Project Report

Part 1 Problem

1.1 Problem Description

The recommendation system of this project aims to provide personalized game recommendations for global game enthusiasts to improve user experience and help the system improve customer retention and increase game sales. By analyzing the browsing and consumption behavior of users, the system can recommend high-quality games that match the interests of users, thereby improving user satisfaction, platform stickiness, and helping game malls increase revenue. The recommendation system will be deployed on a web page, showing 150 games to users at a time through a web interface. This deployment method ensures a consistent user experience on different devices, and since Web technology is now widely used, deploying the web is a wise choice.

1.2 Competition Analysis

In the current field of game recommendations, there are several popular platforms, such as Steam Store, Xbox Game Pass, and PlayStation Now. This section will focus on the Steam Store.

The Steam Store's recommendation system mainly relies on machine learning technology to make personalized recommendations based on the user's game history. It provides game discovery functions, which can suggest new games that users may be interested in and recommend the next game worth playing from the user's game library. In addition, Steam also performs well in social recommendations, further improving the relevance and accuracy of recommendations through interaction and sharing between users. Through these strategies, Steam strives to help users discover more games that match their interests. However, Steam's recommendation system also has some limitations. Since the platform's main source of income is the commission from game sales, it tends to recommend many advertising games, which may affect the user experience. In addition, for non-paying players, the platform's recommendation mechanism may not be friendly enough, resulting in a limited experience for them.

1.3 User inputs and Recommendation

In this project, the data entered by users include their game history, game ratings, and wishlists. In addition, users can also choose to enter their preferences for certain game types or features, such as favorite game tags, keywords, etc. Through these input data, the system can fully understand the user's game preferences and provide more accurate recommendations. Recommendations will include newly released games, highly praised, most popular games, and user-preferred games. Recommendations will be provided in multiple scenarios, such as at the end of a game session, on the user's personal page, and in a dedicated recommendation area. In this way, users can get game recommendations that match their interests whether they are browsing the platform or after completing the game, thereby improving user experience and engagement.

User Feedback Capture and Utilisation

The system collects multiple types of user feedback, including game ratings, like/dislike options, and user interaction metrics (e.g., clicks, views, purchases, etc.). This feedback data is critical for optimising the recommendation system. Specifically, users' game ratings and like/dislike options directly reflect their satisfaction with the game, while users' interaction metrics reveal their actual behaviour and preferences.

The collected feedback data will be used to optimise the recommendation algorithm and update user profiles. By analysing this feedback, the system can continuously adjust the recommendation strategy to improve the accuracy and relevance of recommendations. For example, if a user shows a strong preference for a specific type of game, the system will prioritise such games in future recommendations. In addition, user feedback can help the system identify and filter out unpopular games, thus improving the overall recommendation quality.

Through these measures, our recommendation system is not only able to provide personalised game recommendations, but also to continuously improve and optimise the recommendation effect through real-time feedback from users, ensuring that users can always get the game recommendations that best meet their interests and needs.

1.4 Definition of Recommendation Problems

Our recommendation system mainly involves classification/prediction problems and ranking problems.

Classification/Prediction Problem

The recommender system first predicts which games are likely to be of interest to the user by capturing the user's historical data and personal preferences. This involves classification of user behaviour. In the Retrieval Layer of this system, the classification problem is implemented by predicting which games can be recommended and keeping

them. This step ensures that the games in the recommendation pool are those that are likely to be of interest to the user, thus improving the accuracy of the recommendations.

Ranking Problem

Ultimately, the recommender system needs to sort the games based on the predicted level of interest and estimated ratings to generate a personalised list of recommendations. This is a typical ranking problem, i.e., the games that the user is likely to be most interested in are ranked at the top of the recommendation list. The ranking is based on the predicted probability of interest and other relevant factors. In this system, the ranking layer is responsible for implementing this ranking problem and recommending the most appropriate games to the user by considering various features and user preferences.

By defining the recommendation problem as a classification/prediction and ranking problem, our system is able to comprehensively analyse user behaviours and provide highly accurate and relevant game recommendations to meet users' individual needs. This integrated approach ensures the accuracy and relevance of recommendations, increasing user satisfaction and user stickiness of the platform.

1.5 User Interface and Interaction Design

Two user interface models are designed for this system.

- Homepage User Recommendation (Figure 1-1): Displays a personalized game recommendation list, and users can enter the game details page by clicking on the game they are interested in.
- Game Details Page (Figure 1-2): Provides detailed information about the game, including game operation buttons (such as "like" and "buy"), and users can perform operations to reflect their preferences. The page also includes similar game recommendations and user reviews to increase user interaction and participation.

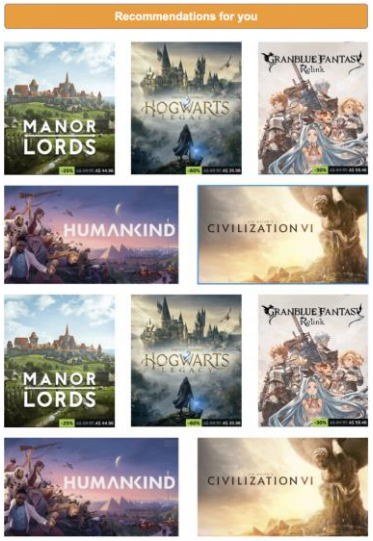

Figure 1-1

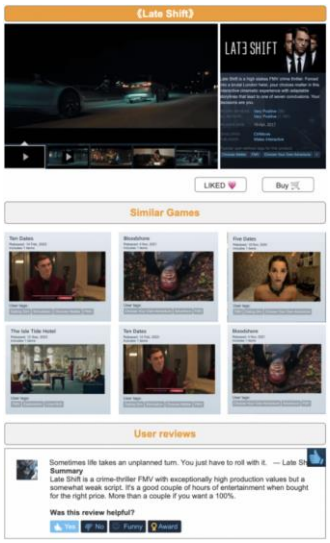

Figure 1-2

Part 2 Datasets

2.1 Introduce

This project utilizes the dataset "Game Recommendations on Steam" from Kaggle (https://www.kaggle.com/datasets/antonkozyriev/game-recommendations-on-steam). The dataset consists of four files

Games.csv: A table containing information about games (see Figure 2-1).

app_id	title	date_release	win	mac	linux	rating	positive_ratio	user_reviews	price_final	price_original	discount	steam_deck
1541020	Kakele Online - M...	2021-04-15	true	true	true	Very Positive	84	158	0.0	0.0	0.0	true
1566230	Turn on all the l...	2021-03-31	true	false	true	Very Positive	93	60	0.99	0.99	0.0	true

Figure 2-1 Data Preview of game.csv

- app_id is the id of the game, it is the real id of the steam game platform, you can find the corresponding game details page through this id.
- date_release is the release date of the game
- win/mac/linux is the system support of the game.
- rating is the overall rating of the game
- user_reviews is the number of times the game has been reviewed
- price_original, price_final are the original and discounted prices of the game.
- discount is the number of discounts for the game
- Translated with DeepL.com (free version)

games_metadata.json: Corresponding detailed information about the games ingames.csv, such as tags and descriptions(see Figure 2-3).

app_id	description	tags
1240050	AD 673. Camelot has fallen. The Round Table must rally to save King...	[Narrative, Strategy, Turn-Based Tactics, Choices Matter, Replay Va...
1263200	Fight Spartans at Brazil Island with fiercely boxing combos on this...	[Side Scroller, Precision Platformer, 2D Platformer, Mystery Dungeo...

Figure 2-2 Data Preview of games_metadata.json

- tags_id is the tags included in the game
- description is the description of the game

Recommendations.csv: A table of user reviews indicating whether a user recommends a product. This table represents a many-to-many relationship between game entities and user entities. The data is crucial for understanding user preferences and for personalizing recommendation systems (see Figure 2-3).

is_recommended	user_id	app_id	date	hours
false	6473513	213610	2021-09-30	3.0
true	2185259	213610	2021-03-29	81.5

Figure 2-3 Data Preview of Recommendations.csv

- is_recommended is an indication of whether or not the game app_id is recommended by user user_id.
- date is the date the user made the recommendation.
- hours is the amount of time the user has played the game.

Users.csv: A table of public user profile information, including the number of purchased products and reviews published (see Figure 2-4).

user_id	products	reviews
7360263	359	0
14020781	156	1
8762579	329	4

Figure 2-4 Data Preview of Users.csv

2.2 Justification

- The dataset is sourced from the Steam Store (https://store.steampowered.com/), a leading online platform for purchasing and downloading video games and other game-related content, ensuring the authority and reliability of the data.
- The dataset has been cleaned and pre-processed to contain over 41 million user recommendations (reviews), and real app_id, which ensures quality and consistency.

2.3 Exploratory Data Analysis (EDA)

This section will provide an in-depth analysis of each file with statistics and visualisation of features.

Games.csv

```
root
 |-- app_id: long (nullable = true)
 |-- title: string (nullable = true)
 |-- date_release: string (nullable = true)
 |-- win: boolean (nullable = true)
 |-- mac: boolean (nullable = true)
 |-- linux: boolean (nullable = true)
 |-- rating: string (nullable = true)
 |-- positive_ratio: long (nullable = true)
 |-- user_reviews: long (nullable = true)
 |-- price_final: double (nullable = true)
 |-- price_original: double (nullable = true)
 |-- discount: double (nullable = true)
 |-- steam_deck: boolean (nullable = true)
```

Figure 2-5 Data Structure of game.csv

By analysing the app_id field, i find that the game.csv file contains 50,872 unique game records. In addition, Figure 2-5 shows the data structure and field types of game.csv, where the rating field is of string type. After counting all the values in this field, it is determined that it contains the following categories: 'Mostly Positive', 'Mostly Negative', ' Overwhelmingly Negative', 'Very Negative', 'Overwhelmingly Positive', ' 'Very Positive', 'Mixed', 'Negative', 'Positive '. To support subsequent data processing and analysis, i converted these string-type ratings to numeric form. To do this, a Rating_Map (see Figure 2-6) was created to map ratings to numeric values. The descriptive statistics of the numerical rating field are shown in Figure 2-7, where i can see that the median rating is 7.0, which indicates that 50% of the games are rated as "Positive". Figure 2-8 shows the distribution of the number of games by rating, showing that the highest number of games are rated 7 and 8, while only a few games are rated 1 and 2.

```python
RATING_MAP = {
    'Overwhelmingly Positive': 9,
    'Very Positive': 8,
    'Positive': 7,
    'Mostly Positive': 6,
    'Mixed': 5,
    'Mostly Negative': 4,
    'Negative': 3,
    'Very Negative': 2,
    'Overwhelmingly Negative': 1
}
```

Figure 2-6 Rating Map

```
count    50872.000000
mean         6.511794
std          1.302575
min          1.000000
25%          5.000000
50%          7.000000
75%          8.000000
max          9.000000
Name: numeric_rating, dtype: float64
```

Figure 2-7 Rating Describe

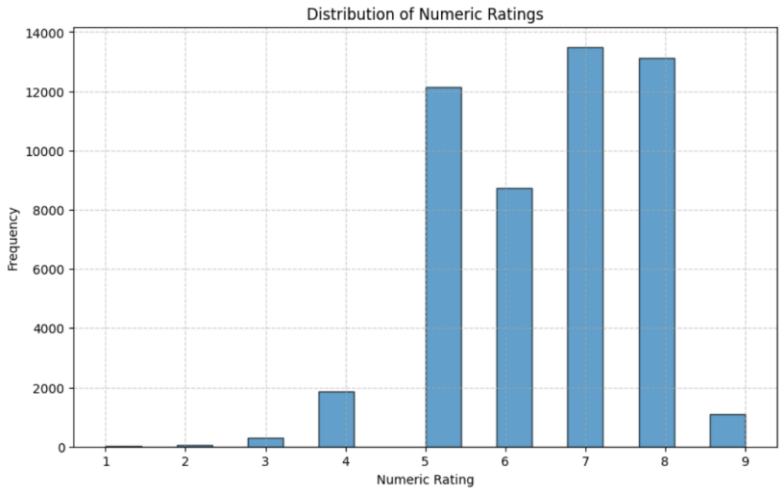

Figure 2-8 Distribution of Numeric Rating

games_metadata.json

Through statistical analysis, i found a total of 441 unique tags. To further understand the distribution of tags, I counted the distribution of tags and generated a histogram of the top 50 tags.Figure 2-Top 50 tag distribution shows the distribution of these tags, in which the top three tags are "indie", "singleplayer" and "action", in which the number of "indie" is close to 30,000, The top three tags are "indie", "singleplayer", and "action", and the number of "indie" is close to 30,000.

