\_r_5\_5s_4\_c_1\_b_1\_bd_2\_bt_2\_nu_3$ (priv room, good reviews)	0.241
$US\_lt_2\_pn_2\_pg_2\_r_5\_5s_2\_c_1\_b_1\_bd_2\_bt_2\_nu_3$ (cheaper, bad reviews)	0.169
$US\_lt_3\_pn_1\_pg_1\_r_5\_5s_3\_c_1\_b_1\_bd_2\_bt_2\_nu_3$ (shared room, bad reviews)	0.121

$$\underset{\theta}{\text{argmax}} \sum_{(u_t,c)\in\mathcal{D}_{book}} \log \frac{1}{1+\exp^{-\mathbf{v}'_c\mathbf{v}_{u_t}}} + \sum_{(u_t,c)\in\mathcal{D}_{neg}} \log \frac{1}{1+\exp^{\mathbf{v}'_c\mathbf{v}_{u_t}}} + \sum_{(u_t,l_t)\in\mathcal{D}_{reject}} \log \frac{1}{1+\exp^{\mathbf{v}'_{l_t}\mathbf{v}_{u_t}}}.$$

(user_type, listing_type)

Rejection event (explicit host feedback)

Train/Test Process

01 Getting Data

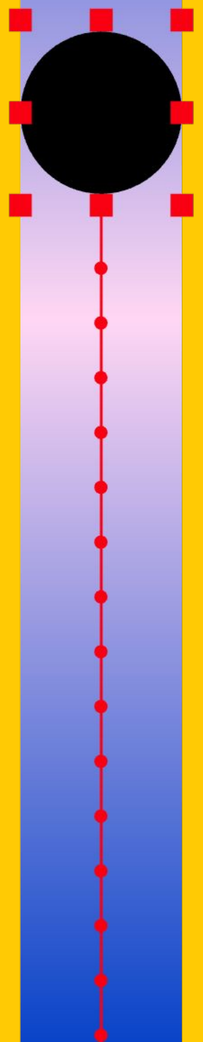
- 800 million click sessions
- Remove accidental, short clicks
- Anonymize sessions

02 Training

- Learned 4.5 million embeddings
- Process new, discard old
- Retrain from scratch

