Sequential Recommendation

Abstract

In this project, we implemented five recommendation models on the sequential recommendation task. We explored two datasets with different density, **Google Local Reviews** and **MovieLens-1M**.The report contains following parts:(1)Description of our task and study of the two datasets. (2)Introduction to three baseline models. (3)Introduction to state-of-the-art models and the implementation of FPMC and SASRec. (4)Results and conclusion.

1 Task and Dataset

Sequential Recommendation aims to recommend items to users that match the users' interests, basing on the users' historical action sequences. Users and items may have features like gender, age, price, location. However, in this project, we ignore these features and only focus on the sequences. We construct our sequences basing on two datasets **Google Local Reviews** [2] [7] and **MovieLens-1M** [1]. We select these two datasets because the density of them have big difference. Applying our models on them respectively can somehow show the influence of data density. In following subsections, we will give detailed information of the two datasets and how we preprocess these dataset.

1.1 Google Local Reviews(GLR)

This dataset contains reviews about businesses. In this report, we call each review an action. GLR includes user features like name, job, current place and item features like address, gps coordinates. Each review has a timestamp. We focus on the reviews of businesses located in California. we are actually using a 'Google California Reviews' dataset. We extract these reviews according to the address feature of the businesses. After that, we only use the userID, itemID and timestamp of the review. We construct a user to items dictionary which records the items reviewed by each user and the review timestamps. We also construct an item to users dictionary in the same way. Basing on the two dictionaries, we discard users and items with less than 10 actions. Next, for each user in the user-items dictionary, we sort the items reviewed by he/she according to the timestamp. After that we get a sequence for each user, then we can apply our models on these sequences.

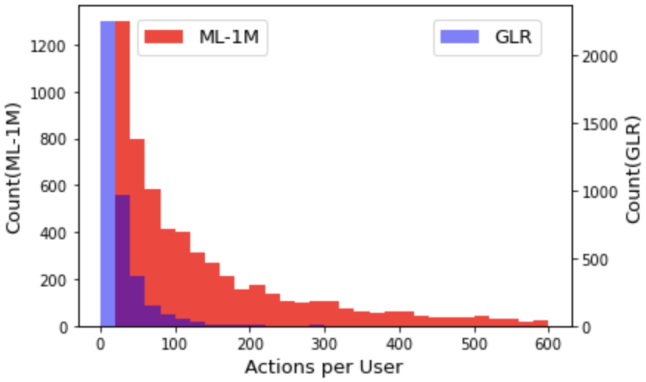

Figure 1. Distribution Histogram of Actions Per User. In this plot, actions per user ranges from 0 to 600.For GLR, 99.97% users' number of actions are in this range. For ML-1M, 95.71% users' number of actions are in this range. Each bin's length is 20. From this histogram we can tell GLR's actions/user are more centralized at small values, which suggests a relative sparsity comparing to ML-1M

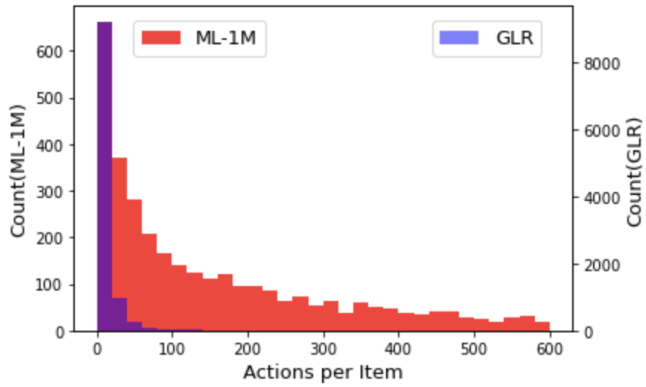

Figure 2. Distribution Histogram of Actions Per Item. In this plot, actions per item ranges from 0 to 600.For GLR, 100% items' number of actions are in this range. For ML-1M, 86.54% items' number of actions are in this range. Each bin's length is 20. GLR's actions/item are more centralized at small values

Table 1 is the statistics of the datasets, we can see that the average actions per user and average actions per item of GLR is much smaller than the other dataset. This 'Google California Reviews' dataset is relatively sparse. In fact, when applying the state-of-the-art models, we

Table 1. Dataset Statistics

	#users	#items	#actions	avg. acts /user	avg. acts /item
GLR	3955	10648	113261	28.6	10.6
ML-1M	6040	3706	1000209	165.6	269.9

perform further preprocessing which makes it even sparser. We also show the actions per user distribution histogram of it in Figure 1. The x-axis is the number of actions per user, the y-axis is the count of how many user's number of actions fall in related bins. We also show the actions per item distribution in Figure 2

1.2 MovieLens-1M(ML-1M)

This dataset contains reviews about movies. It includes user features like age, gender and item features like genres. Each review has a timestamp. We didn't discard users and items for this dataset. Then we preprocess the original data exactly the same way as the Google Local Reviews. From Table 1, we can see that this dataset is dense. Figure 1 and Figure 2 are distributions of actions per user and actions per item.

