HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION COMMUNICATION TECHNOLOGY
****

WEB MINING PROJECT

Topic: Sequential Recommendation

Lecturer: Prof. NGUYEN KIEM HIEU

GROUP 3

Phạm Đinh Gia Dũng	20194428
Nguyễn Tuấn Dũng	20194427
Phùng Quốc Việt	20194463

Ha Noi, 02/2023

TABLE OF CONTENT

I. Introduction **3**

II. Methods **4**

1. Factorizing Personalized Markov Chains (FPMC) 4
    b. Factorizing Transition Graphs 7
    c. FPMC model 8
2. GRU4REC 8
    a. GRU layer 9
    b. GRU4REC 10
3. SASREC 11
    a.Embedding layer 11
    b. Self-Attention Block 11
    c. Stacking Self-Attention Blocks 12
    d. Prediction layer 12
4. Loss Functions 12
    4.1. Cross Entropy Loss 12
    4.2. BPR Loss 13
    4.3. TOP 1 Loss 13

III. Experiments And Results **13**

1. Dataset 13
2. Data Preprocessing And Augmentation 14
    2.1. Data Preprocessing 14
    2.2. Data Augmentation Methods 14
3. Experiments 16
    3.1. Method 16
    3.2. Metrics 17
    3.3. Results 18

IV. Conclusion **20**

VI. REFERENCES **20**

I. Introduction

Nowadays, with the rise of many online services such as social media, e-commerce sites, streaming services…, recommendation systems have become an integral part of many businesses and services. Recommendation systems are trained to capture the preferences and behavioral patterns of the users using their gathered interaction and purchase data, and make appropriate recommendations and suggestions to those users.
There are many different categories of recommender systems which leverage different types of users' and products' data, like collaborative filtering or content-based filtering. In this project, we focus on sequential recommendation algorithms, which leverage the historical interactions of an user in order to capture the dynamic and sequential behavior and preferences of that user in real time. For example, if a person has recently bought a football and a football shirt, it's likely that they want to buy a football boots afterwards . Sequential recommendation models help capture these valuable patterns and information, while other methods like collaborative filtering or content based filtering fail to achieve this since they don't take the order of these interactions into account.

In this project, we implement, train and evaluate the performance of 3 different sequential recommendation systems for the task of recommending video games, based on the Amazon Video Games dataset. The three algorithms we will be using is:

- Factorizing Personalized Markov Chains (FPMC): Sequential Recommendation method that models both personalization user preference (matrix factorization) and short-term sequential dynamics (markov chains).
- GRU4REC: Sequential Recommendation method that models user's long term sequential behavior using the GRU layer.
- SASREC: Sequential Recommendation method that models user's long term sequential behavior using the self-attention mechanism.

We also experiment with three different types of data augmentation techniques and evaluate their performances in order to see if they can improve upon those existing algorithms, both when we have the full training dataset and when we only have a small fraction of data available for training.
Additionally, we also implement and try out three different types of loss function for the sequential recommendation problem: Cross Entropy Loss, Bayesian Personalized Ranking (BPR) loss and TOP 1 Loss , and evaluate their performances.

II. Methods

1. Factorizing Personalized Markov Chains (FPMC)

FPMC is originally applied for the problem of next-basket recommendation, where the input is a basket of items (which is a set of items), and the task is to predict the next basket in the sequence. Our problem of sequential recommendation is quite similar to this, where we can consider each input item to be a basket with only 1 item in them.

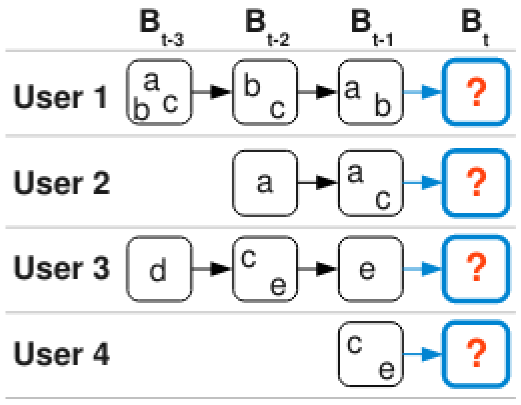


Figure 1: Example of a next basket prediction

a. Personalized Markov Chains

Unpersonalized Markov Chains

Given a set, a Markov Chain of order m can be defined as

$$p(X_t = x_t | X_{t-1} = x_{t-1}, \ldots, X_{t-m} = x_{t-m})$$

with $X_t, \ldots, X_{t-m}$ being random variables.