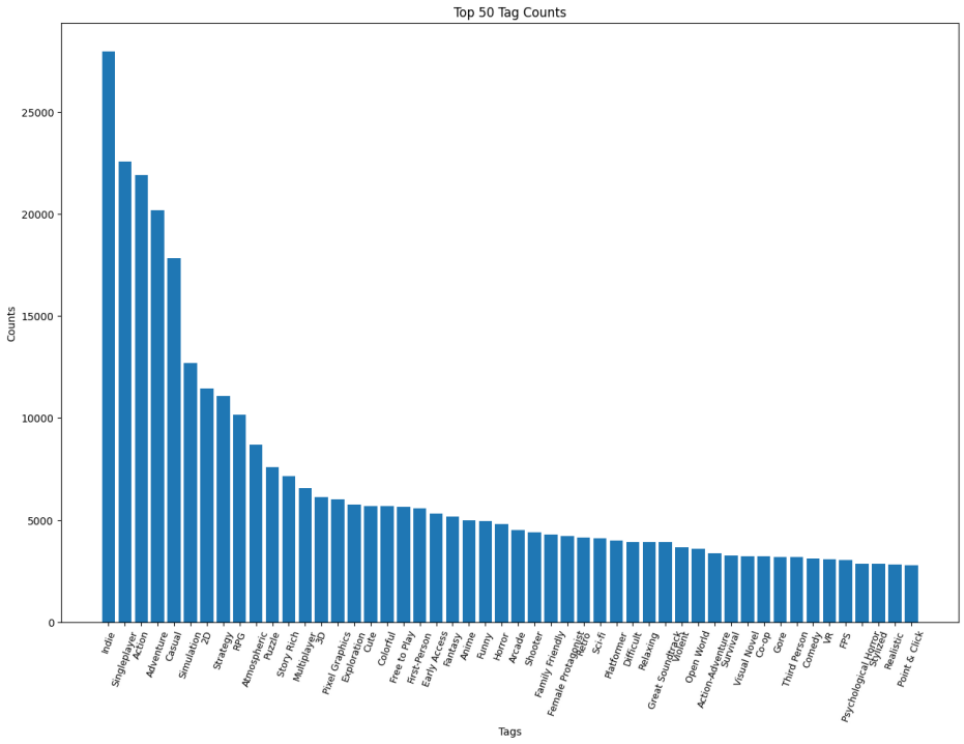

Figure 2-9 Top 50 tag distribution

Recommendations.csv

Through statistical analysis, this dataset contains 41,154,794 rows of user-game interaction data, covering 13,781,059 unique users and 37,610 unique games, providing a comprehensive picture of user-game interactions.

```
count    13781059.00
mean           2.99
std            8.12
min            1.00
25%            1.00
50%            1.00
75%            3.00
max         6045.00
Name: app_id, dtype: float64
```

Figure 2-10 user reviewed games

```
count    41154794.00
mean          100.60
std           176.17
min             0.00
25%             7.80
50%            27.30
75%            99.20
max          1000.00
Name: hours, dtype: float64
```

Figure 2-11 time users spent playing

Figure 2-10 provides a detailed description of the user interaction data: the most frequently interacted users engaged with as many as 6,045 games, while the least frequently interacted with only 1. Figure 2-11 describes the amount of time users spent playing: the users who spent the most time playing invested a total of 1,000 hours, with an average of 100.6 hours per game.

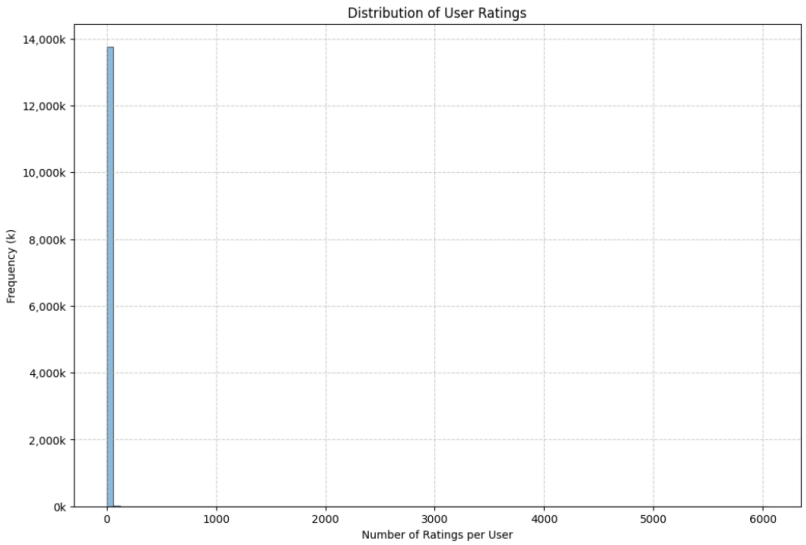

Figure 2-12 User Rating Count Distribution

I grouped the user_id and counted the number of game interactions for each user, and the results were plotted in a histogram (shown in Figure 2-12). This graph helps us to analyse the distribution of user activity. As can be seen from the figure, the distribution of data is extremely uneven, with most users having only a small number of interaction records, while a very small number of users have a large number of game interaction records.

To further quantify the analysis, I performed numerical statistics on these data. Figure 2-13 shows the descriptive statistics of the number of user interactions. Figure 2-13 shows the descriptive statistics of the number of user interactions. The results show that at least one user interacted with 6,045 games. The average number of user interactions was 2.99 games, while the median was 1 game.

```
count    13781059.00
mean            2.99
std             8.12
min             1.00
25%             1.00
50%             1.00
75%             3.00
max          6045.00
dtype: float64
```

Figure 2-13 User Rating Count Description

2.4 Advantages and disadvantages of the dataset

After performing Exploratory Data Analysis (EDA) on the data, i found that the recommendation.csv file contains 12,781,059 user interactions, which is a large and very rich amount of data. This provides a solid foundation for the subsequent model in capturing user preferences and features. In addition, by combining the

game_metadata.json and game.csv files, complete information about the game features can be obtained, including the game type, release date, labels, descriptions, and so on. This information is crucial for understanding game features and user preferences.

However, as can be seen from the graph, the data distribution in recommendations.csv is extremely uneven. Some popular games have a large number of rating records, while many games have a very small number of ratings. This imbalance may affect the coverage of the recommendation system, making it difficult for some games to make it to the recommendation list. The data also shows a clear long-tail effect: a few popular games receive attention and ratings from the majority of users, while a large number of cold games receive almost no ratings. This may cause the recommendation system to favour popular games, thus reducing the diversity of the recommendation results. Although game.csv provides rich information about game features, recommendations.csv lacks contextual information about users, such as their mood, time, and environment while playing. This contextual information is crucial for improving the personalisation and accuracy of recommendations, but is not sufficiently considered in the existing data.

Part 3 System Architecture

In this section, the composition of the whole system as well as the workflow will be outlined, and the use and evaluation of the method will be described in detail in Part4 Methods. The system functionality can be divided into four major modules, namely the data processing layer, the Retrieval Layer, the ranking layer and the evaluation layer. The following is a detailed description of each module and its main responsibilities.

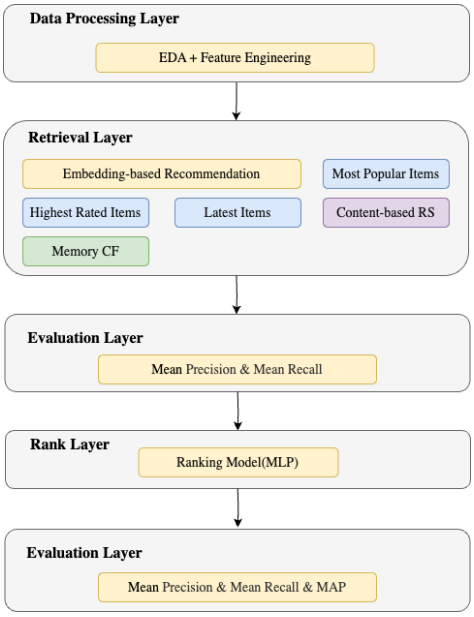

Figure 3-1 System Structure

3.1 Data Processing Layer

The data processing layer is mainly responsible for data preprocessing and feature engineering. Specific tasks include:

- Exploratory Data Analysis (EDA): initial exploration of the dataset to understand the basic structure and distribution of the data.
- Feature extraction: filtering useful features from the raw data, such as users' game preferences and game features. These features will be used for subsequent modelling and recommendation.
- Data Transformation: Convert the data into a format suitable for model training, e.g. encode tag features into multihot encoding, normalise numerical features, numericalize Boolean and text features, etc.

3.2 Retrival Layer

The Retrieval Layer is responsible for generating initial recommendations by filtering a large dataset to identify potential items for recommendation. This layer employs several methods, including:

- Embedding-Based Recommendation: This approach involves using DeepWalk to capture user game sequences, which are then fed into a Word2Vec model to train and generate user and game embeddings. Recommendations are made through approximate nearest neighbor search, allowing the system to capture deep relationships between users and games.
- Content-Based Recommendation: This method recommends games by comparing the content features (such as genres and tags) of games that a user likes with those of other games.
- Memory-Based Collaborative Filtering (CF): Collaborative filtering is used to recommend games by leveraging user behavior data, such as ratings, to find similar users or items.
- Latest Publish Items: This method recommends newly released games, helping users discover the latest content.
- Most Popular Items: This method recommends the most popular games, based on metrics such as download numbers and the frequency of ratings.
- Highest Rating Items: This method recommends the highest-rated games, based on user review data.

The strategies of recommending Latest Publish Items, Most Popular Items, and Highest Rating Items effectively address the cold start problem. In scenarios where user

preferences are unknown, recommending high-quality, popular games ensures user retention and provides more opportunities to observe user preferences.

3.4 Ranking Layer

The ranking layer's primary task is to rank the recommendations generated by the Retrieval Layer, ensuring the relevance and personalization of the final recommendation list. Specific tasks include:

- Model-Based Ranking: This involves training an MLP model using the features of games that align with user preferences. The model predicts the relevance of the Retrieval Layer's results and ranks them accordingly, further enhancing the accuracy of recommendations.

3.5 Evaluation Layer

The evaluation layer is responsible for assessing the methods and models used in both the recall and ranking layers to ensure the effectiveness and performance of the recommendation system. Specific tasks include:

- Evaluation Metrics: The performance of recommendations from the recall and ranking layers is evaluated using metrics such as Mean Precision, Mean Recall, and Mean Average Precision (MAP).
- Offline Evaluation: The ranking models are subjected to offline evaluations using historical datasets to ensure the reliability and stability of the models before they are deployed in a live environment.