1.3 Equal Timestamp

There is a problem worth mentioning: when sorting the items basing on timestamps, some timestamps can be equal. If we put items with equal timestamps in the same sequence, the information learned by the models could be unreal. For example, we have to put item B after A or A after B in the sequence, but actually they have the same timestamp. This problem won't affect our baseline models, since they make use of the concrete timestamp values. However, the two state-of-the-art models FPMC and SASRec will suffer from this problem because they only make use of the order of items in the sequences. So in these two models, for items in the same sequence with the same timestamp, we only keep one of them and discard all the others to prevent the sequence from providing unreal information to the models. 13% actions for GLB and 53% actions for ML-1M will be discarded. These ratios show that the equal timestamp is not a small problem especailly for ML-1M dataset.

1.4 Data Split and Performance Evaluation

We split the historical action sequence for each user into three parts as [5]:

(1) The most recent action for testing.
(2) The second most recent action for validation.
(3) All remaining actions for training.

The performance of the models would be evaluated using Hit Rate@10. Hit@10 counts the fraction of times that the ground-truth next item is among the predicted top 10 items. To avoid heavy computation, for each user u, we randomly sample 100 negative items, and rank these items with the ground-truth item.[5][3]

2 Baseline Models

The most straightforward method is recommending the most popular item. However, due to the large scale of the dataset, there are ties that many items have the same popularity score. Thus we designed more detailed methods to score the candidates basing on item-to-item similarity and history interactions.

2.1 Jaccard Similarity(JS)

Denote each item as a set of users who has visited the item. To predict the next item, we simply take the last item in the sequence as the query item and calculate the Jaccard similarity between the query item and the candidates. The most similar candidate will be our prediction.

2.2 Time-weight Jaccard Similarity(TJS)

In the JS model, we only take into consideration the most recent historical interaction, all the other information is discarded, thus the performance of the baseline is relatively weak. In this model, we make use of the whole historical sequence. We calculate the time-weight Jaccard similarity between the candidate and each item in the sequence as following:

$$r(u,i) = \frac{\sum_{j \in S^u \setminus \{i\}} \cdot \mathrm{Sim}(i,j) \cdot f(t_{u,i,j})}{\sum_{j \in S^u \setminus \{i\}} f(t_{u,i,j})} \tag{1}$$

$$f(t_{u,i,j}) = e^{-\lambda * |t_{u,i} - t_{u,j}|} \tag{2}$$

S^u denotes the sequence. We chose λ to be 0.1 for both datasets. Candidate with the largest time-weighted similarity will be our prediction.

2.3 Time-weight Score(TS)

Instead of using Jaccard similarity, we apply a new method to score the candidate. We traverse through the historical sequences, for each item, we record all the items that appeared after it in the same sequence. When predicting the next item basing on the historical sequence, we go through the items in the sequence, the

score gained from a item is the number of times our candidate appeared after the item in same sequences. The total time-weighted score is:

$$rankscore = \frac{\sum_{j \in S^u \setminus \{i\}} score(i,j) \cdot f(t_{u,i,j})}{\sum_{j \in S^u \setminus \{i\}} f(t_{u,i,j})} \quad (3)$$

$$f(t_{u,i,j}) = e^{-\lambda * |t_{u,i} - t_{u,j}|} \quad (4)$$

We chose λ to be 0.6 for GLB and 0.1 for ML-1M. Candidate with the largest time-weighted score will be our prediction.

3 State-of-the-art Methods

State-of-the-art methods for sequential recommendation include: first order Markov chains models such as FPMC[8], TransRec[2]; deep-learning based models trying to mine the sequential patterns such as GRU4Rec[4], SASRec[5]; models trying to make use of the temporal information such as TiSASRec[6].

Our baseline models make use of the temporal information, now we hope to try the models focusing on the sequential patterns. We implemented: (1)FPMC basing on factorized first order Markov chains; (2)SASRec basing on the whole sequence and the self-attention mechanism.