However, this the size of the state space for this chain is $2^{|I|}$, with I being the set of items. In order to simplify the state space, the authors only considered Markov Chain for data of order 1

$$p(B_t | B_{t-1})$$

Markov Chains are represented by transition matrix A , resulting in the state space to be $2^{|I|} \times 2^{|I|}$. Instead, we represent the transitions over the $|I|$ binary variables which describe a basket:

$$p(i \in B_t | \ell \in B_{t-1}) =: A_{\ell,i}$$

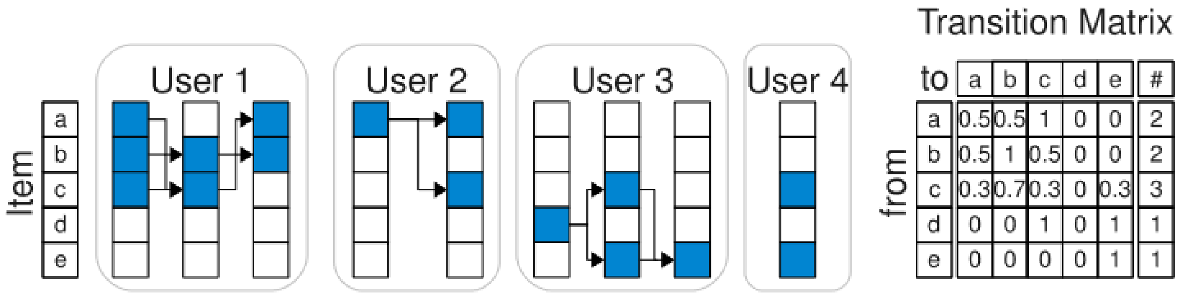
Now, the transition matrix A has a size of $|I|^2$. In order to perform item recommendation, we calculate the probability of an item being purchased given the last basket, which is the average of all probabilities of batches transitions from the start up to the latest basket:

$$p(i \in B_t | B_{t-1}) := \frac{1}{|B_{t-1}|} \sum_{\ell \in B_{t-1}} p(i \in B_t | \ell \in B_{t-1})$$

The maximum likelihood estimator of probability $A_{l,i}$ given the history of baskets B:

$$\hat{A}_{\ell,i} = \hat{p}(i \in B_t | \ell \in B_{t-1}) = \frac{\hat{p}(i \in B_t \wedge \ell \in B_{t-1})}{\hat{p}(\ell \in B_{t-1})} =$$
$$= \frac{|\{(B_t, B_{t-1}) : i \in B_t \wedge \ell \in B_{t-1}\}|}{|\{(B_t, B_{t-1}) : \ell \in B_{t-1}\}|}$$

Here is an example of a transition matrix with the Maximum Likelihood Estimator for the transition probability $p(i \in B_t | l \in B_{t-1})$. The transition matrix is calculated from the example in Figure 1.



	a	b	c	d	e	#
a	0.5	0.5	1	0	0	2
b	0.5	1	0.5	0	0	2
c	0.3	0.7	0.3	0	0.3	3
d	0	0	1	0	1	1
e	0	0	0	0	1	1

Figure 2: An example of a Markov Chain's MLE

For example, for user 4, we can calculate the next item probabilities as follows:

$p(a \in B_t | \{c, e\}) = 0.5\,(0.3 + 0.0) = 0.15$

$p(b \in B_t | \{c, e\}) = 0.5\,(0.7 + 0.0) = 0.35$

$p(c \in B_t | \{c, e\}) = 0.5\,(0.3 + 0.0) = 0.15$

$p(d \in B_t | \{c, e\}) = 0.5\,(0.0 + 0.0) = 0.00$

$p(e \in B_t | \{c, e\}) = 0.5\,(0.3 + 1.0) = 0.65$

From the calculated probabilities, item *b* will be the most recommended item for user 4, since they already purchased item *e*. However, we can see that item *d* might be the

better recommendation, since user 3 who also purchased item *c* and *e* like user 4 purchased item *d.*
This inaccuracy comes from the lack of personalization in the Markov Chain, where each individual users' histories are not accounted for.