Part 4 Methods

In Part 3, i outlined the system architecture and explained the foundational modules. In this section, i will delve into the method modules, discussing the feature processing steps and the recommendations generated by each method. This report will focus on two key methods:

- Retrival Layer - Embedding-Based Recommendation
- Ranking Layer - Deep Ranking Recommendation Model

For each method, i will detail the feature processing steps and provide specific results and analyses of the recommendation evaluations.

4.1 Embedding Based Recommendation

As mentioned in the system architecture in Part 3, this method is part of the Retrieval Layer within the system. This section will focus on the data selection, method introduction, Top-K evaluation under different interaction data volumes, and comparative analysis of the performance metrics.

4.1.1 Dataset Selection

is_recommended	user_id	app_id	date	hours
false	6473513	213610	2021-09-30	3.0
true	2185259	213610	2021-03-29	81.5

In Part 2.3 (EDA), i identified that the recommendation.csv file contains 12,781,059 user interaction records. For the experiments, i utilized four subsets of the interaction data, each cut from a different perspective:

1. Full Interaction Data: This subset includes all user-game interaction data, with the goal of comprehensively evaluating the method's performance.
2. 80% User Interaction Data: This subset is created by splitting each user's interaction data, using 80% of their interaction records as training data. The purpose is to assess the recommendation system's predictive ability without data leakage. For example, if a user has interacted with 10 games, 8 of these interactions will be included in the training set. This consistent split ensures that each user has sufficient representation in both the training and test sets, allowing the model to learn general behavior patterns across all users.
3. Top 10% Most Active Users' Interaction Data: This subset includes only the interaction data of the top 10% most active users, determined by the number of games they have interacted with. This subset is used to evaluate how the method performs when focusing on the most engaged users.
4. Top 5% Most Active Users' Interaction Data: This subset includes only the interaction data of the top 5% most active users. The experiment aims to observe whether the more concentrated feature information in a smaller subset leads to an improvement in performance metrics.

In the initial experiments, I used the full dataset for analysis. However, I found that processing the full dataset required an enormous amount of computation and was extremely time-consuming. Consequently, I decided to gradually reduce the dataset size and observe how these smaller datasets impacted the performance metrics. During this process, I compared the interaction data of the Top 5% and Top 10% most active users

against the full dataset. I generated a stacked chart to visualize the relationship between these datasets. The Figure 4-1 reveals that the data from the Top 5% most active users alone accounts for 38.15% of the full dataset, which still represents a substantial amount of data.

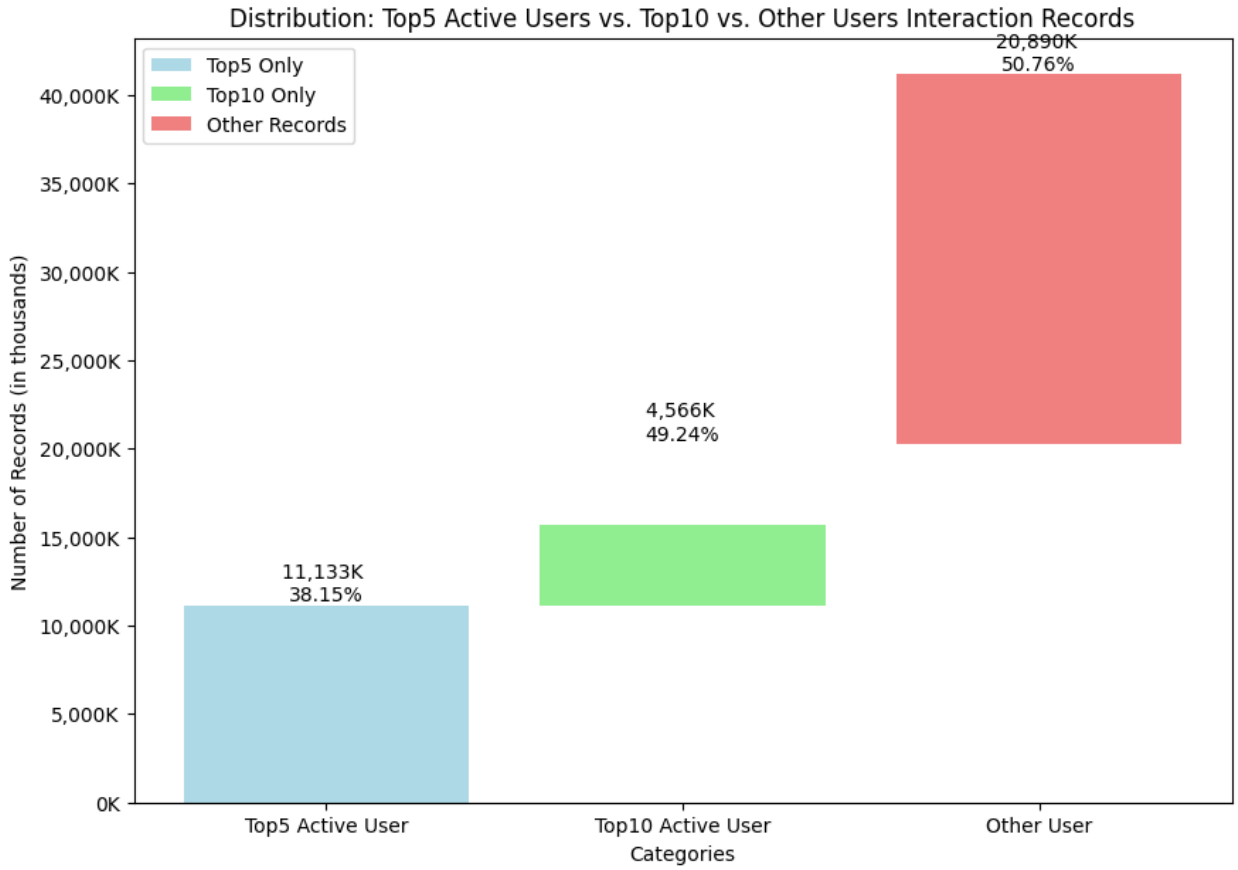

Figure 4-1 System Structure

4.1.2 Method Introduction

This section provides a detailed overview of the methods and their applications within the recommendation system. The following outlines the sequence of steps and the methodologies employed:

- Generate Game and User Embeddings (see 4.1.2.1): The first step involves generating embeddings for both games and users. This is accomplished by training on subset data to create game embeddings and user embeddings.
- Apply Locality-Sensitive Hashing (LSH) (see 4.1.2.2): Next, LSH is used to partition the generated embeddings, facilitating efficient approximate nearest neighbor search. This step significantly accelerates computations in large-scale datasets.

- Generate Game Recommendations (see 4.1.2.3): With the partitioned embeddings, personalized game recommendations are generated for users based on their embeddings and the corresponding game embeddings.
- Evaluating the generated recommendations(see 4.1.2.4): the game recommendations for users are evaluated using mean precision and recall.

All mentioned functions can be found in the Project Implementation.

4.1.2.1 Generating Game and User Embeddings

The build_embedding() method is used to generate embeddings for games and users. The primary function of this method is to convert game and user features into embedding vectors, which can then be used for game recommendations and evaluation within the recommendation system.

```
# key code
games_embedding_map, users_embedding_map = build_embedding(spark_review_df, sampling_times = 10000,
walk_depth = 40)
```

Parameter Descriptions

- spark_review_df: This DataFrame contains user interaction data, which can be derived from any of the four subsets mentioned in section 4.1.1. By constructing embeddings from different subsets, various experiments can be conducted, and the results can be compared across four key metrics.
- sampling_times: This parameter specifies the number of game sequences sampled in Step 4. The default value is 10,000, indicating that 10,000 game sequences will be sampled.
- walk_depth: This parameter represents the depth of each game sequence, i.e., the length of each sequence. The default value is 40, meaning each game sequence will contain 40 game IDs (app_id).

Thus, setting sampling_times = 10,000 and walk_depth = 40 indicates that 10,000 game sequences will be sampled, with each sequence comprising 40 game IDs.

Return Values

- games_embedding_map: A dictionary where the keys are game IDs (app_id) and the values are the corresponding game embedding vectors. This map represents the embedding of each game.
- users_embedding_map: A dictionary where the keys are user IDs (user_id) and the values are the corresponding user embedding vectors. The user embeddings are

computed by averaging the embedding vectors of the games the user prefers. This map represents the embedding of each user.

Step1: Constructing the Co-occurrence Matrix

The first step is constructing a co-occurrence matrix based on users' game records. The co-occurrence matrix counts the frequency with which games co-occur in users' game sequences, helping to understand the relationships and similarities between different games. The parallel_game_matrix_processing() function is the entrance for computing the Co-occurrence Matrix:

```
game_matrix = parallel_game_matrix_processing(game_sequences)
```

the parallel_game_matrix_processing() function calculates the co-occurrence matrix through parallel processing, enabling rapid matrix construction in a large-scale data. Next, it will calculate the co-occurrence frequency for each pair of games. The specific steps are as follows:

```
for x in range(user_play_game_num):
    for y in range(x + 1, user_play_game_num):
        game_a_id = sequence[x]
        game_b_id = sequence[y]
        if game_a_id != game_b_id:
            local_matrix[game_a_id][game_b_id] += 1
            local_matrix[game_b_id][game_a_id] += 1
```

In this code snippet, each pair of app IDs is iterated over, and the co-occurrence frequency in the local matrix is updated. user_play_game_num represents the number of games the user has played, and sequence is the user's game sequence. By examining each pair of game IDs, their counts in the co-occurrence matrix are incremented.

Finally, all local matrices are merged to form a global co-occurrence matrix, which allows us to determine the co-occurrence frequency of each pair of games and further analyze their relationships.

Step2: Calculating Transition Probability Matrix

After constructing the co-occurrence matrix, the next step is to calculate the Transition Probability Matrix. This matrix describes the probability of a user transitioning from one game to another, which is a critical component in the recommendation system as it helps predict the next game a user might be interested in.

```
def compute_trans_probs(games_chain):
    apps_id = list(games_chain.keys())
```

```
        total_weight = sum(games_chain.values())
        probs = [games_chain[id] / total_weight for id in apps_id]
        return {'apps_chain': apps_id, 'probs': probs}


def build_trans_prob_matrix(game_matrix):
        """构建转移概率矩阵."""
        trans_prob_matrix = {}
        for app_id, games_chain in game_matrix.items():
        result = compute_trans_probs(games_chain)
        trans_prob_matrix[app_id] = result
        return trans_prob_matrix
```

The core function, compute_trans_probs, calculates the transition probability for each game. Specifically, it computes the probability of transitioning from the current game to all neighboring games. The returned result includes two parts:

- apps_chain: An array of IDs for all neighboring games of the current game.
- probs: An array of transition probabilities for each neighboring game, calculated by dividing the weight of each neighboring game by the total weight of all neighboring games of the current game.

In essence, the compute_trans_probs function determines the transition probability from the current game to each neighboring game by calculating the ratio of the weight of each neighboring game to the total weight of all neighboring games. This probability reflects the likelihood of a user transitioning to each neighboring game after selecting a particular game. These transition probabilities are used in subsequent DeepWalk algorithms to simulate user behavior and predict the user's next choice.

Step3: Calculating Game Entry Probability

Next, i calculate the game entrance probability. The game entry probability reflects the likelihood that a user will enter each game for the first time, which is crucial for understanding the user's initial points of interest. With this probability, i can identify which games are the user's first choice or preference.

```
# key code
all_app_ids, entry_game_probs = build_entry_games_probs(trans_prob_matrix, game_matrix)
```

In this code, the build_entry_games_probs function uses the transfer probability matrix (trans_prob_matrix) and the game co-occurrence matrix (game_matrix) to calculate the entry probability for each game. The returned all_app_ids contains the IDs of all games, and entry_game_probs corresponds to the entry probabilities for each game. Further, entry probabilities are used to measure how often a user selects a game for the first time in their sequence of games. A higher entry probability indicates that the game is more likely to be the user's first choice or initial point of interest in their game sequence.