3.1 Factorized Personalized Markov Chains (FPMC)

The first order Markov Chains based models try to construct a transition matrix to represent transition probabilities $P(i|j)$ for each item pair i,j and recommend next item only basing on the last visited item and its transition probabilities. Directly fitting the transition matrix is unrealistic because of the problem of data sparsity. The matrix's shape is $number\ of\ items \times number\ of\ items$, This size is too large, it's unrealistic to have enough training data to get every matrix element well trained. So factorization is necessary, the simplest factorization method is to represent items with embedded vectors and factorize the transition probabilities as the dot products of the vectors.

Factorization can solve the problem of sparsity. However, as [8] indicates, calculating the transition probability only basing on the last visited item without considering the user's interests will cause another problem. For example, a user is interested in science fiction movies and watched a lot of them. After once the user happened to watch a romantic movie, the first order Markov chains recommender is likely to keep recommending romantic movies. For this problem, FPMC represents the users with embedding vectors and calculates the transition $P(item_j|user_k, item_i)$ by:

$$a_{i,j,k} = v_k^{user} \cdot v_j^{next} + v_k^{user} \cdot v_i^{last} + v_j^{next} \cdot v_i^{last} \quad (5)$$

Each item will be encoded to 2 embedded vectors v^{last} and v^{next}. v^{last} is the representation of the item when it is the last visited item, v^{next} is the representation of the item as a candidate to be recommended. This two-vectors design aims to represent the asymmetry of item transitions, which means $P(j|i)$ can be different from $P(i|j)$. v^{user} denotes the embedding vector for the user. v_k^{user} is learned from $user_k$'s interactions with the items in his/her historical sequence. From this point of view, FPMC's Markov chains are not simply first order chains because the user vector contains information of all the items visited by the user.

As in [8], we take Sequential Bayesian Personalized Ranking(S-BPR) as our optimization objective because what we really care about is not the value of the ground-truth item's score but the rank of the score. The objective function is:

$$\arg\max_{\Theta} \sum_{u_k \in U} \sum_{i_t \in S^u} \sum_{l \in I \setminus i_{t+1}} log(\sigma(a_{i_t,i_{t+1},k} - a_{i_t,l,k})) - \lambda ||\hat{\Theta}||^2 \quad (6)$$

Where U denotes the user set, S^u denotes the user's historical sequence, i_t is the t^{th} item in S^u, i_{t+1} is the real next item of i_t, I denotes the item set and l denotes a item in the item set excluding i_{t+1}, λ is the regularization constant, $||\cdot||^2$ denotes the l2 regularization.

3.2 Self-Attentive Sequential Recommendation Model (SASRec)

The self-attention mechanism comes from the model Transformer which achieved great performance in sequential NLP tasks. Unlike RNN based models, this mechanism doesn't suffer from the problem of long-term dependencies. The following subsections will describe how this self-attention based model works.

3.2.1 *Item Embedding and Position Embedding*

SASRec can only take fix-length sequences as its input, so we need to modify the sequences to a fixed length n. If the original length is shorter than n, then we need to repeatedly adding a padding item to fill the sequence; if the original length is longer than n, we only take the last n items into account. Each item will be encoded to a d-dimensional embedding vector.

And since self-attention's calculation doesn't take the position into consideration, so we need to add the position information to the embedding. [5] indicates

using a learnable position embedding can achieve better performance, which is, encoding each position index into a d-dimensional embedding vector and adding this positional embedding to the embedding of the item at this position then keep optimizing these embedding during the training process. The embedding process transfer the original sequence to an embedding matrix $\hat{E}$ with shape $n \times d$. Note that the padding item also has its embedding vector.

3.2.2 Self-Attention Block

First, project the embedding matrix to query matrix Q, key matrix K and value matrix V by:

$$Q = \hat{E}W^Q, K = \hat{E}W^K, V = \hat{E}W^Q \tag{7}$$

W^Q, W^K, W^V are all $d \times d$ matrix. Next, calculate the attention matrix S by:

$$S = Softmax(\frac{Mask(QK^T)}{\sqrt{d}})V \tag{8}$$

When predicting the item at position j, we should only use the items before j in the sequence because the items after j appears in the future. We need to mask the upper triangular part of QK^T. Also, we add the padding item to sequences shorter than the fixed length n, we need to mask the elements of QK^T generated from the padding item.