Personalized Markov Chains

The above Markov Chain has been unpersonalized for the user. For user-specific Markov Chains, we obtain the following transition probability:

$$A_{u,\ell,i} := p(i \in B_t^u | \ell \in B_{t-1}^u)$$

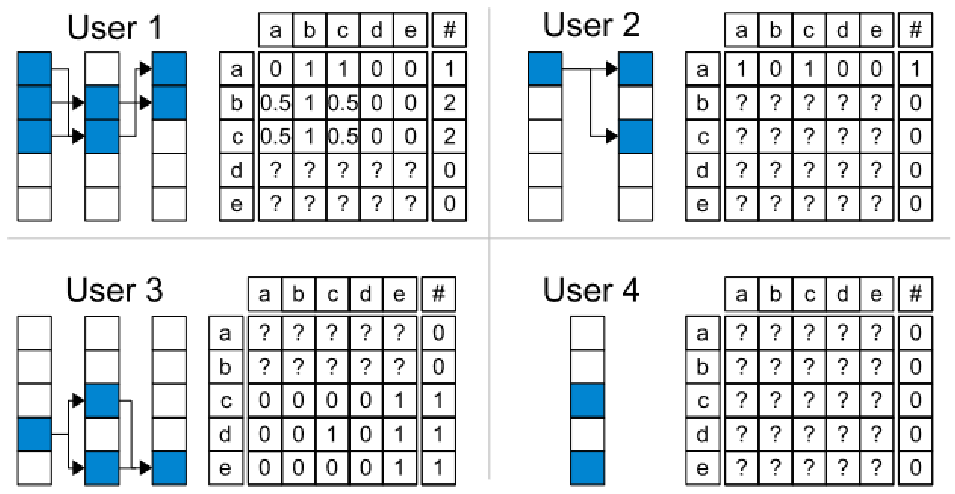
Thus, the personalized prediction probability for the next basket will be:

$$p(i \in B_t^u | B_{t-1}^u) := \frac{1}{|B_{t-1}^u|} \sum_{\ell \in B_{t-1}^u} p(i \in B_t^u | \ell \in B_{t-1}^u)$$

The MLE for $A_{u,l,i}$ given historical baskets set B is:

$$\hat{A}_{u,\ell,i} = \hat{p}(i \in B_t^u | \ell \in B_{t-1}^u) = \frac{|\{(B_t^u, B_{t-1}^u) : i \in B_t^u \wedge \ell \in B_{t-1}^u\}|}{|\{(B_t^u, B_{t-1}^u) : l \in B_{t-1}^u\}|}$$

The figure 3 below is an example of a Personalized Markov Chain, calculated for the example at Figure 1. We can see many fields have to be left empty due to no users' data to estimate the probabilities.



User 1

	a	b	c	d	e	#
a	0	1	1	0	0	1
b	0.5	1	0.5	0	0	2
c	0.5	1	0.5	0	0	2
d	?	?	?	?	?	0
e	?	?	?	?	?	0

User 2

	a	b	c	d	e	#
a	1	0	1	0	0	1
b	?	?	?	?	?	0
c	?	?	?	?	?	0
d	?	?	?	?	?	0
e	?	?	?	?	?	0

User 3

	a	b	c	d	e	#
a	?	?	?	?	?	0
b	?	?	?	?	?	0
c	0	0	0	0	1	1
d	0	0	1	0	1	1
e	0	0	0	0	1	1

User 4

	a	b	c	d	e	#
a	?	?	?	?	?	0
b	?	?	?	?	?	0
c	?	?	?	?	?	0
d	?	?	?	?	?	0
e	?	?	?	?	?	0

Figure 3: An example of personalized MC on sparse data

As we can see, MLE fails against sparse data due to the lack of reliable estimates.

To solve this, the authors factorize the transition cube, which is all transition matrices of individual users stack onto each other.

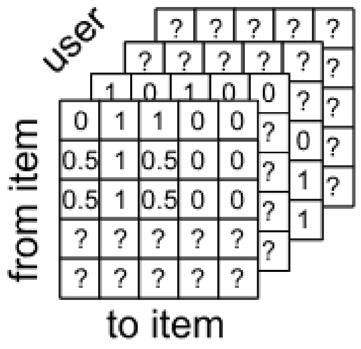


Figure 4: A transition cube

b. Factorizing Transition Graphs

From the transition cube A, the authors modeled the unobserved transition tensor A by a low rank approximation $\widehat{A}$, whereas $\widehat{A}$ is modeled by the pairwise interaction tensor factorization

$$\hat{A}_{u,\ell,i} := \langle v_u^{U,I}, v_i^{I,U} \rangle + \langle v_i^{I,L}, v_\ell^{L,I} \rangle + \langle v_u^{U,L}, v_\ell^{L,U} \rangle$$

which is equivalent to

$$\hat{A}_{u,\ell,i} \quad := \quad \sum_{f=1}^{k_{U,I}} v_{u,f}^{U,I}\, v_{i,f}^{I,U} \quad + \quad \sum_{f=1}^{k_{I,L}} v_{i,f}^{I,L}\, v_{\ell,f}^{L,I} \quad + \quad \sum_{f=1}^{k_{U,L}} v_{u,f}^{U,L}\, v_{\ell,f}^{L,U}$$