Step4: DeepWalk Sample Game Sequence

In this phase, i generate game sequences by random walks. Through random walks, it could able to simulate the user's gaming behaviour sequence and based on that it can predict the user's future gaming preferences. In this code, deep_walks function is used to generate multiple game sequences.

Parameter description:

- sampling_times: the number of game sequences to be generated. It simulates sampling_times a sequence of user game behaviours. In this procedure, the default value is 10,000, which means that the system will generate 10,000 user game sequences based on a probability distribution. These sequences are used to capture user transfer patterns between games.
- walk_depth: defines the length of each generated game sequence, i.e. the number of games contained in each sequence. walk_depth determines the depth of the games in a single sequence, which affects the accuracy of the modelling of user behaviour. The value sets how many games each user will experience in their game sequence.
- all_app_ids: a collection of unique identifiers (IDs) for all games. the list of all_app_ids is used as a starting point during random walks to ensure that all games have the potential to appear in the generated sequences.
- entry_game_probs: represents the entry probability of a game, i.e., the probability that a user will select each game for the first time. This probability reflects the user's initial interest in each game and provides the basis for weighting the generated sequence.
- trans_prob_matrix: a transfer probability matrix that describes the probability of transferring from one game to another. It is calculated based on the user's game behaviour records and is used to simulate the user's transfer behaviour between games, helping to generate more realistic game sequences.

```
# key code
games_habits = deep_walks(sampling_times, walk_depth, all_app_ids, entry_game_probs, trans_prob_matrix)
```

The result returned by the function: games_habits is a rich sequence of games generated when simulating the user's gaming behaviour, providing data support for subsequent recommendations.

Step5: Word2Vec training game sequence generation embedding

```
# key code
```

```
vec_model = build_word2vec_model(spark, games_habits, size = 150, iter = 15, wsize = 20)

def build_word2vec_model(spark, games_habits, size = 100, iter = 15, wsize = 20):
        habit2vec = Word2Vec(vectorSize=size, maxIter=iter, windowSize=wsize, minCount=2, stepSize=0.025
```

In the build_word2vec_model function, size, iter, and wsize are hyperparameters of the Word2Vec model. The following is a detailed explanation of these parameters:

- size (vectorSize): indicates the dimension of each word vector, i.e., the length of the word vector. This parameter determines how many values will be included in the representation of each game ID (app_id) in the vector space. The higher the dimension, the more complex features the model can capture, but it may also increase the computational overhead and risk of overfitting.
- iter (maxIter): indicates the number of iterations to train the model, i.e. the number of rounds the model is trained on the dataset. A higher number of iterations usually allows the model to learn patterns in the data better, but it also increases the training time. Therefore, a balance needs to be found between training quality and computational resource consumption.
- wsize (windowSize): indicates the size of the context window considered during training. The context window determines how many words before and after each word the model will consider to predict the target word during training. For example, if windowSize is set to 20, when training an app_id in a sequence, the model will consider 20 app_ids before and after the sequence as the context to predict the current app_id. the choice of window size has a direct impact on the scope of contextual information captured by the model. A smaller window captures more local context information, while a larger window captures a wider range of context information.

Step6: Calculate game embedding and user embedding

Briefly, User embedding is obtained by arithmetically averaging the game embedding of the user's preferred games. First, i use the trained Word2Vec model to obtain the game embedding:

```
# use trained Word2VecModel
item_vec = vec_model.getVectors().collect()
games_embedding_map = {item.word: item.vector.toArray() for item in item_vec}
```

Next, a function is defined to construct the user embedding: in this function, it constructs a sequence of game embeddings for the user by extracting the embedding for each app_id from games_embedding_map. Then, the arithmetic mean of all non-empty game embeddings is computed to get the final embedding representation of the user.

```
def build_user_embedding(user_game_sequence):
        user_profile = [games_embedding_map.get(app_id, []) for app_id in user_game_sequence]
        user_embedding = np.mean([x for x in user_profile if len(x) > 0], axis=0)
```

4.1.2.2 LSH performs approximate nearest neighbour search for embedding

This method uses the FAISS (Facebook AI Similarity Search) library. faiss.IndexLSH is an index type for Locality-Sensitive Hashing (LSH).LSH is an efficient approximate nearest-neighbour search algorithm mainly used for fast similarity searches in large-scale datasets. The constructor of faiss.IndexLSH accepts several parameters, of which nbits and bucketSize have the following meanings. nbits is the number of hash codes in the hash:

- nbits is the number of bits in the hash code. In LSH, each embedding vector is mapped to a binary hash code. nbits specifies the length of the hash code, i.e., the number of bits in the hash code. larger values of nbits result in a higher resolution of the hash code, which usually leads to more accurate approximation of the search results, as well as an increase in computational and storage overhead.
- bucketSize is the maximum size of each hash bucket. In LSH, embedding vectors are mapped into different hash buckets. bucketSize specifies the maximum number of embedding vectors that can fit in each bucket. An appropriate bucketSize improves retrieval efficiency and reduces the probability of hash conflicts.

Specific steps

First, instance LSH using the parameters game_embedding_map , nbits and bucketSize and add the game embedding to the LSH index. game_embedding_map is the 4.1.2.1 Generating Game and User Embedding Return result.

```
# key code
# Initialize LSH with nbits and bucketSize
self.lsh_instance = faiss.IndexLSH(embedding_len, nbits)
self.lsh_instance.bucketSize = bucketSize
self.lsh_instance.add(np.asarray(game_embeddings, dtype=np.float32))
```

In addition, the IndexLSH instantiation provides a search() method to return the N closest embedding vectors to a given embedding vector. In the following code, the D and I returned by the search() method represent the distance and index of the nearest search_num embedding vectors, respectively:

```
def search(self, embedding, search_num=40):
        D, I = self.lsh_instance.search(np.asarray([embedding], dtype=np.float32), search_num)
```

The return values D and I are explained below:

- D: an array containing the distances of the nearest search_num embedding vectors. Each element represents the distance between the embedding vector of the

corresponding index and the query embedding vector. The smaller the distance, the higher the similarity.

- I: an array containing the indexes of the nearest search_num embedding vectors. Each index points to an embedding vector in lsh_instance. These indexes can be used to find the embedding vector that is most similar to the query embedding vector.

4.1.2.3 Generating game recommendations for users

The process of recommending games needs to be based on user embedding and Local Sensitive Hashing (LSH) techniques. Specifically, the user embedding map has been obtained in Section 4.1.2.1. With this map, i can get the corresponding embedding vector for each user. Next, use the generate_recommendations method and pass in an array of user IDs to be recommended. This method searches each user's embedding vector using the lsh_instance.search method to find similar game embeddings. In the end, the game corresponding to the K most similar game embeddings will be recommended to the user.

```
def lsh_search_for_user(self, user_id, k=20):
        user_embedding = self.users_embedding_map[user_id]
        recommended_ids = self.lsh_instance.search(user_embedding, search_num=k)
        return recommended_ids

def generate_recommendations(self, user_ids):
        return {str(user_id): self.lsh_search_for_user(str(user_id), self.k) for user_id in user_ids}
```

4.1.2.4 Evaluating the generated recommendations

Based on the method in Section 4.1.2.3, the part that generates game recommendations for users, we have obtained a map of game recommendations for each user (user_recommendations_map). Based on this, by calling the evaluate_precision_recall method and passing in the user's actual game records (user_actual_games_map), the user's precision and recall can be calculated.

Specifically, the evaluate_precision_recall method will calculate the precision and recall for each user based on the match between the user's actual game records and the recommended results. Then, the mean of the precision and recall of all users is calculated, i.e., the mean precision and mean recall are obtained. These metrics can be used to evaluate the performance of the recommender system in a group of highly active users to understand the effectiveness and precision of the recommender system.

```
# compute mean precision and recall
def evaluate_precision_recall(user_recommendations_map, user_actual_games_map, k):
```

4.1.3 Experimentation and evaluation of different subsets

In section 4.1.1 Dataset Selection, i introduced the four datasets that will be used for the experiments. This section describes the experimentation process based on these datasets, particularly how they will be used for training and evaluation of the recommender system. The evaluation metrics for this approach are Mean Precision and Mean Recall.

4.1.3.1 Introduction to the Experimental Procedure The experimental procedure will be described in detail.4.1.3.2 Experiments with Different Subset and Parameters: this subsection describes the experiments carried out on different data subsets and parameter settings to evaluate the effect of different subsets and parameters on the recommendation metrics.

4.1.3.1 Introduction to the experimental flow

This section will present the experimental flow step by step, from loading the computed data to viewing the evaluation metrics. Because of the large amount of data, the calculated results will be stored at each step.

Step1.Data Loading and Preparation

First, the previously computed user and game embedding data are loaded from storage. In this example, i use the interaction data of Top 10% highly active users, and generate the corresponding embedding data through 10,000 samples. The specific loading operations are as follows:

```
users_embedding_map_top10 =
load_dill_file(f"/content/drive/MyDrive/9727/saved_data_top10_review/users_embedding_map.dill")
games_embedding_map_top10 =
load_dill_file(f"/content/drive/MyDrive/9727/saved_data_top10_review/games_embedding_map.dill")
```

This data is generated through the steps in Method 4.1.2.1 and contains embedded vectors of users and games that will be used for subsequent recommendation and evaluation.

Step2.Create LSH Instance

Next, the loaded game embedding data is used to create Local Sensitive Hashing (LSH) instances for efficient similarity search. In this example, i construct the LSH search space based on game embeddings of Top 10% highly active users:

```
lsh_instance_top10 = get_lsh_instance(games_embedding_map_top10, nbits=256)
```

Step3. Generate Game Recommendations

Initialise the recommender system using the generated LSH instance and user embedding and generate a recommendation list for the specified users. In this example 200 games are recommended for top10% high activity users.

```
rec_sys_top10 = EmbeddingRecSys(lsh_instance_top10, users_embedding_map_top10, k=200)
user_recommendations_map_top10users = rec_sys_top10.generate_recommendations(evaluate_user_ids)
```

Here the EmbeddingRecSys class uses LSH instances and user embedding to generate the most relevant game recommendations for each user.

Step4. Evaluating Recommendation Results

Finally, i calculate the Mean Precision and Mean Recall of the recommendation results. These metrics are used to evaluate the performance of the recommendation system. In this example, i recommend 200 games for Top 10% highly active users and calculate the corresponding evaluation metrics.

```
mean_precision_top10users, mean_recall_top10users =
evaluate_precision_recall(user_recommendations_map_top10users, user_actual_games_map, k=200)
```

The following figure shows the results of the metrics output by Step4

```
users number: 1472700 games number: 24554
LSH index added 24554 vectors with nbits=256 and bucketSize=50
Average Precision: 0.03141061173195774
Average Recall: 0.6102483522034078
```

4.1.3.2 Experiments with different subsets and parameters

This section describes the experiments conducted with different subsets and various parameter settings. The goal is to assess the impact of these subsets and parameters on the recommendation metrics. Below table summarizes the experimental results for different data subsets and parameter settings, including sampling times, walk depth, LSH parameters, number of recommendations, recommended users , mean precision, and mean recall. Here is an explanation of each table header:

- ID: A unique identifier for each experiment, used to distinguish between different experimental setups.
- Data Subsets: Refers to the specific subset of interaction data used in the experiment. For example, “Full Interaction Data” indicates that the experiment was conducted using the entire dataset of user-game interactions.
- Sampling Times: Indicates the number of game sequences sampled in the DeepWalk process. This value reflects how many times the random walk was performed to generate user and game embeddings.
- Walk Depth: Refers to the depth of each game sequence, i.e., the number of games included in each sampled sequence. This parameter determines how long each game sequence is in the embedding generation process.