03 Hyperparam

- Dimensionality=32
- Context window size=5
- 10 iterations over training data



Results

Demonstration of Embeddings Evaluation Tool

***Lower values = higher ranking**

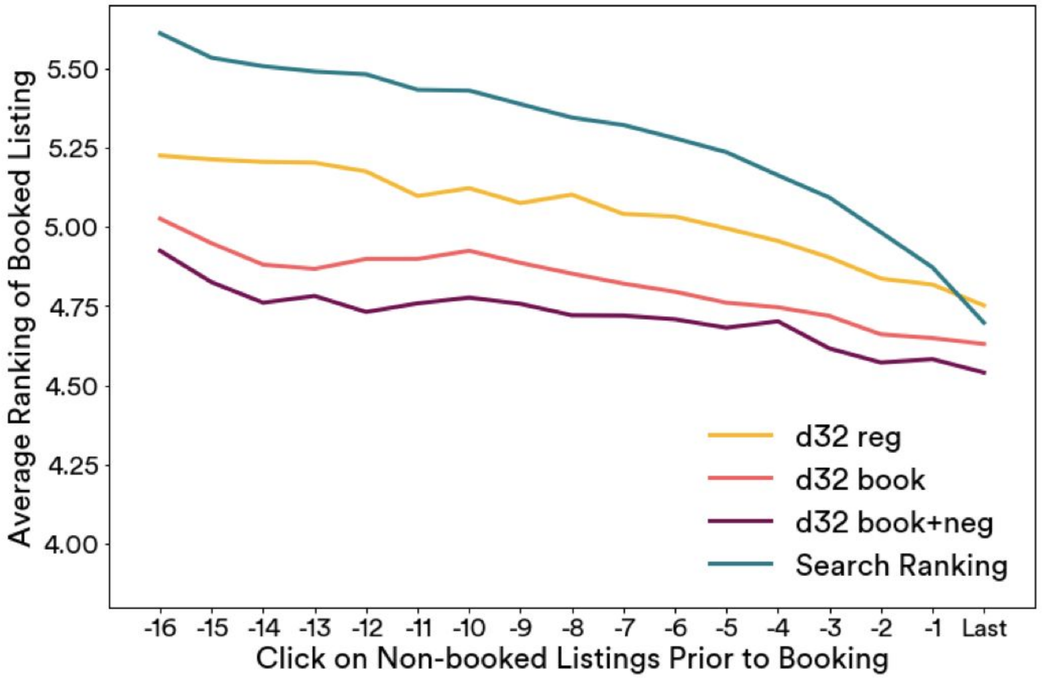


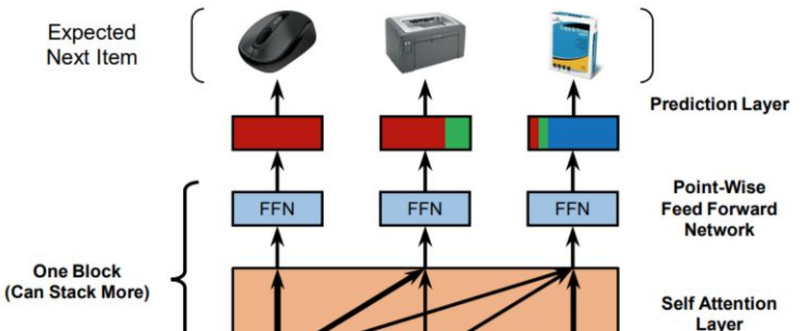
Figure 6: Offline evaluation of Listing Embeddings

Paper: Self-Attentive Sequential Recommendation

Self-Attentive Sequential Recommendation

Wang-Cheng Kang, Julian McAuley
UC San Diego
{wckang,jmcauley}@ucsd.edu

***Abstract*—Sequential dynamics are a key feature of many modern recommender systems, which seek to capture the 'context' of users' activities on the basis of actions they have performed recently. To capture such patterns, two approaches have proliferated: Markov Chains (MCs) and Recurrent Neural Networks (RNNs). Markov Chains assume that a user's next action can be predicted on the basis of just their last (or last few) actions, while RNNs in principle allow for longer-term semantics to be uncovered. Generally speaking, MC-based methods perform**



2018

Paper: Self-Attentive Sequential Recommendation

Challenge: Better analyze patterns in users' sequential and historical activity

- **Understand the "context" of users' activities based on recent actions**
- **Identify relevant actions or items based on historical data**
- **Predict the user's next actions based on the previous action(s)**

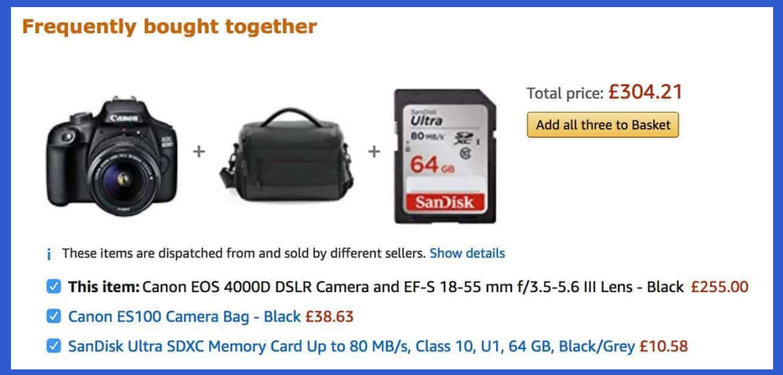
https://doi.org/10.48550/arXiv.1808.09781
https://github.com/kang205/SASRec

Markov Chains (MCs)