This factorization model captures the pairwise interactions between user U and item I, user U and item L and item I and item L.

c. FPMC model

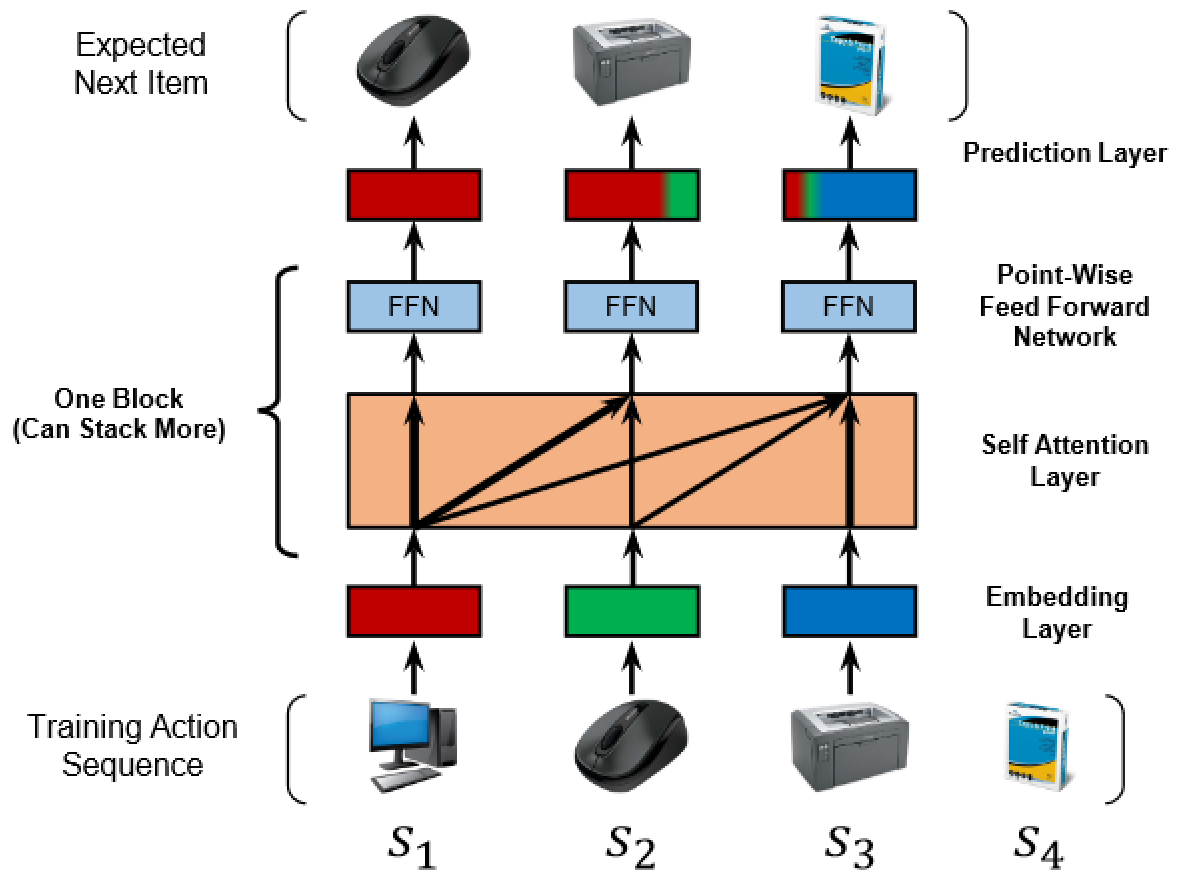
In summary, the FPMC model is:

$$\hat{p}(i \in B_t^u | B_{t-1}^u) = \langle v_u^{U,I}, v_i^{I,U} \rangle + \frac{1}{|B_{t-1}^u|} \sum_{\ell \in B_{t-1}^u} \left(\langle v_i^{I,L}, v_\ell^{L,I} \rangle + \langle v_u^{U,L}, v_\ell^{L,U} \rangle \right)$$

Whereas the transition cube is modeled by a low-rank approximation $\widehat{A}$.