- LSH Parameters: Specifies the Locality-Sensitive Hashing (LSH) configuration used for approximate nearest neighbor search. The parameters include nbits, which represents the number of bits used for hashing, and bucketSize, which indicates the size of the hash buckets.
- TopK: Represents the number of recommendations provided to the user by using Embedding based Method. This value shows how many games were selected to be presented to the users.
- User: Indicates the user group targeted in the experiment. For example, “Top 10% Most Active Users” means that the experiment was conducted on the most active 10% of users based on their interaction data.
- Mean Precision: The average precision across all users in the experiment.
- Mean Recall: The average recall across all users in the experiment.

ID	data subsets	sampling times	walk depth	LSH parameters	TopK	User	Mean Precision	Mean Recall
1	Full Interaction Data	10000	40	nbits=256 bucketSize=50	200	Top 10% Most Active Users	0.0006983042137365046	0.01078432809381 4877
2	80% User Interaction Data	10000	40		200		0.01753776756615217	0.38529685224438903
3	Top 10% Most Active Users' Interaction Data	10000	40		200		0.03141061173195774	0.61024835220340 78
		15000	40		200		0.032654905834587133	0.63093862063410 11
4	Top 5% Most Active Users' Interaction Data	20000	50		200	Top 5% Most Active Users	0.03964862579118432	0.53356820493441 27
					100		0.06642765506404251	0.46376512049198 39
					300		0.029148384155819274	0.57508763521525 69

4.1.4 Evaluation Metric Comparison

In this section, I visualise the assessment metrics in Table 4.1.3 and analyse the reasons for them. Specifically, I visualise the three sets of metrics for Experiment IDs 1 to 3 as Figure 4-1 and the three sets of data metrics for Experiment ID 4 as Figure 4-2.

Figure 4-1 Analysis

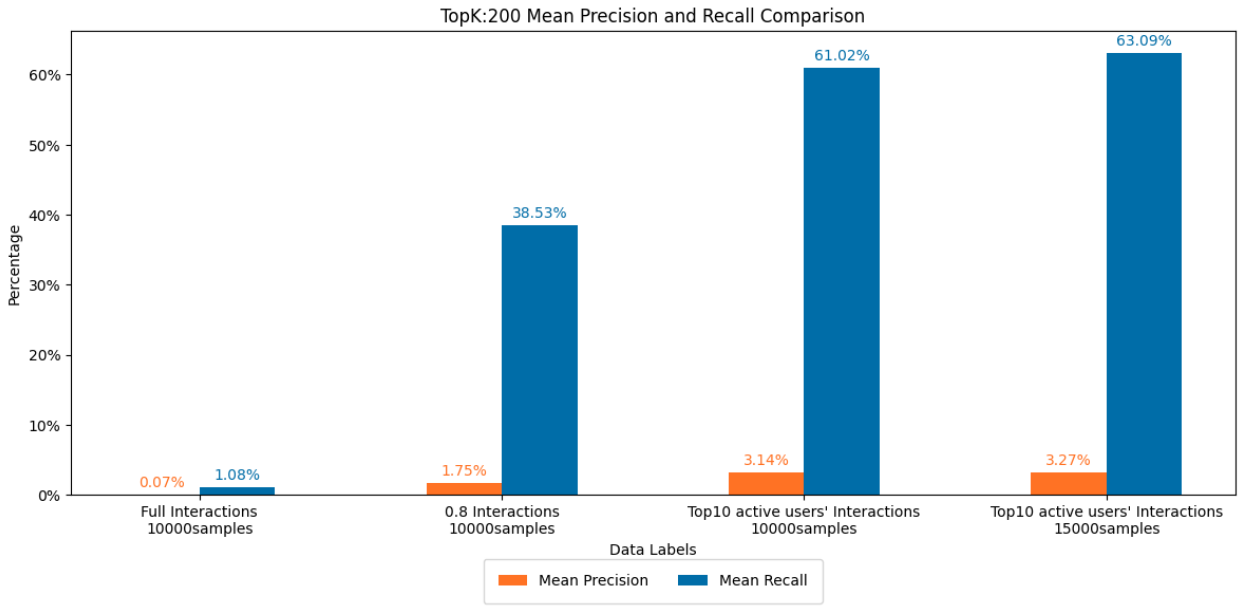

Figure 4-1 Top20

Figure 4-1 visualises the recommendation effect for different subsets of data and number of samples, with the goal of recommending 200 games to the top 10% of highly active users. The following is a detailed analysis of Figure 4-1:

- The first set of histograms in Figure 4-1 shows the results of sampling 10,000 game sequences from the full set of data, and the mean precision is very low at 0.07%. This indicates that the full data contains a lot of noise, which directly affects the accuracy of the recommendations. This noise may come from less interactive or irrelevant user behavioural data, making it difficult for the recommendation method to identify the user's true interests and gaming habits.
- The second set of histograms in Figure 4-1 shows the results of sampling with 80% of user interaction data, where mean precision improves to 1.75% and mean recall improves to 38.53%. This shows that by reducing the noise in the data, the recommendations can be significantly improved. The improvement here suggests that excluding the sparse user interaction data portion helps the word2vec model to capture the sequence of user gaming habits in a higher quality and reduces the interference of less interactive information in the model.
- The third set of histograms in Figure 4-1 shows the results of sampling the interaction data of the top 10% of highly active users 10,000 times and 15,000 times, respectively. In the 10,000 samples, the mean precision increases to 3.14% and the mean recall increases to 61.02%; in the 15,000 samples, the mean

precision further increases to 3.27% and the mean recall increases to 63.09%. These results show that data sampling for highly active users not only improves the data quality, but also further improves the performance of the recommender system by increasing the number of samples. This may be due to the fact that the behavioural data of highly active users are richer and more representative of their real interests, and the model can more accurately learn the preferences of these users through more sampling times.

These results provide an important reference for further optimisation of the recommender system, suggesting that more targeted game sampling and high-quality user interaction data are the key factors in this approach to improve the effectiveness of recommendations. Both the reduction of data noise and the increase in the number of samples can significantly improve the precision and recall of the recommendation system.

Figure 4-2 Analysis

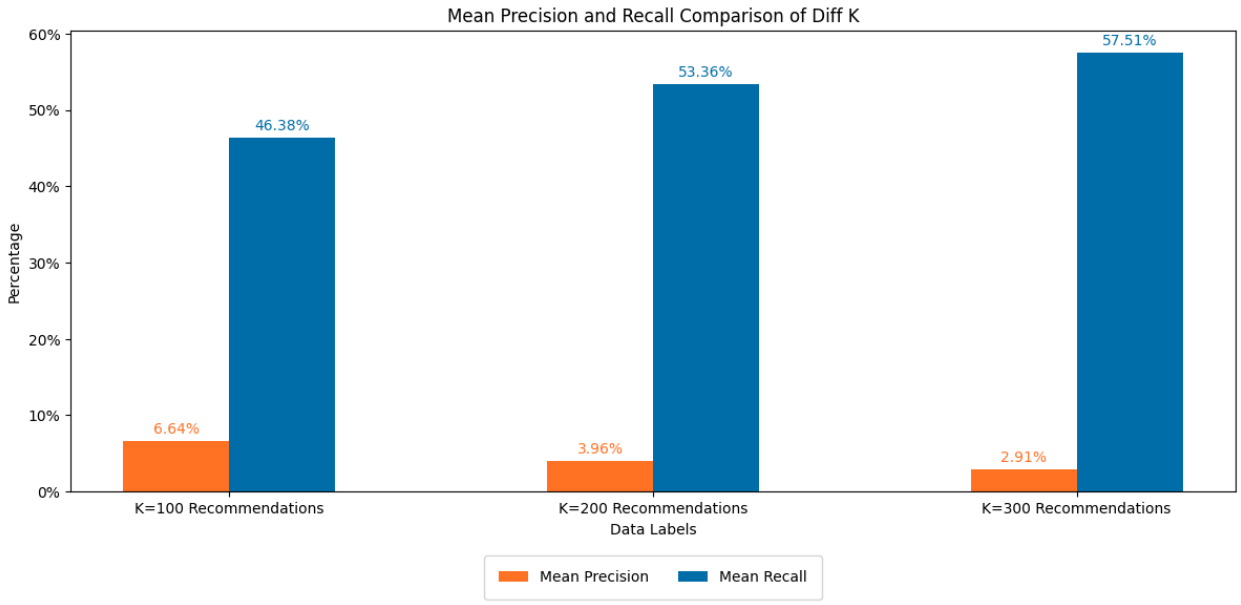

Figure 4-2 Diff K

Figure 4-2 compares the impact of different number of recommendations (K) on the metrics with the same embedded data and the same recommended users (Top 5% active users). From the figure, we can observe that as the number of recommendations increases, the mean precision decreases while the mean recall increases. As the number of recommendations gradually increases, the system recommends more games and the proportion of relevant items included in these games decreases, leading to a decrease in precision, but more relevant items are included in the recommendation list leading to an

increase in recall. This means the recommendation method can capture more items that may be of interest to the user when expanding the recommendation list. Fewer recommendations match the user's interests more accurately, but the coverage is insufficient.

4.2 Deep Ranking Recommendation Model

As mentioned in the system architecture of Part 3, this method belongs to the Ranking Layer of the system module. The main role of this method is to sort the recommendations generated by the Retrieval Layer and filter the games that are more in line with the user's preference to the user, thus improving the metrics such as Mean Precision, Mean Recall, and MAP.This part will focus on the feature engineering, data selection, model training, and evaluation of the MLP Rank Model method.

4.2.1 Feature Engineering

In this section it will be described how the project prepared the feature data needed for model training and prediction.

4.2.1.1 Model training features

Complete game information table

In Part2.1 Datasets Introduce, game.csv and game_matadata.json were introduced, and since tag is an important feature of the game, these two files need to be merged to obtain the complete game information. Figure below shows the merged complete game datasheet, which provides the basis for the subsequent feature engineering and model training. Figure 4 shows the merged game data table, which provides the basis for subsequent feature engineering and model training.

app_id	title	app_release_date	win	mac	linux	rating	positive_ratio	user_reviews	price_final	price_original	discount	steam_deck	description	tags	app_release_ts	price_diff	numeric_rating
1240050	Pendragon	2020-09-22	1	1	0	Mixed	68	94	16.99	16.99	0.0	1	AD 673. Camelot h...	[Narrative, Strat...	1600732800	0.0	5
1263200	Smashing Spirits:...	2020-09-10	1	1	1	Positive	84	33	3.99	3.99	0.0	1	Fight Spartans at...	[Side Scroller, P...	1599696000	0.0	7

User Interaction Table

Next, i need to process the user interaction data. Figure below shows the processed interaction data with two new fields: label and review_ts. review_ts is generated by converting the original review_date into a timestamp form; The label is a digitization of the original is_recommended field, which will be used as the real value for training the model.

is_recommended	user_id	app_id	review_date	hours	review_ts	label
false	6473513	213610	2021-09-30	3.0	1632960000	0
true	2185259	213610	2021-03-29	81.5	1616976000	1

User game preference features

Next, I need to merge the user interaction table with the full game information table. Since i have an app_id field in the user interaction table, i can merge it with the game information table, resulting in the following data:

app_id	is_recommended	user_id	review_date	hours	review_ts	label	app_release_ts	win	mac	linux	numeric_rating	positive_ratio	user_reviews	price_diff	discount	steam_deck	tags
1499540	false	11695280	2021-08-26	0.1	1629936000	0	1635811200	1	0	0	5	63	19	0.0	0.0	1	[eSports, Real Ti...
1499540	true	9283098	2021-11-04	6.6	1635984000	1	1635811200	1	0	0	5	63	19	0.0	0.0	1	[eSports, Real Ti...
1499540	false	13086459	2022-04-03	0.1	1648944000	0	1635811200	1	0	0	5	63	19	0.0	0.0	1	[eSports, Real Ti...
1499540	false	9981325	2021-08-26	0.2	1629936000	0	1635811200	1	0	0	5	63	19	0.0	0.0	1	[eSports, Real Ti...
1499540	true	11553593	2021-11-11	2.5	1636588800	1	1635811200	1	0	0	5	63	19	0.0	0.0	1	[eSports, Real Ti...