- Predicts a user's next action based on their last (or last few) actions
- Performs better in extremely sparse datasets or when you want to minimize the number of model parameters
- Simpler and easier to implement, but has low accuracy
- May fail to capture the intricate dynamics of more complex scenarios
- Ex. Next word prediction or weather predictions

Recurrent Neural Networks (RNNs)

- Summarizes all previous actions via a hidden state, which is used to predict the next action
- Performs better in denser datasets or when it is possible to use more model parameters
- Complex and has longer execution time, but more accurate
- Requires very large and dense datasets
- Exhibit similar behavior to how human brains function
- Ex. Siri or Google Translate



“These are videos YouTube recommends based on what viewers were watching beforehand, related topics, and past watch history.”

Attention

- Focuses on the most relevant parts of the input sequence by assigning weights
- Used in conjunction with other models
  - Ex. attention+RNN model
- Elements in the input are assigned alignment scores and weights and are fed into the decoder algorithm along with a context vector computed with the weights
- Used in image captioning and machine translation

Transformers

- New sequential model
- Uses “self-attention” instead of convolutional or recurrent algorithms
- Very efficient and capable of understanding syntax and semantics
- Ex. Chat Generative Pre-trained Transformer (ChatGPT) is a NLP based Transformer

Self-Attention based Sequential Recommendation model (SASRec)

- Goal is to combine the strengths of MCs and RNNs
  - able to draw context from all actions in the past
  - able to frame predictions in terms of just a small number of actions
- Accomplished by adaptively assigning weights to previous items/actions at each time step
- Due to the "self-attention" mechanism, SASRec tends to focus on more recent actions with sparse datasets and a longer range of activity with denser datasets
- Works well with datasets that have varying density
- Faster and more accurate

https://doi.org/10.48550/arXiv.1808.09781
https://github.com/kang205/SASRec

SASRec - Methodology

python main.py --dataset=Video --train_dir=default

Initial data is a sequence of historical user activity:

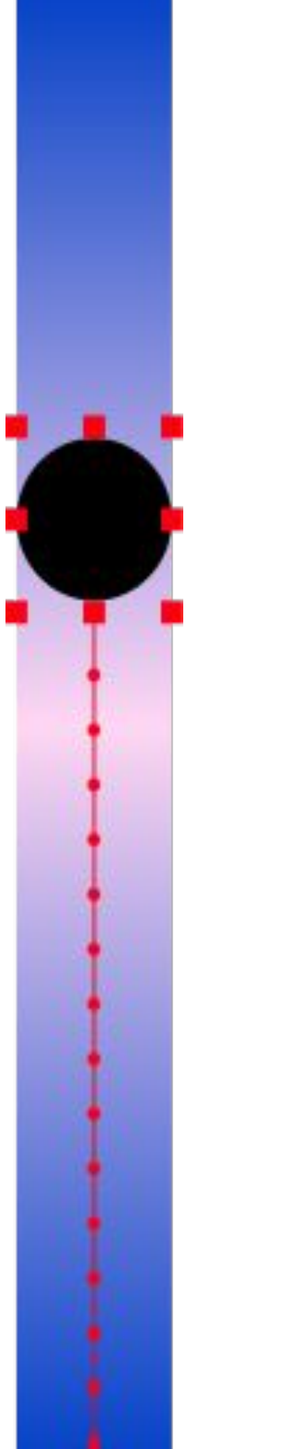
$$(\mathcal{S}_1^u, \mathcal{S}_2^u, \ldots, \mathcal{S}_{|\mathcal{S}^u|-1}^u)$$

Output is the predicted next activity:

$$\mathcal{S}_{|\mathcal{S}^u|}^u$$

It is intuitive to think of the output as a shift of the input sequence, by removing the oldest activity and adding the predicted activity to the end:

$$(\mathcal{S}_2^u, \mathcal{S}_3^u, \ldots, \mathcal{S}_{|\mathcal{S}^u|}^u)$$



SASRec - Methodology

Embedding layer -> Self Attention layer(s) -> Prediction layer

Embedding layer: inject a learnable position embedding into the input embedding to get E

$$\widehat{\mathbf{E}} = \begin{bmatrix} \mathbf{M}_{s_1} + \mathbf{P}_1 \\ \mathbf{M}_{s_2} + \mathbf{P}_2 \\ \cdots \\ \mathbf{M}_{s_n} + \mathbf{P}_n \end{bmatrix}$$