2. GRU4REC

This models' main goal is quite straightforward, which is using (Gated Recurrent Unit) GRU layers to leverage the sequential pattern of an users' interaction. The diagram below summarizes the architecture of our model



a. GRU layer

GRU is a RNN layer, which is developed to model variable length sequences and capture their dynamic behaviors using recurrent connections. An internal state h_t is passed recurrently through the network and updated after processing each time step in the input sequence.

$$h_t = g(Wx_t + Uh_{t-1})$$

h_t : *internal state at time step t*

x_t : *Input item from the sequence at time step t*

W, U : *Weight matrices*

GRU is a variation of the RNN layer with an aim to minimize the problem of gradient vanishing. The current internal state of GRU depends on gates at a GRU cell: the reset gate which controls how much of the previous state we might still want to remember, and an update gate which enables us to control how much of the new state is just a copy of the old state.

$$h_t = (1 - z_t) h_{t-1} + \widehat{h_t}$$

with the update gate:

$$z_t = \sigma(W_z x_t + U_z h_{t-1})$$

and the candidate activation function, which is controlled by the reset gate:

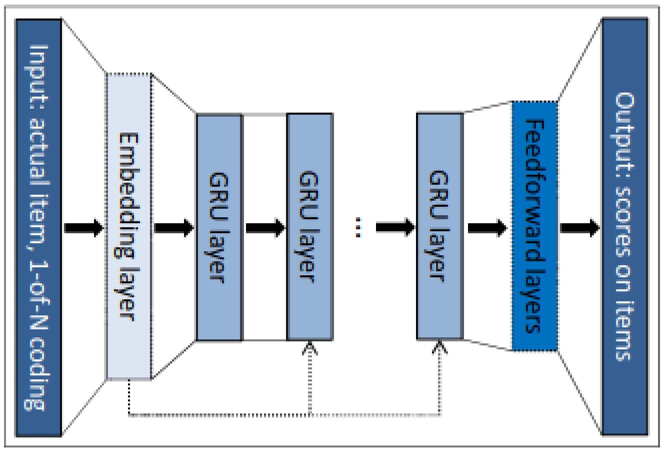
$$\widehat{h_t} = tanh(W x_t + U(r_t \odot h_{t-1}))$$

and the reset gate:

$$r_t = \sigma(W_r x_t + U_r h_{t-1}) h_{t-1}$$

b. GRU4REC

GRU4REC leverages the GRU layer for sequential recommendation. The model's architecture is summarized in the figure below:



The model takes in the current state of the sequence as the input, which can either be the latest item in the sequence or a set of items. In the former case, the item is

represented using one hot encoding, while in the latter case the input to the model will be the weighted sum of one hot encodings of the items in the set.
After representing the input as encoding, the resultant vector is passed through a series of GRU layers with some feed forward layers in between.
The model's final layer outputs the probability score of the items to be the next item in the series.

3. SASREC

Like mentioned above, SASREC is a model similar to the Transformer architecture in NLP and Computer Vision, which aims to utilize self-attention to capture the dynamics of the items in the sequence.

a.Embedding layer

The fixed length item sequence of size n $(s_1, s_2, ... s_n)$ is treated as the input to SASREC. This input vector will be fed through two embedding layers:

- The item embedding layer M which is used to transform the data into item embeddings $E \in R^{n \times d}$, with d being the embedding dimension.
- The positional embedding layer, which helps encode the positional information of the items in the input sequence. This is a learnable embedding $P \in R^{n \times d}$.

The item embedding E and positional embedding P are then added together to create unified embedding $\widehat{E}$.

b. Self-Attention Block

The resulting embedding is used as input for the self attention layer, which learns the interactions between items within the sequence.

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

$$where\ W^Q,\ W^K,\ W^K \text{ are linear projection layers}$$

The core attention mechanism used in this model is Scaled Dot-product Attention. For each token, we have 3 projections: query, key and value. All queries are packed into a matrix Q, and similarly, all keys and values are packed into 2 matrices K and V respectively. The attention matrix is computed as follows:

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

Here, the weight between query i and value j represents the relationship between query i and key j.
The attention vector is passed through a feed forward layer to produce F_i, which is the output of the self-attention layer number i.