By merging these two datasets, we can obtain a comprehensive view of user preference features and the features of the games they favor. However, these features require further processing. Specifically, the following features need to be refined:

- user_hours_mean: The average game time for each user provides insights into their gaming patterns, which can be utilized to deliver more personalized recommendations.
- user_hours_std: The standard deviation of a user's game time reveals the variability and consistency in their gaming duration, serving as a key indicator of user preferences.
- user_liked_release_ts_mean: The average release dates of games preferred by users can indicate whether they favor newer or older titles.
- user_liked_tags_top50: Identifying each user's top 50 favorite tags enables more precise recommendations that align closely with their specific interests.

```
liked_interactions_df = user_item_interactions_df \
.withColumn('user_hours_mean', F.round(F.mean(F.col('hours')).over(windowSpec), 2)) \
.withColumn('user_hours_std', F.round(F.stddev(F.col('hours')).over(windowSpec), 2)) \
.withColumn('user_liked_release_ts_mean',
F.round(F.mean(user_recommended_filter('app_release_ts')).over(windowSpec),0))\
.withColumn("top50_liked_tags",topN_liked_tags('collected_tags', F.lit(50)))
```

Processed Features

After obtaining the user game preference feature table, further processing is required to enhance the model training's effectiveness. Specifically, the raw features must undergo Min-Max normalization and Multi-Hot encoding. Figure 4-3 illustrates the processed user game preference feature table. The following steps are essential before proceeding with model training:

- Min-Max Normalization: The user game preference features are scaled to the [0, 1] range through Min-Max normalization. This step helps eliminate differences in

feature value ranges, thereby improving the model's convergence speed and prediction accuracy.

- Multi-Hot Encoding: The tag features in the table are encoded using Multi-Hot encoding, which allows the model to effectively utilize discrete label features.

```
# Normalize game feature values using Min-Max scaling.
item_features2minmax = [
'numeric_rating',
'positive_ratio',
'user_reviews',
'price_diff',
'discount',
'app_release_ts',
]
user_features2minmax = [
'user_review_count',
'user_hours_mean',
'user_hours_std',
'user_liked_release_ts_mean',
]
user_minmax_scaler = NumMinMaxScaler(user_features2minmax)
item_minmax_scaler = NumMinMaxScaler(item_features2minmax)
item_minmax_scaler.fit(liked_interactions_df)
user_minmax_scaler.fit(liked_interactions_df)
liked_interactions_df = item_minmax_scaler.transform(liked_interactions_df)
liked_interactions_df = user_minmax_scaler.transform(liked_interactions_df)

item_tags_encoder = TagEncoder('tags')
item_tags_encoder.fit(liked_interactions_df, spark)
liked_interactions_df = item_tags_encoder.transform(liked_interactions_df)
liked_tags_encoder = TagEncoder('top50_liked_tags')
liked_tags_encoder.fit(liked_interactions_df, spark)
liked_interactions_df = liked_tags_encoder.transform(liked_interactions_df)
```

app_id	is_recommended	user_id	review_date	hours	review_ts	label	app_release_ts	win	mac	linux	numeric_rating	positive_ratio	user_reviews	price_diff	discount	steam_deck
220200	true	177	2020-01-09	388.9	1578528000	1	1430092800	1	1	1	9	95	94712	-10.0	0.0	1
1938090	true	301	2022-11-03	59.7	1667433600	1	1666828800	1	0	0	5	59	429206	0.0	0.0	1
1091500	true	301	2022-12-19	49.1	1671408000	1	1607472000	1	0	0	8	80	557051	-60.0	0.0	1
230410	false	411	2020-03-05	1.2	1583366400	0	1364169600	1	0	0	8	86	542198	0.0	0.0	1
626680	true	548	2017-06-30	28.5	1498780800	1	1497484800	1	0	0	8	81	942	0.0	0.0	1

user_hours_mean	user_hours_std	user_liked_release_ts_mean	collected_tags	top50_liked_tags	numeric_rating_scaled	positive_ratio_scaled	user_reviews_scaled	price_diff_scaled	discount_scaled	app_release_ts_scaled
388.9	NULL	1.4300928E9	[[]]	[]	1.0	0.95	0.01	0.35	0.0	0.68
59.7	NULL	1.6668288E9	[[]]	[]	0.5	0.59	0.06	0.4	0.0	0.97
54.4	7.5	1.6371504E9	[[], []]	[]	0.88	0.8	0.07	0.1	0.0	0.9
1.2	NULL	NULL	[]	[]	0.88	0.86	0.07	0.4	0.0	0.6
28.5	NULL	1.4974848E9	[[Free to Play, R...	[Free to Play, Ra...	0.88	0.81	0.0	0.4	0.0	0.76

user_review_count_scaled	user_hours_mean_scaled	user_hours_std_scaled	user_liked_release_ts_mean_scaled	tags_multihot	top50_liked_tags_multihot
0.0	0.39	NaN	0.68	[0, 0, 0, 0, 0, 0...	[0, 0, 0, 0, 0, 0...
0.0	0.06	NaN	0.97	[0, 0, 0, 0, 0, 0...	[0, 0, 0, 0, 0, 0...
0.0	0.05	0.01	0.93	[0, 0, 0, 0, 0, 0...	[0, 0, 0, 0, 0, 0...
0.0	0.0	NaN	NaN	[0, 0, 0, 0, 0, 0...	[0, 0, 0, 0, 0, 0...
0.0	0.03	NaN	0.76	[1, 1, 0, 1, 1, 0...	[1, 1, 0, 1, 1, 0...

Figure 4-3

After completing the above processing steps, I have obtained the comprehensive user preference features for all users. Each record in the table now includes both user-specific

features and the features of the games they interact with. Specifically, these features include:

User Features:

- user_hours_mean_scaled: The Min-Max normalized result of the user's average game time.
- user_hours_std_scaled: The Min-Max normalized result of the standard deviation of the user's game time.
- user_liked_release_ts_mean_scaled: The Min-Max normalized result of the average release date of games that the user prefers.
- top50_liked_tags_multihot: The Multi-Hot encoded result of the user's top 50 favorite tags.

User-Game Interaction Features:

- win: Whether the game supports the numerical features of the Windows platform, where 1 means yes and 0 means no.
- mac: Whether the game supports the numeric features of the Mac platform, where 1 means yes and 0 means no.
- linux: Whether the game supports the numeric features of the Linux platform, where 1 means yes and 0 means no.
- steam_deck: Whether the game supports the numerical features of the Steam Deck (1 if yes, 0 if no)
- numeric_rating_scaled: The Min-Max normalized result of the game rating
- positive_ratio_scaled: This is the Min-Max normalization of the percentage of positive ratings for the game.
- user_reviews_scaled: The Min-Max normalized result of the number of comments the user has made.
- price_diff_scaled: Min-Max normalized result of game price differences.
- discount_scaled: The Min-Max normalized result of the game discount.
- app_release_ts_scaled: The Min-Max normalized result of the game's release date
- tags_multihot: Multi-Hot encoding result of game tags.

Through these preprocessing steps have ensured that all features are numerically scaled, standardized, and encoded using Multi-Hot encoding, making them suitable for model training.

4.2.1.2 Features for Model Prediction

In this section, i will extract the latest features and user interaction game features belonging to the Top 5% highly active users from the processed feature data. The newest feature of a user means that only the most recent interaction records of each user are kept in chronological order. This is because we have all the data about users and interactive games in our processed feature data, but model training only needs the most recent features for each user for our model prediction.

Latest user features

- The features of the Top 5% highly active users are filtered from the complete user feature data.
- Only the most recent features of each user are kept by sorting by timestamp (review_ts).

```
latest_top5_user_features_df = full_user_features_df.join(
            broadcast_unique_user_ids,
            on="user_id",
            how="inner").select(
            "user_id",
            "review_ts",
            'user_hours_mean_scaled',
            'user_hours_std_scaled',
            'user_liked_release_ts_mean_scaled',
            'top50_liked_tags_multihot')


# Create temporary views and use SQL queries.
latest_top5_user_features_df.createOrReplaceTempView("user_features")


# Use SQL queries to select the most recent record for each user.
query = """
            SELECT * FROM (
            SELECT user_id, review_ts, user_hours_mean_scaled, user_hours_std_scaled,
            user_liked_release_ts_mean_scaled, top50_liked_tags_multihot,
            ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY review_ts DESC) as row_num
            FROM user_features
            ) tmp
            WHERE row_num = 1
"""
latest_top5_user_features_df = spark.sql(query)
```

Game Features

In this section, the game records of the Top 5% highly active users are extracted. I need to filter out the corresponding app_id records in Process Features and remove duplicates.

```
unique_app_active_review_df = spark_high_5_active_review_df.select("app_id").distinct()
```

```
item_features = [
'app_id', 'win', 'mac', 'linux', 'steam_deck', 'numeric_rating_scaled', 'positive_ratio_scaled',
'user_reviews_scaled', 'price_diff_scaled', 'discount_scaled', 'app_release_ts_scaled', 'tags_multihot']

top5liked_game_feature_df = unique_app_active_review_df.join(
full_user_features_df, on="app_id", how="inner").select(item_features)

top5liked_game_feature_df = top5liked_game_feature_df.dropDuplicates()
```

4.2.2 Datasets Selection

In this section, i will introduce two datasets that consist of features (X) and target variables (y) for model training. the two datasets are Processed Features from the Top 5% Most Active Users and 80% Subset of Processed Features from the Top 5% Most Active Users. In the following, I will go into more detail about both the content of the data and its role in model training.

Processed Features from the Top 5% Most Active Users

- Data introduction: In the step of 4.2.1.1 model training features, i processed the game preference features of the full number of users. The features of the Top 5% highly active users are extracted and used for the subsequent training of the offline model. The label field is used as the target variable (y) for model training.
- Reason: Since this ranking model is an offline model, it needs to be applied to online prediction in the future, so it must be trained with the full data of these 5% high active users. By training the model to learn the features and preferences of these users, we can lay a solid foundation for subsequent prediction and ranking tasks, ensuring that the model accurately reflects the interests and needs of highly active users.

Holdout-Split 80% Subset of Processed Features from the Top 5% Most Active Users

- Data introduction: Based on the extracted Top 5% highly active user features, holdout method is used to split the data into 80% training data. In this way, we provide the model with more representative training data, which improves its generalization ability and predictive performance.
- Reason: By using this 80% subset, it is possible to tune the parameters during model training and test the performance of the model. This process helps us understand the performance of the model and optimize parameters to improve its accuracy and stability in preparation for production deployment.

By selecting and processing these datasets, the system can ensure that the model is fully learned during the training process and has good generalization ability to cope with future online prediction challenges.

4.2.3 MLP Rank Model

In this section, the structure of MLP Rank Model will be introduced in detail, including the training process of the model and the basic model evaluation method.

Model structure

This section describes the structure of the Multi-layer Perceptron (MLP) ranking model. Figure 4-4 is a diagram of the model generated with the plot_model method.