Notation	Description
$\mathcal{U}, \mathcal{I}$	user and item set
$\mathcal{S}^u$	historical interaction sequence for a user u: $(\mathcal{S}_1^u, \mathcal{S}_2^u, ..., \mathcal{S}_{\lvert\mathcal{S}^u\rvert}^u)$
$d \in \mathbb{N}$	latent vector dimensionality
$n \in \mathbb{N}$	maximum sequence length
$b \in \mathbb{N}$	number of self-attention blocks
$\mathbf{M} \in \mathbb{R}^{\lvert\mathcal{I}\rvert \times d}$	item embedding matrix
$\mathbf{P} \in \mathbb{R}^{n \times d}$	positional embedding matrix
$\widehat{\mathbf{E}} \in \mathbb{R}^{n \times d}$	input embedding matrix
$\mathbf{S}^{(b)} \in \mathbb{R}^{n \times d}$	item embeddings after the b-th self-attention layer
$\mathbf{F}^{(b)} \in \mathbb{R}^{n \times d}$	item embeddings after the b-th feed-forward network

SASRec - Methodology

Self-Attention layer(s): calculates a weighted sum of all values for each item

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)\mathbf{V},$$

$$\mathbf{S} = \text{SA}(\widehat{\mathbf{E}}) = \text{Attention}(\widehat{\mathbf{E}}\mathbf{W}^Q, \widehat{\mathbf{E}}\mathbf{W}^K, \widehat{\mathbf{E}}\mathbf{W}^V),$$

A point-wise two-layer feed-forward network is applied to all S:

$$\mathbf{F}_i = \text{FFN}(\mathbf{S}_i) = \text{ReLU}(\mathbf{S}_i\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)},$$

Normalization and Dropout techniques are also used to stabilize the neural network and prevent overfitting

Notation	Description
$\mathcal{U}, \mathcal{I}$	user and item set
$\mathcal{S}^u$	historical interaction sequence for a user u: $(\mathcal{S}_1^u, \mathcal{S}_2^u, ..., \mathcal{S}_{\lvert\mathcal{S}^u\rvert}^u)$
$d \in \mathbb{N}$	latent vector dimensionality
$n \in \mathbb{N}$	maximum sequence length
$b \in \mathbb{N}$	number of self-attention blocks
$\mathbf{M} \in \mathbb{R}^{\lvert\mathcal{I}\rvert \times d}$	item embedding matrix
$\mathbf{P} \in \mathbb{R}^{n \times d}$	positional embedding matrix
$\widehat{\mathbf{E}} \in \mathbb{R}^{n \times d}$	input embedding matrix
$\mathbf{S}^{(b)} \in \mathbb{R}^{n \times d}$	item embeddings after the b-th self-attention layer
$\mathbf{F}^{(b)} \in \mathbb{R}^{n \times d}$	item embeddings after the b-th feed-forward network

SASRec - Methodology

Prediction Layer: predict the next item/relevance of item i based on F

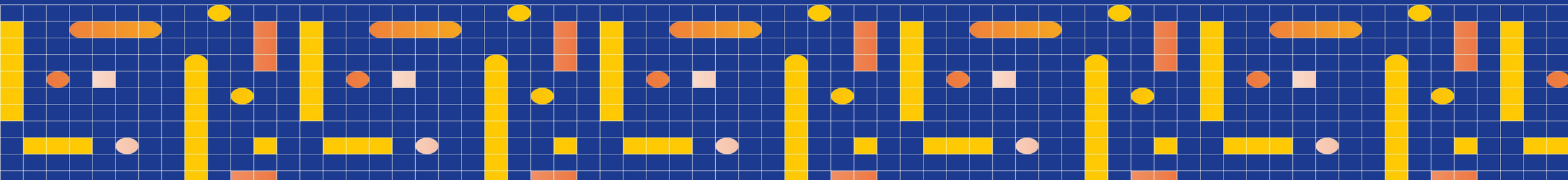
$$r_{i,t} = \mathbf{F}_t^{(b)} \mathbf{N}_i^T,$$