$$F_i = FFN(S_i)$$

c. Stacking Self-Attention Blocks

Multiple self-attention blocks can be stacked on top of each other in order to capture more diverse and complex relationships.
However, the problem of vanishing gradient and overfitting due to having too many parameters can arise in deep models. To combat these problems we apply layer normalization before passing through later self-attention layers to stabilize the training process, afterwards dropout regularization is applied to prevent overfitting, and followed by residual connections to propagate information on the lower layers to the higher ones.

$$g(x) = x + Dropout\,(\,g(LayerNorm(x)))$$

$g(x)$*: self attention or feed-forward layer*

d. Prediction layer

After passing through self-attention blocks, we predict the next item score based on the output of the last self-attention block $F_t^{(b)}$. We simply take the product of this vector and the item embedding M as the prediction score for the items.

$$r_{i,t} = F_t^{(b)} M_i^T$$

4. Loss Functions

There are multiple functions that can be used to optimize and train the models we mentioned above. Below are the three losses that we implemented and experimented with.

4.1. Cross Entropy Loss

This is the standard Cross Entropy Loss used in classification problems.

$$-\sum_{S^u \in S} \sum_{t \in [1,2,\ldots,n]} \left[\log(\sigma(r_{o_t,t})) + \sum_{j \notin S^u} \log(1 - \sigma(r_{j,t})) \right].$$

S^u: *user sequence*
$r_{j,t}$: *relevance of item j appearing given the first t items*
o_t: *expected output after time step t*
Using the cross entropy loss

4.2. BPR Loss

$$L_s = -\frac{1}{N_S} \cdot \sum_{j=1}^{N_S} log\left(\sigma\left(r_{s,i} - r_{s,j}\right)\right)$$

N_S: *sample size*
$r_{s,k}$: *score on item k at the given point*
i : *target item*
j : *negative samples*

This loss function is a pairwise ranking loss, which compares the score of the positive item with the score of a sample of negative items, and returns the average difference. This loss aims to learn an accurate personalized ranking of all items for the user, where negative items are forced to be close to 0.

4.3. TOP 1 Loss

$$L_s = \frac{1}{N_S} \cdot \sum_{j=1}^{N_S} \sigma\left(r_{s,i} - r_{s,j}\right) + \sigma\left(r_{s,j}^{\ 2}\right)$$

This ranking loss is the regularized approximation of the relative rank of the target item, which is specified as: $\frac{1}{N_S} \cdot \sum_{j=1}^{N_S} I\{r_{s,j} > r_{s.i}\}$, where $I\{\cdot\}$ is approximated by a sigmoid function.
Since there are items in the dataset that are both a negative sample and a positive sample, which leads to the score becoming increasingly high. To combat this, the author added a L2 regularization term at the end.

III. Experiments And Results

1. Dataset

The dataset we use in this project is the Amazon Video Games dataset, which contains the sequential purchases of video games of over 24303 Amazon users. The dataset also contains meta-information about the video games, such as the games' ratings, descriptions… However, in this project we only utilize the historical sequence of video game purchases of the users, since all of our algorithms only require those information.
Here are the summary of our dataset:

Number of users	24,303
Number of games	50,953
Number of interactions	231,780
Average sequence length	6,88

2. Data Preprocessing And Augmentation

2.1. Data Preprocessing

Firstly, each video game in our dataset is encoded with an unique index from 0 to 50,953. Thus, an users' interaction history is represented as a list of indices.
To create the train, validation and test sets, we follow the "leave-one-out" evaluation procedure commonly used in sequential recommendation. For every sequence, we leave the last item out as target item for testing, the second last item for validation, and the rest are used for training. An example of this type of splitting is demonstrated below:

	Input sequence	**Target Item**
Full sequence	1,2,3,4,5,6	
Train	1,2,3	4
Validation	1,2,3,4	5
Test	1,2,3,4,5	6

In order to feed the data into our models, which receive fixed size tensors as input, we need to make all of the input sequences of the same length. We set the fixed sequence length to be 20, any sequences longer than this will be broken down into multiple smaller sequences with length equals or less than 20. In contrast, sequences shorter than the chosen length will be padded with 0s at the end.

2.2. Data Augmentation Methods

In addition to the 3 algorithms presented above, we also experimented with 3 different data augmentation methods for sequential recommendation algorithms. Since the size of this dataset is decently big and diverse, applying data augmentation might not yield significantly greater results. However, with little training data, models tend to easily overfit and lead to poor performances, this is where data augmentation might make a greater difference.

a. Noise Injection

In this method, we augment our training data by adding a random negative item (item not included in the original sequence) into a random position of that sequence. However, we want to preserve the length of the original sequence, thus an item should be removed from the sequence. Since studies have shown that further-away items tend to have less impact on the prediction, we remove the first item of the sequence.