Then S will be the input of a Feed-Forward Networks(FFN). But before we feed S to the FFN, we apply Dropout, Residual Connections and Layer Normalization(DRL) to S, which is:

$$S = DRL(S) = LayerNorm(S + Dropout(S)) \tag{9}$$

Dropout works as a regularization technique to alleviate the overfitting problem. Residual Connections help the model to efficiently use the information in lower layers and alleviate the vanishing gradient problem. Layer Normalization make the training process converge faster. After the DRL operation, we then apply the FFN with one hidden layer:

$$F = ReLU(SW^{(1)} + b^{(1)})W^{(2)} + b^{(2)} \tag{10}$$

$W^{(1)}$ and $W^{(2)}$ are all $d \times d$ matrices. $b^{(1)}$, $b^{(2)}$ are d-dimensional vectors. Apply the DRL operation again to F as the output of the self-attention block, which is:

$$F = DRL(F) \tag{11}$$

Denote all the operations above in a Self-Attention Block as SAB, we get:

$$F = SAB(\hat{E}) \tag{12}$$

3.2.3 Stacked Self-Attention Blocks and Prediction

As the original Transformer model, we stack multiple Self-Attention Blocks(SAB) to gain better performance. The lower level SAB's output will be the input of the higher level SAB. To be exact, if we have L Blocks:

$$\begin{aligned} F^{(1)} &= SAB^{(1)}(\hat{E}) \\ F^{(2)} &= SAB^{(2)}(F^{(1)}) \\ &\cdots \\ F^{(L)} &= SAB^{(L)}(F^{(L-1)}) \end{aligned} \tag{13}$$

We make predictions basing on $F^{(L)}$. The i^{th} row of the $F^{(L)}$, denoted as $F_i^{(L)}$, is a d-dimensional vector representing our prediction of the $(i+1)^{th}$ item in the sequence. Calculate the dot product between $F_i^{(L)}$ and the item embedding mentioned in the Embedding subsection. The item with the largest dot product will be our prediction for the $(i+1)^{th}$ item. Note that here we can simply use item embedding rather than encode each item to two vectors for input and prediction respectively as in the FPMC model. This is because this model SASRec can achieve the asymmetry of item transitions.

To train the model, we use the same objective function as [5] does. As in FPMC's objective function, denote the user set as U, the user's historical sequence excluding the padding items as S^u, the t^{th} item in S^u as i_t, the $(t+1)^{th}$ item as i_{t+1}. Denote the dot product between item i's embedding and $F_t^{(L)}$ as $r_{i,t}$, the objective function is:

$$\arg\max_{\Theta} \sum_{u \in U} \sum_{i_t \in S^u} [log(\sigma(r_{i_t,t}) + \sum_{j \notin S^u} log(1 - \sigma(r_{j,t}))] \tag{14}$$

3.3 Evaluation

We preprocess the datasets and evaluate the results in the way mentioned in the first section. For items in the same sequence with the same timestamp, to avoid misleading information as we mention in the first subsection, we only keep one of them and discard all the others.

Some sequences are very long, in order to train FPMC and SASRec on our PCs, we set the maximum sequence length to be 18(which suggests a fixed length of 15 for SASRec, see the example in next paragraph) and discard all the previous items in a sequence. After these preprocessing, the number of users, items and interactions are listed in Table 2.

We split the datasets to training, validation and testing sets in the way mentioned in the first section. Here is an example of these models' training and predicting

Table 2. Dataset for FPMC and SASRec

	#users	#items	#actions	avg. acts /user	avg. acts /item
GLR	3823	10023	55182	14.4	5.5
ML-1M	6035	3352	101738	16.9	30.4

Table 3. Hit Rate@10 of the Models on the two datasets

	GLR	ML-1M
JS	0.104	0.514
TJS	0.211	0.544
TS	0.237	0.571
FPMC	0.486	0.572
SASRec	0.564	0.652

process: if a sequence's length is 18, we use the first 16 items for training which means fifteen (user, last item, next item) tuples for FPMC and the input length is 15 for SASRec, the 16^{th} item only works as the 15^{th} item's prediction label in the training process; we use the (user, 16^{th} item) to predict the 17^{th} item for FPMC's validation and use the 2^{nd} to the 16^{th} item to predict the the 17^{th} item for SASRec's validation; we use the $(user, 17^{th}\ item)$ to predict the 18^{th} item for FPMC's testing and use the 3^{rd} to the 17^{th} item to predict the the 18^{th} item for SASRec's testing.