Higher ri,t means a higher relevance, and items with high values are ranked higher to score recommendations

Notation	Description
$\mathcal{U}, \mathcal{I}$	user and item set
$\mathcal{S}^u$	historical interaction sequence for a user u: $(\mathcal{S}_1^u, \mathcal{S}_2^u, ..., \mathcal{S}_{\lvert\mathcal{S}^u\rvert}^u)$
$d \in \mathbb{N}$	latent vector dimensionality
$n \in \mathbb{N}$	maximum sequence length
$b \in \mathbb{N}$	number of self-attention blocks
$\mathbf{M} \in \mathbb{R}^{\lvert\mathcal{I}\rvert \times d}$	item embedding matrix
$\mathbf{P} \in \mathbb{R}^{n \times d}$	positional embedding matrix
$\widehat{\mathbf{E}} \in \mathbb{R}^{n \times d}$	input embedding matrix
$\mathbf{S}^{(b)} \in \mathbb{R}^{n \times d}$	item embeddings after the b-th self-attention layer
$\mathbf{F}^{(b)} \in \mathbb{R}^{n \times d}$	item embeddings after the b-th feed-forward network

SASRec - Evaluation

1. Does SASRec outperform state-of-the-art models including CNN/RNN based methods?
2. What is the influence of various components in the SASRec architecture?
3. What is the training efficiency and scalability (regarding n) of SASRec?
4. Are the attention weights able to learn meaningful patterns related to positions or items’ attributes?



SASRec - Evaluation 1

- vs popularity ranking, Bayesian ranking, MCs, CNNs, and RNNs
- Hit Rate@10: counts the proportion of times that the ground-truth next item is among the top 10 items

$$\text{Hit Rate@10} = \frac{\text{Number of users with at least one relevant item in top 10 recommendations}}{\text{Total number of users}}$$

- Normalized Discounted Cumulative Gain at a cutoff of 10 (NDCG@10): measures the relevance of recommended items and their positions in the ranked list

$$DCG@10 = \sum_{i=1}^{10} \frac{rel_i}{\log_2(i+1)}$$

SASRec - Evaluation 1

Table III: Recommendation performance. The best performing method in each row is boldfaced, and the second best method in each row is underlined. Improvements over non-neural and neural approaches are shown in the last two columns respectively.

Dataset	Metric	(a) PopRec	(b) BPR	(c) FMC	(d) FPMC	(e) TransRec	(f) GRU4Rec	(g) GRU4Rec$^+$	(h) Caser	(i) SASRec	Improvement vs. (a)-(e)	Improvement vs. (f)-(h)
Beauty	Hit@10	0.4003	0.3775	0.3771	0.4310	0.4607	0.2125	0.3949	0.4264	**0.4854**	5.4%	13.8%
	NDCG@10	0.2277	0.2183	0.2477	0.2891	0.3020	0.1203	0.2556	0.2547	**0.3219**	6.6%	25.9%
Games	Hit@10	0.4724	0.4853	0.6358	0.6802	0.6838	0.2938	0.6599	0.5282	**0.7410**	8.5%	12.3%
	NDCG@10	0.2779	0.2875	0.4456	0.4680	0.4557	0.1837	0.4759	0.3214	**0.5360**	14.5%	12.6%
Steam	Hit@10	0.7172	0.7061	0.7731	0.7710	0.7624	0.4190	0.8018	0.7874	**0.8729**	13.2%	8.9%
	NDCG@10	0.4535	0.4436	0.5193	0.5011	0.4852	0.2691	0.5595	0.5381	**0.6306**	21.4%	12.7%
ML-1M	Hit@10	0.4329	0.5781	0.6986	0.7599	0.6413	0.5581	0.7501	0.7886	**0.8245**	8.5%	4.6%
	NDCG@10	0.2377	0.3287	0.4676	0.5176	0.3969	0.3381	0.5513	0.5538	**0.5905**	14.1%	6.6%