Original sequence	2,4,6,8,10,12
Augmented sequence	4,6,3,10,12

b. Redundancy Injection

This data augmentation method adds a random positive item (item in the original sequence) into a random position different from the items' original position of the sequence. Similarly, we need to remove the first item of the original sequence.

Original sequence	2,4,6,8,10,12
Augmented sequence	4,10,8,10,12

c. Item Mask

We randomly choose *p%* items of the original sequence and mask them to exclude them from the training input. In this project, we set *p = 30%.*

Original sequence	2,4,6,8,10,12,14
Augmented sequence	2,0,8,10,0

3. Experiments

3.1. Method

a. Models' experiment

To compare the performances of our main three models, we train, validate, and test their results on the same split dataset. Additionally, we also compare the models' results with a simple baseline POP: a recommender who always outputs the most popular item in the train set.

In order to keep the comparison fair, we set the item's embedding size of all models to be 64. Due to time and resource constraint, we weren't able to properly tune all models' hyper-parameters, so ínstead we used hyper-parameters recommended by the models' respective original papers.

All models are trained for 200 epochs, and early stop if there are no improvements on the validation set for more than 20 epochs. The models' checkpoint is saved at the epoch with the best validation result.
All models are trained on the Google Colab Environment, with the two deep learning algorithms GRU4REC and SASREC utilizing its GPU resources.
For this experiment, we choose the BPR loss as the default loss function for all three models.

b. Data augmentation experiment

We choose the model with the best performance, and train that model on data augmented by the three augmentation methods mentioned above.
As mentioned above, training only a small dataset might make data augmentation methods more beneficial. Thus, we also try training the model on only a fraction of the

original data, with or without augmentation. The portions we choose are 10%, 25% and 50% of the original data, selected randomly.
For all methods, the size of the augmented data is the same as the original data picked for training.
The hyper-parameters and training setups are similar to the previous experiment.

c. Loss functions' experiment

We choose the model with the best performance, and train that model with the three different loss functions that we implemented.
The hyper-parameters and training setups are similar to the previous experiment.

3.2. Metrics

We pick 2 main commonly used metrics in recommendation systems as the evaluator for our project: Hit Rate (HR@10) and Normalized Discount Cumulative Gain (NDCG@10).

Hit Rate

Hit Rate calculates the percent of the ground-truth items that are among the top K recommended items by the model, or in other words the items with the highest predicted probabilities. Here, we choose top K = 10

Normalized Discount Cumulative Gain (NDCG)

This metric measures how relevant the recommended results are, which takes into account the rank of the actual item among the predicted items. The higher the rank of the actual item is in the top K recommended items, the higher this score will be.
First, we calculate the top K Cumulative Gain, which is the sum of gains of the first K items recommended.

$$CG_{@K} = \sum_{i=1}^{K} G_i$$

Here G_i is the “gain” of the item recommended ranked number i. If it's the target item then $G_i = 1$, else $G_i = 0$.
Next, we can calculate the Discount Cumulative Gain, which gives more weight to items recommended at the top.

$$CG_{@K} = \sum_{i=1}^{K} \frac{G_i}{log_2(i+1)}$$

Finally, NDCG is measured by the DCG divided by a normalization factor, which is the ideal DCG score in the case where the actual item is the highest recommended item.

$$NDCG_{@K} = \frac{DCG_{@K}}{IDCG_{@k}}$$

$$IDCG_{@K} = \sum_{i=1}^{K^{ideal}} \frac{G_i^{\ ideal}}{log_2(i+1)}$$

3.3. Results

a. Models' results

Algorithm	HR@10	NDCG@10
POP	0.3826	0.2186
FPMC	0.6538	0.4217
GRU4REC	0.6821	0.4446
SASREC	**0.6898**	**0.4669**

Firstly, we can see that all 3 of our models all achieve much better performance compared to the simple baseline POP, which is always recommending the most popular item of the training set. The lowest-performing model of the 3, FPMC, has a NDCG@10 score of almost twice as POP's score.