Input layer

- Input 1 (feat_tags_multihot): Multi-Hot encoded features for game tags. This feature captures the category and attribute information of games and helps the model to understand the similarity between games.
- Input 2 (feat_liked_tags_multihot): Multi-Hot encoded features of the tags that the user likes. This feature reflects the user's preference and helps the model to make personalized recommendations according to the user's interests.
- Input 3 (feat_item): Features of the game: win, mac, linux, steam_deck, numeric_rating_scaled, positive_ratio_scaled, user_reviews_scaled, price_diff_scaled, discount_scaled, app_release_ts_scaled.
- Input 4 (feat_user): User features: user_hours_mean_scaled, user_hours_std_scaled, and user_liked_release_ts_mean_scaled.

Compression Layers

- compact_x1: Compresses the game tag features through a fully connected layer with 40 units, using the ReLU activation function.
- compact_x2: Compresses the user’s favorite tag features through a fully connected layer with 40 units, using the ReLU activation function.

Merge Layer

- merge: Combines the compressed game tag features, user’s favorite tag features, other game features, and other user features into a single layer.

Hidden Layers

- h1: The first hidden layer, consisting of 256 neurons, with the ReLU activation function.

- h2: The second hidden layer, consisting of 128 neurons, with the ReLU activation function.
- h3: The third hidden layer, consisting of 64 neurons, with the ReLU activation function.

Output Layer

- output: The output layer, consisting of 1 neuron, with the Sigmoid activation function, used for binary classification tasks.

Model compilation parameters

- Optimizer: Adam with 0.0001 learning rate.
- Loss function: binary_crossentropy
- Evaluation metric: accuracy

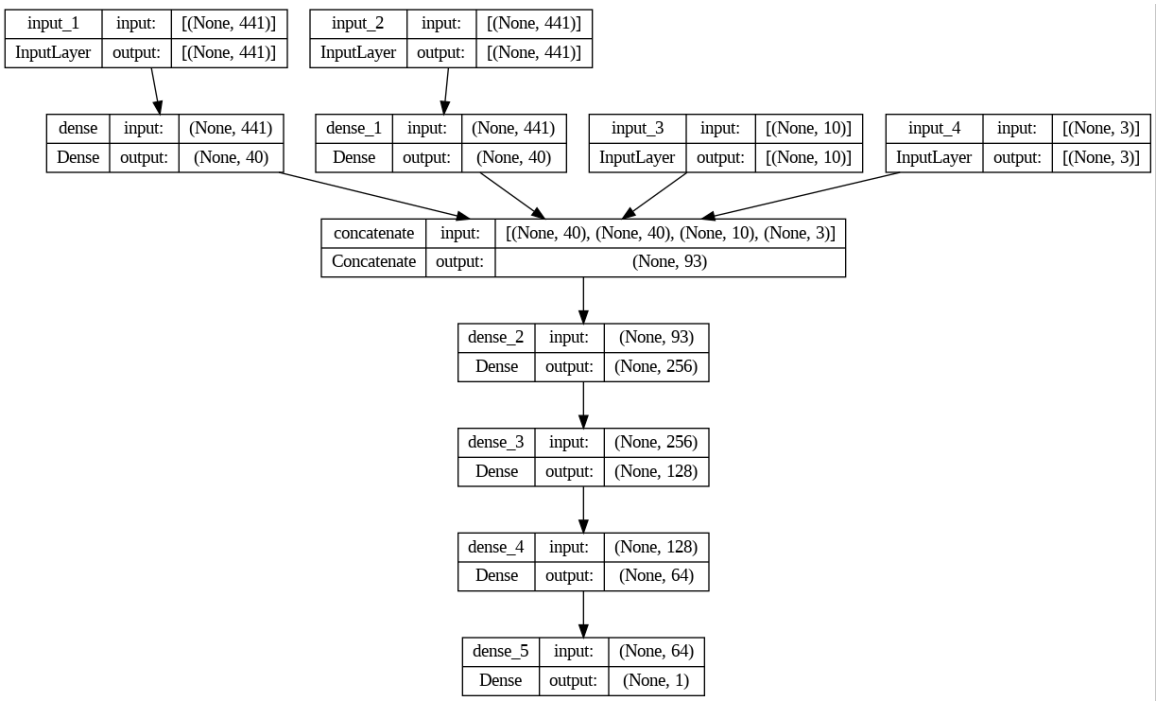

Figure 4-4

Model Training

In this section, I will train 2 Rank models using the two subsets mentioned in the 4.2.2 Datasets Selection.

Rank Model name	Subset(see 4.2.2)
mlp_full_model	Processed Features from the Top 5% Most Active Users
mlp_train_test_split_model	Holdout-Split 80% Subset of Processed Features from the Top 5% Most Active Users

Model Evaluation and Analysis

In this section, I use the remaining 20% data in the Holdout-Split 80% Subset of Processed Features from the Top 5% Most Active Users in the dataset selection in 4.2.2 for testing. To evaluate the performance of the model.

```
test_loss, test_accuracy = mlp_model.evaluate(test_feats, test_labels)
```

```
98118/98118 [==============================] - 136s 1ms/step - loss: 0.4426 - accuracy: 0.8380
Test Loss: 0.442577987909317
Test Accuracy: 0.8379607200622559
```

The test results show that the model trained by Holdout-Split 80% Subset of Processed Features has high accuracy and can effectively judge the target variable y (label). Despite the high accuracy of the model, from the loss value, the model still has some errors on the test data, which indicates that there is a certain gap between the prediction results of the model and the true labels. In order to further improve the performance of the model, I plan to optimize it in the following ways:

- Grid Search: Find the best combination of parameters for a model by tuning the hyperparameters.
- Feature engineering: The existing features are appropriately adjusted or new features are introduced to enhance the expressive power of the model.
- Replay validation: i introduced the Replay validation method to verify the model's performance through the time series, to improve its effect in practical applications.

4.2.4 Experiments

In this section, the setup of the controlled experiment and the relevant parameter instructions will be detailed. The goal of these experiments is to evaluate the performance of different ranking models after ranking the recommended items from the Retrieval Layer. It should be noted that all the recommended items are generated based on the Top 5% highly active users, both in the Retrieval Layer and the ranking layer.

The Retrieval Layer uses the embedding-based recommendation method to generate the initial recommendation list, while the task of the ranking layer is to optimally rank these recommended items based on this to improve the recommendation quality and relevance. The following table lists the main parameters of the experiment and their results:

id	User Group	Rank Model Name	Recommendations (from 4.1)	Ranked Recommendations	Mean precision	Mean Recall	MAP

1	Top 5% Most Active Users	mlp_train_test_split_model	300	250	0.032799999999999996	0.55172957477305331	0.062035239041017175
2		mlp_full_model	300	250	0.0316	0.533218665392578 5	0.049477951643377 76
3		mlp_train_test_split_model	200	150	0.048666666666666667	0.484378085682433 46	0.070804408581748 7
4		mlp_full_model	200	150	0.049999999999999996	0.506527047831395 7	0.076163385598175 73

- ID: A unique identifier for each experiment or test, used to distinguish between different experimental setups.
- User Group: Represents the category of user grouping. In these experiments, all recommendations are based on the Top 5% most active users.
- Rank Model Name: The name of the ranking model (see 4.2.3 MLP Rank Model training).
- Recommendations (Retrieval): The number of game recommendations retrieved from the Retrieval Layer. These recommendations are generated using the 4.1 embedding-based recommendation method and serve as the input for the ranking model to predict user preference probabilities.
- Ranked Recommendations: The number of recommendations after being ranked by the Rank Model. This refers to the number of recommendations presented to the user after the recommendation list has been sorted based on the model's predicted interest levels.
- Mean Precision: The average precision across the recommended lists for the user group, indicating the proportion of relevant items in the recommendation list.
- Mean Recall: The average recall across the recommended lists for the user group, indicating the proportion of all relevant items that were successfully recommended.
- MAP (Mean Average Precision): This metric combines precision and ranking performance, assessing the overall effectiveness of the recommendation system across multiple queries. It is calculated by averaging the precision scores of relevant items in the recommendation list across all users.

4.2.5 Evaluation Matrics Comparison

In this section, I will visualize the key metrics from the experiment results and provide an in-depth analysis of the reasons behind the observed changes.

Comparison Case 1

In Figure 4-5, two bar charts illustrate the recommendation performance of the Retrieval Layer (using the embedding-based recommendation method, see Section 4.1) and the ranking layer (using the Deep Ranking Model method, see Section 4.2).

- Label: Recall 300 Recommendations
  This bar chart shows the performance metrics when recommending 300 games to the Top 5% most active users during the initial recall stage. At this stage, the recommendation system generates a large number of potential recommendations without further sorting or optimization.
- Label: Rank[:250]
  This bar chart presents the performance metrics after the top 250 recommendations from the initial 300 have been re-ranked using the Rank Model (Experiment ID: 1). Through the optimization provided by the Rank Model, the final 250 recommendations align more closely with the users' interests and preferences.

By comparing these two sets of data, it is evident that Mean Precision improves after applying the Rank Model. This result confirms the effectiveness of the Rank Model in enhancing the accuracy of recommendation results.

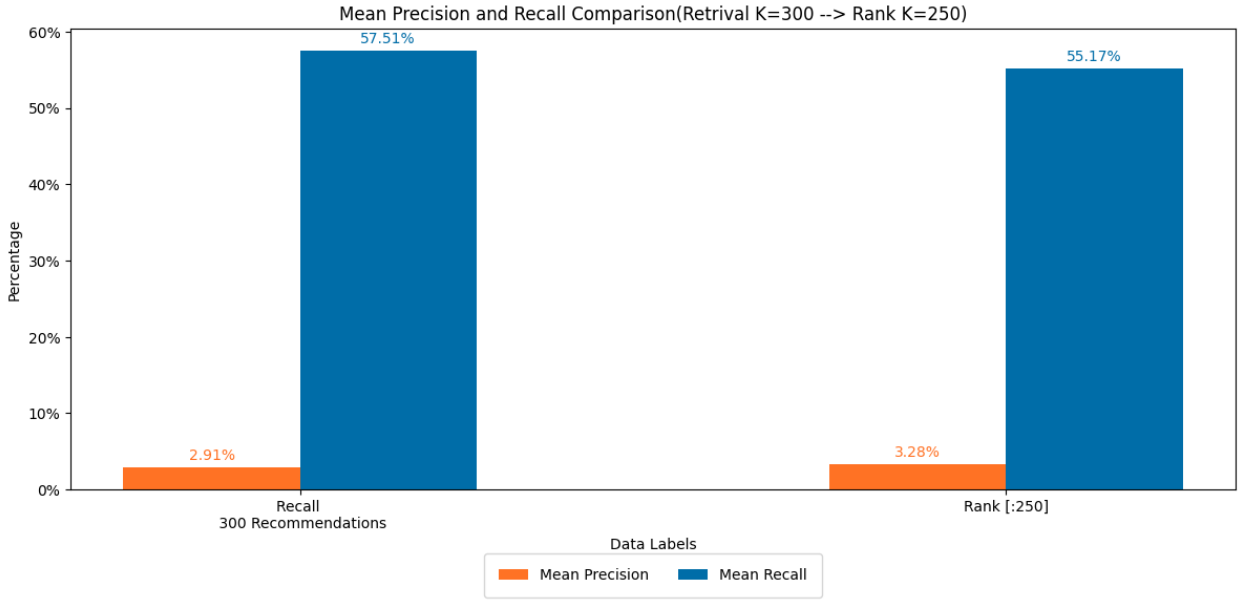

Figure 4-5

Comparison Case 2

Case 2 was experimented in a similar way to Case 1, with the only difference being the amount of data for the recommended items. In Figure 4-6, i evaluate the performance of the ranking model with a smaller number of recommendations.