SASRec - Evaluation 2

- **Ablation Analysis: Removing certain components to understand the contribution of the component to the overall system**

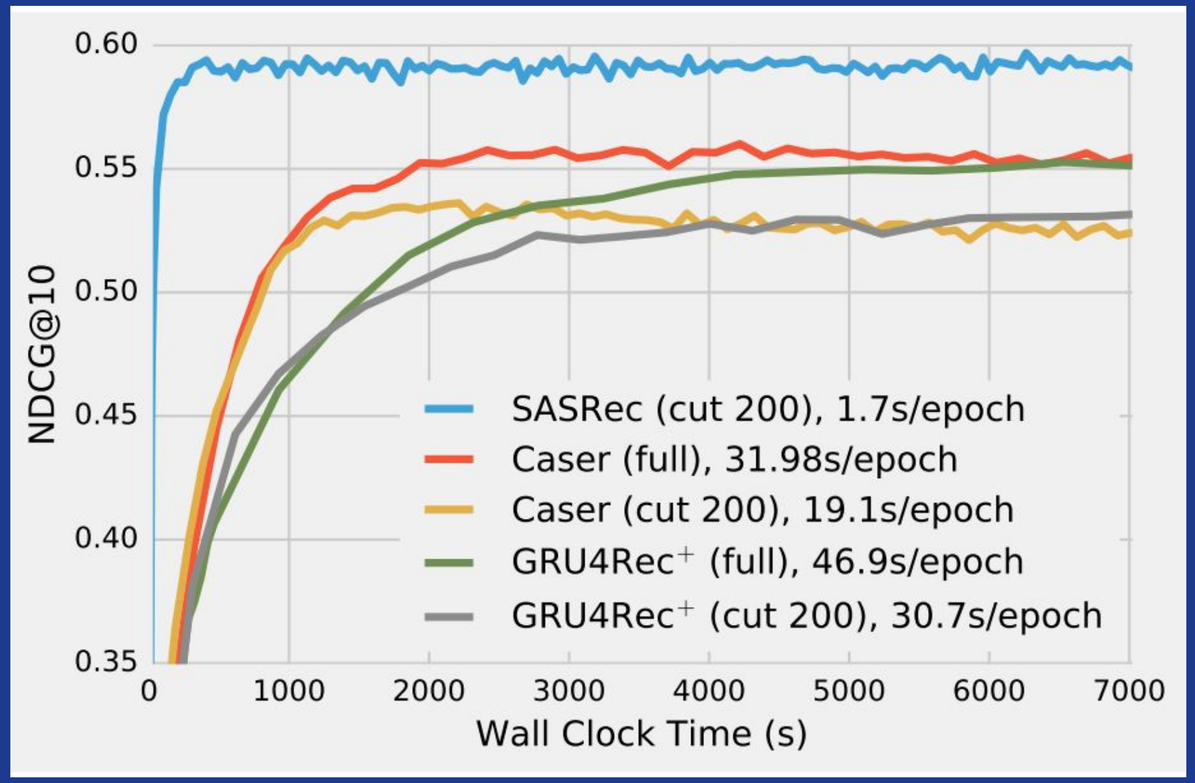
- **Unshared Item Embedding: likely causes overfitting**
- **Remove Residual Connections: difficult to propagate information in sparse datasets**
- **Remove Dropout: not regularizing the model hurts performance**
- **0 Block (b=0): model only depends on the last item**

Table IV: Ablation analysis (NDCG@10) on four datasets. Performance better than the default version is boldfaced. '↓' indicates a severe performance drop (more than 10%).