We can see that the 3 models' effectiveness are quite close, with none of them significantly outperforming the rest. These 3 models all have different ways of successfully capturing the sequential nature of users' behaviors: FPMC with Markov Chain and Matrix Factorization, GRU4REC with RNN layers, SASREC with self-attention mechanism

The method with the highest result is SASREC, which is to be expected since the self-attention mechaním has proven to be a more effective method of learning sequential data compared to Markov Chains or RNN-based architectures. In Markov

Chains, the current state is only dependent on the previous (or a previous few) states, which can lose a lot of long-term dynamic information. RNN's connectionist architectures make them susceptible to the vanishing gradient problem, and might prove to be not effective against long-term interactions. In contrast, the self-attention layer ensures that our model can capture the relationship between every-pair of items within the sequence, which helps leverage many sequential patterns across the sequence.

b. Data augmentation results

Metrics: NDCG@10

Size	None	Noise Injection	Redundancy Injection	Item Mask
10%	0.2558	**0.3118 (+21.9%)**	0.3102 (+21.3%)	0.2585 (+1.1%)
25%	0.3587	**0.3963 (+10.4%)**	0.3961 (+10.4%)	0.3564 (-0.7%)
50%	0.4221	**0.4407 (+4.2%)**	0.4400 (+4.2%)	0.4365 (+3.4%)
100%	0.4669	0.4702 (+0.7%)	**0.4710 (+0.8%)**	0.4667 (-0%)

As we can see from the above table, data augmentation seems to have a more noticeable effect on the models' performance when they are trained on a small dataset, since augmentations' regularization effect helps reduce the problem of overfitting. On 10% and 25% of the dataset, the best performing method helps improve the result by 21.9% and 10.4%, respectively.
As the training data gets larger, the effect of augmentation decreases. On 50% of the training data, the best performing method only raises the result by 3.4%. And on the full training data, the difference in performance is almost inconsequential.
Among the three augmentation methods, Noise Injection and Redundancy Injection proved their effectiveness, with noticeable boosts in performance on sparse training data. This might be due to the fact that these methods help our models capture the skipping patterns in short-term user interactions, which is valuable if little data is accessible.

However, the Item Mask method doesn't seem to achieve its purpose, with little to no improvement even on only 10% of training data. We hypothesize that the reason for this is that by masking a portion of the sequence during training, we are making the input become more noisy and harder to extract patterns from. We tried lowering the percentage of masked items in a sequence, but none of them seem to achieve a desirable result.

c. Loss function results

Loss function	**HR@10**	**NDCG@10**
Cross Entropy	0.6898	0.4645
BPR	0.6898 (+0%)	0.4669 (+0.5%)
TOP 1	**0.7163 (+3.8%)**	**0.4847 (+4.3%)**

As we can see, BPR doesn't seem to offer a greater performance compared to regular cross entropy. On the other hand, TOP 1 showed improvements over the other two losses. One particular reason for this could be TOP 1's optimization of the target items' rank among the output predictions, which is an important criteria that the other two losses ignore.

IV. Conclusion

In summary, in this project our group had done research into sequential recommendation and its current methods: from its models, data preprocessing, evaluation… We implemented three main models in this project and evaluated their performance on the Amazon Video Games dataset:

- Factorizing Personalized Markov Chain (FPMC) utilizing both long-term user preference (matrix factorization) and short-term sequential dynamics (markov chains)
- GRU4REC: RNN-based model to model long-term user interactions
- SASREC: self-attention based model, an attention mechanism relating different positions of a single sequence, to capture the dynamic relationship between items within a sequence.

We observed all three of these models achieving acceptable results, with SASREC being the best performing model due to its powerful mechanism.

In addition to these, we also explored and experimented with data augmentation methods Noise Injection, Redundancy Rejection,and Item Mask. We found that Noise Injection and Redundancy Injection seems to be beneficial for the models' performance, especially trained on a small amount of data. In contrast, Item Mask proves to be ineffective, one possible reason for this is the noise this method introduces to the data.

Finally, we also explored some different types of loss functions used for model training. We found that TOP 1 loss helps improve the result over a standard Cross Entropy loss, which could be thanks to taking the target items' rank into consideration.

Our source code for this project is stored in this repo:
https://github.com/tuandung2812/Group-3---Seq_Rec.git

VI. REFERENCES

[1] Balázs Hidasi and Alexandros Karatzoglou. 2018. Recurrent neural networks with top-k gains for session-based recommendations. In Proceedings of the 27th ACM international conference on information and knowledge management. 843–852.

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

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

[4] Song, Joo-yeong and Suh, Bongwon. Data Augmentation Strategies for Improving Sequential Recommender Systems

[5] Amazon datasets

[6] Reference code for GRU4REC implementation

[7] Reference code for SASREC and FPMC implementation