3.4 Implementation details

For both models, we use Adam optimizer and the learning rate=0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-7}$. we set the batch size to be 128 and use 16-dimensional embedding vectors for both models.

For SASRec, the input sequence length is 15, dropout rate is 0.5, number of self-attention blocks is 4. In its loss function(Equation 14), all the negative items'($j \notin S^u$) outputs for each item in every sequence are summed, this will be a heavy burden for our PCs. So instead, we randomly sample 100 negative items for each item in every sequence rather than use all of them.

4 Results and Conclusions

Table 3 shows the Hit Rate@10 of the models we implement on the two datasets. The state-of-the-art models outperform our baseline models, although they only make use of the last 18 items in each sequence. SASRec outperforms FPMC. All the models have better performance on ML-1M dataset than Google Locals. This is because ML-1M has a larger value of average actions per item. This value reflects how many times an item's parameters can get trained. The more actions an item has, the more frequently it will be trained. If an item has few actions, it will be hard for the model to construct its relation between other items.

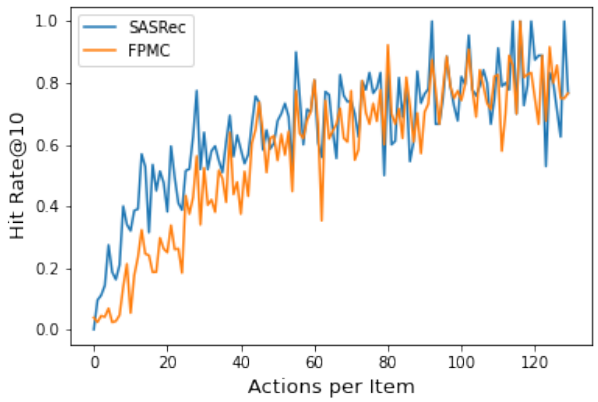

Figure 3. Hit Rate@10 vs Actions per Item with Actions per Item ranging from 0 to 129. For actions per item larger than 129, the general Hit Rate@10 are all 0.85 for FPMC and SASRec

To prove our argument, we explored how actions/item influences the Hit Rate@10 on ML-1M dataset. For each item in the test set, we count how many actions it has in the training set. We got 207 numbers of actions ranging from 0 to 517(Google Local Reviews only has 78 numbers, so we perform this exploration on ML-1M). For each number of actions, we have a set of items whose number of actions in the training set equal this number, the Hit Rate@10 of this set of items can describe the model's performance on this number of actions per item. For actions/item from 0 to 129, there is a non-empty set of items for each of them; for actions/item from 130 to 517, some values have empty sets of items. So we plot the Hit Rate@10 vs Actions per Item with Actions per Item ranging from 0 to 129 in Figure 3. The trend in the figure clearly shows a positive correlation between the model performance and actions per item.

In general, we explored two datasets with different density for sequential recommendation. We implemented 3 baseline models and 2 state-of-the-art models on the two datasets respectively. The difference of the performance on the two datasets indicates the influence of data density. We then did further exploration of how

data density(actions per item) will influence the performance on ML-1M dataset and the results show that models have better performance when actions per item is larger.

References

[1] F Maxwell Harper and Joseph A Konstan. 2015. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)* 5, 4 (2015), 1–19.

[2] Ruining He, Wang-Cheng Kang, and Julian McAuley. 2017. Translation-based recommendation. In *Proceedings of the eleventh ACM conference on recommender systems*. 161–169.

[3] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*. 173–182.

[4] Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, and Domonkos Tikk. 2015. Session-based recommendations with recurrent neural networks. *arXiv preprint arXiv:1511.06939* (2015).

[5] Wang-Cheng Kang and Julian McAuley. 2018. Self-attentive sequential recommendation. In *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE, 197–206.

[6] Jiacheng Li, Yujie Wang, and Julian McAuley. 2020. Time interval aware self-attention for sequential recommendation. In *Proceedings of the 13th international conference on web search and data mining*. 322–330.

[7] Rajiv Pasricha and Julian McAuley. 2018. Translation-based factorization machines for sequential recommendation. In *Proceedings of the 12th ACM Conference on Recommender Systems*. 63–71.

[8] Steffen Rendle, Christoph Freudenthaler, and Lars Schmidt-Thieme. 2010. Factorizing personalized markov chains for next-basket recommendation. In *Proceedings of the 19th international conference on World wide web*. 811–820.