Architecture	*Beauty*	*Games*	*Steam*	*ML-1M*
(0) Default	0.3142	0.5360	0.6306	0.5905
(1) Remove PE	**0.3183**	0.5301	0.6036	0.5772
(2) Unshared IE	0.2437↓	0.4266↓	0.4472↓	0.4557↓
(3) Remove RC	0.2591↓	0.4303↓	0.5693	0.5535
(4) Remove Dropout	0.2436↓	0.4375↓	0.5959	0.5801
(5) 0 Block (b=0)	0.2620↓	0.4745↓	0.5588↓	0.4830↓
(6) 1 Block (b=1)	0.3066	**0.5408**	0.6202	0.5653
(7) 3 Blocks (b=3)	0.3078	0.5312	0.6275	**0.5931**
(8) Multi-Head	0.3080	0.5311	0.6272	0.5885

SASRec - Evaluation 3

SASRec's training efficiency is much higher in terms of training time



SASRec - Evaluation 4

- SASRec is typically able to identify similar items and assigns larger weights between them without being aware of categories in advance
- Attention mechanism works and is adaptive, position-aware, and hierarchical

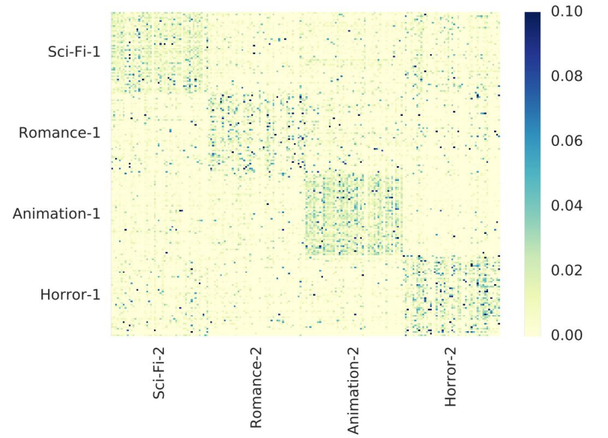
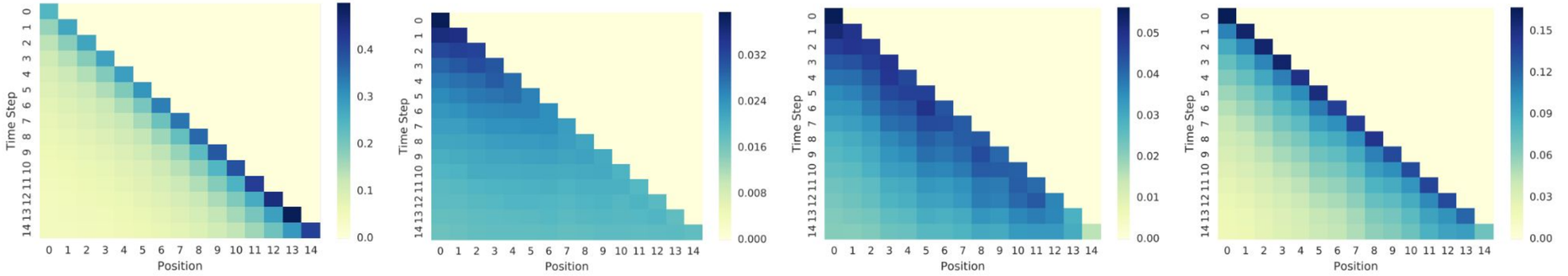


Figure 5: Visualization of average attention between movies from four categories. This shows our model can uncover items' attributes, and assigns larger weights between similar items.



(a) *Beauty*, Layer 1 (b) *ML-1M*, Layer 1, w/o PE (c) *ML-1M*, Layer 1 (d) *ML-1M*, Layer 2

Figure 4: Visualizations of average attention weights on positions at different time steps. For comparison, the heatmap of a first-order Markov chain based model would be a diagonal matrix.

Questions?