- Label: Recall 200 Recommendations
  This bar chart shows the performance of the metrics when recommending 200 games for the Top 5% of highly active users in the Retrieval Layer.
- Label: Rank 150 Recommendations
  This bar shows the performance of the top 150 items ranked by Rank Model (experiment ID: 3).

Although the amount of data of the recommended items is reduced, the Rank Model can still improve the Mean Precision, which further verifies the role of the Rank Model in improving the recommendation quality in the recommendation system.

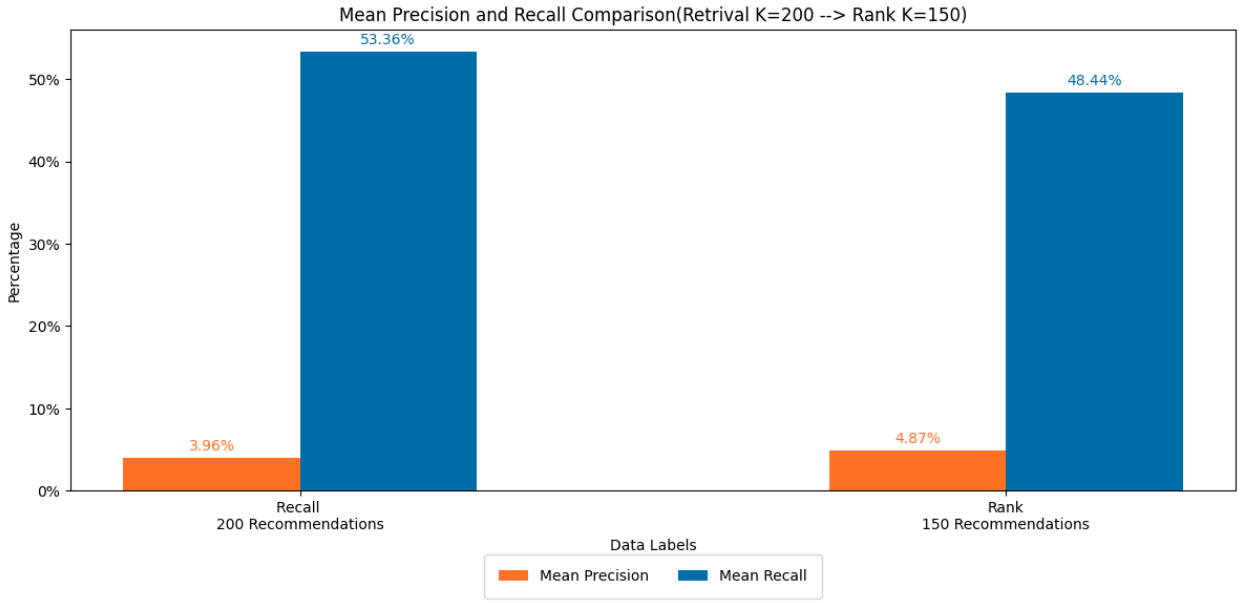

Figure 4-6

Comprehensive analysis of Case1 and 2

Through the analysis of Case 1 and Case 2, we can see that the Rank Model can significantly improve the Mean Precision under different amounts of data, which indicates that the ranking model plays an important role in optimizing the relevance and accuracy of the recommendation results. However, along with the improvement of Mean Precision, we also observe a decrease in the Mean Recall metric. This phenomenon reflects the trade-off between Precision and Recall.

In general, ranking models prefer to optimize Precision, that is, ranking the most relevant items at the top of the recommendation list. Although this optimization strategy effectively improves the precision of the recommendation list, the Recall will decrease because some relevant items are excluded from the final recommendation list. In other words, although it improves the relevance of the recommended items, it may not be able to fully cover all the interest points of the user. To further optimize model performance, I plan to improve this trade-off in future work by:

- Further optimizing the offline model: The Mean Precision can be further improved by adjusting the model parameters or introducing more representative features.
- Hybrid strategy: adding a small number of low ranking relevant items to the ranked recommendation list to improve the Mean Recall. This strategy can maintain or improve Recall without significantly reducing Precision, so as to ensure the comprehensiveness of the recommendation results while maintaining high precision.

This balance strategy can provide a more comprehensive user experience for the recommender system, which can not only ensure the accuracy of the recommendation, but also cover more potential interests of users.

Comparison Case 3

In Figures 4-7, the first and second set of bars show the MAP (Mean Average Precision) comparison of 150 and 250 ranked recommendations, respectively. The graph shows that the MAP decreases significantly as the number of recommendations increases, regardless of whether the model trained on the 0.8 dataset or the model trained on the full dataset is used. This phenomenon indicates that with the increase of recommended items, the model may introduce some items with weak correlation with user interests when ranking. These low correlation items occupy a part of the recommendation list, resulting in the decrease of overall correlation and then the decrease of MAP.

This situation mainly reflects the limited generalization ability of the model, which makes it difficult to effectively distinguish between high and low relevance items. In addition, the ranking model itself has some limitations. Ranking models usually prioritize optimizing Precision, that is, placing items with the highest predictive relevance at the top of the recommendation list. But when the number of recommendations increases, the ranking task becomes more complex, and the model may not be able to maintain the same ranking quality, resulting in a decrease in MAP.

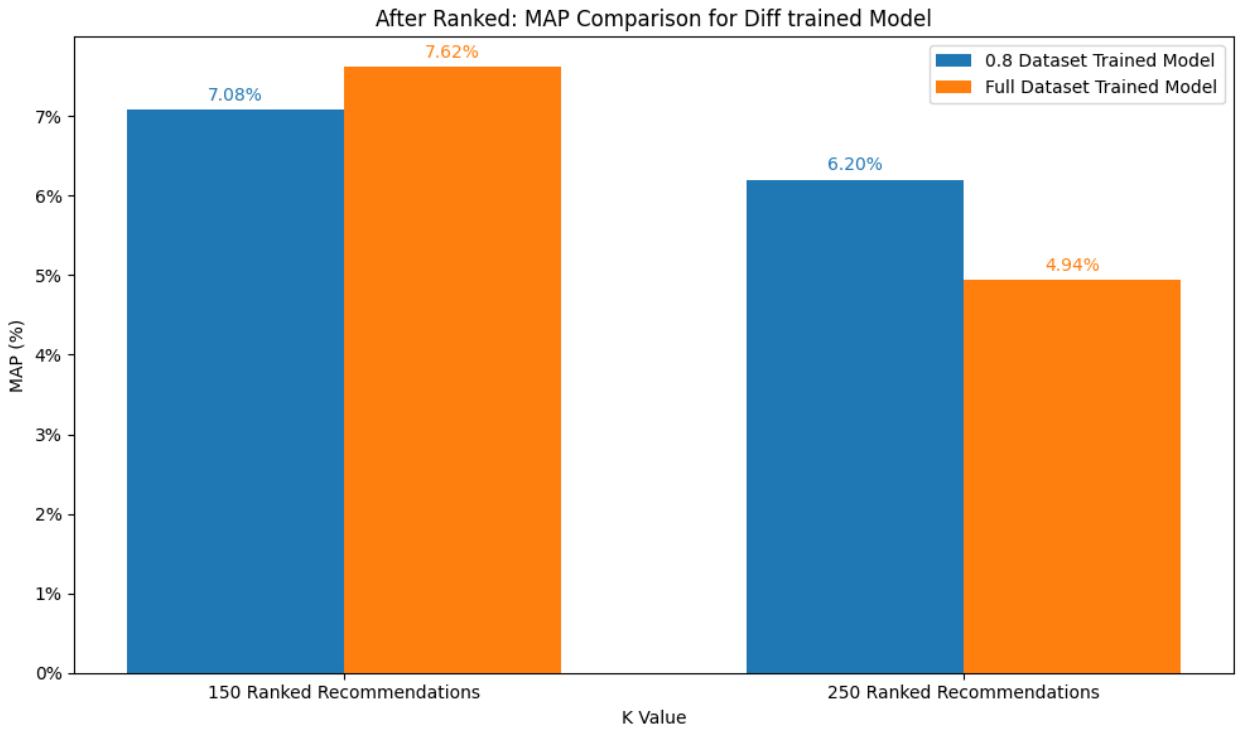

Figure 4-7

Part 5 Reflection

In this project, I successfully built an ideal recommendation system, covering the complete process from the Retrieval Layer to the ranking layer, to provide personalized recommendations for users. During the evaluation phase, I performed a detailed comparison of metrics across different subsets of the data, which effectively revealed directions that could be optimized later.

However, there are some challenges in the project, such as high computational resource requirements and bottlenecks in big data processing. Although I used Spark and multi-process computing to speed up the data processing, it still cost a lot of time. In view of these problems, I will consider the following improvement measures in future work:

- Data subset optimization: Narrowing the data subset to 100 highly active users for testing will help reduce the computational resource consumption and improve the efficiency of the experiment.

- Experiment with parameters and methods: On a smaller dataset, I can have more time to tune parameters and test different distance/similarity calculation methods, such as K-means, etc., to optimize the performance of the recommender system.
- Computational resource management: Further explore more efficient data processing methods and techniques, such as using distributed computing frameworks or improving data preprocessing steps, to reduce computational bottlenecks and time costs.

In the direction of future improvement, I think Sequential Recommendation can be an important extension of the recommendation system I developed. Sequential recommendation can capture the temporal dynamic features of user behavior, and predict the future points of interest by analyzing the user's historical interaction sequence. This approach is particularly suitable for game recommendation, as a user's game preferences are often influenced by their previous game experience. The introduction of sequential recommendation can improve the accuracy and relevance of recommendation, and make the system more intelligent to predict the user's next choice. In addition, to further optimize model performance, I plan to introduce an offline model validation method called Replay. By using time series data for verification, Replay ensures that future information is not introduced in model training and verification, to be closer to the actual application scenario. This verification method can not only help to identify the performance of the model at different time nodes, but also can repeatedly adjust the parameter Settings and structure of the model in the offline environment, and finally improve the prediction accuracy and recommendation effect of the model. Combining the Replay method and the sequential recommendation strategy, I believe that the precision and user satisfaction of the recommender system can be significantly enhanced.

Reference

1. Shi, J., Chaurasiya, V., Liu, Y., Vij, S., Wu, Y., Kanduri, S., ... & Yu, J. (2023, July). Embedding based retrieval in friend recommendation. In Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval (pp. 3330-3334).
2. Exploring graph embeddings: DeepWalk and Node2Vec. (n.d.). Towards Data Science. Retrieved from https://towardsdatascience.com/exploring-graph-embeddings-deepwalk-and-node2vec-ee12c4c0d26d

3. Multi-layer perceptron (MLP) models on real-world banking data. (n.d.). Becoming Human. Retrieved from https://becominghuman.ai/multi-layer-perceptron-mlp-models-on-real-world-banking-data-f6dd3d7e998f
4. Recommendation systems: Ranking/Scoring. (n.d.). Aman.ai. Retrieved from https://aman.ai/recsys/ranking/#:~:text=The%20items%20with%20the%20highest,presented%20to%20the%20user%20first
5. A dummy's guide to Word2Vec. (n.d.). Medium. Retrieved from https://medium.com/@manansuri/a-dummys-guide-to-word2vec-456444f3c673
6. Evaluating recommender systems: Behavioral metrics. (n.d.). Evidently AI. Retrieved from https://www.evidentlyai.com/ranking-metrics/evaluating-recommender-systems#behavioral-metrics
7. Wang, S. (n.d.). Recommender system [GitHub repository]. Retrieved from https://github.com/wangshusen/RecommenderSystem/tree/main
8. Offline evaluation. (n.d.). Geek Time. Retrieved from https://time.geekbang.org/column/article/317319