Dean&Francis

ISSN 2959-6157

Implementation and optimization of recommendation systems

Jinghan Liang

Nankai University, Jinnan Campus, Nankai University, No.38 Tongyan Road, Haihe Education Park, Jinnan, Tianjin, China

Abstract:

This study focuses on the implementation and optimization of a recommendation system using deep learning-based collaborative filtering algorithms. The system utilizes user-item interactions from provided datasets to predict user ratings for items in a test set. We introduce a hybrid model that incorporates both collaborative filtering and matrix factorization techniques to enhance prediction accuracy. The collaborative filtering approach exploits similarities between user preferences, while the matrix factorization method decomposes the user-item matrix to capture latent features. The effectiveness of the system is evaluated through various metrics, including precision and recall, with results indicating substantial improvements in recommendation accuracy and system robustness.

Keywords: recommendation systems, collaborative filtering, matrix factorization, deep learning, user-item interactions, prediction accuracy, system optimization

Introduction

In the realm of digital services, personalized recommendation systems have become a pivotal component in enhancing user experience and satisfaction. These systems not only facilitate user engagement by personalizing user interactions but also drive increased revenue for service providers by recommending relevant products or services. Traditional recommendation methods have predominantly relied on collaborative filtering (CF) techniques, which recommend items based on the similarity of users' past behaviors. However, with the exponential growth of available data and the increasing complexity of user preferences, traditional CF techniques often suffer from issues such as data sparsity and scalability. Recent advancements in deep learning have provided new opportunities to tackle these challenges through more sophisticated models. This paper explores the implementation and optimization of a recommendation system that integrates deep learning with collaborative filtering. We aim to enhance the predictive accuracy and efficiency of the system by employing matrix factorization techniques to effectively handle large-scale data and complex user-item interactions. The objective is to deliver a more accurate, scalable, and robust recommendation system that can adapt to varying user preferences and behaviors.

Dataset introduction and related processing

1. Dataset:

(1) Train.txt (user rating records), used for model

training.
(2) Test.txt (user sequences that have not been rated and item sequences that these users need to rate), used for testing.
(3) ItemAttribute.txt (for similarity calculation and collaborative filtering), used for model training.
(4) ResultForm.txt, which is the format of the result file. The format of the dataset is explained in DataFormatExplanation.txt.
2. Data processing
(1) load_training_data:
The load_training_data function is used to load data from a given training data file and divide it into training data and test data. The function randomly determines whether a rating record is put into the training dataset or the test dataset by setting a split ratio (split_ratio).
The details are as follows:
Open the training data file and read the file content line by line.
a) If the current line contains the character |, it indicates that this is a new user information line:
- Using map(int, line.strip().split('|')) get the User ID.
- Initialize an empty dictionary in training_data and testing_data for the user.

b) If the current line does not contain the character |, it means this is a rating record for the user:
- Use map(int, line.strip().split()) to extract item IDs and scores.
- Use map(int, line.strip().split()) to extract item IDs and scores.
- Use random.random() to generate a random number between 0 and 1. If the random number is less than split_ratio, add the record to testing_data, otherwise add it to training_data

```
def load_training_data(train_file_path, split_ratio):
    # Set a random seed to ensure reproducibility
    random.seed(36)

    # Initialize empty dictionaries for training and testing data
    training_data = {}
    testing_data = {}

    # Open the training data file
    with open(train_file_path, 'r') as file:
        for line in file:
            # If the line contains ' |', it indicates a new user
            if ' |' in line:
                user, _ = map(int, line.strip().split(' |'))  # Get the user ID
                # Initialize dictionaries for this user in training and testing
data
                training_data[user] = {}
                testing_data[user] = {}
            else:
                # Otherwise, it represents a rating record for the user
                item, rating = map(int, line.strip().split())  # Get the item ID
and rating
                # Randomly assign the record to testing or training data based on
split_ratio
                if random.random() < split_ratio:
                    testing_data[user][item] = rating
                else:
                    training_data[user][item] = rating

    # Return the training and testing data
    return training_data, testing_data
```

ISSN 2959-6157

(2) load_testing_data:
This function is used to load data from a given test data file and return a list of user and item pairs. Each pair (user, item) indicates that the user's rating for the item needs to be predicted.
The details are as follows:
Open the test data file and read the file content line by line.
a) If the current line contains the character |, it indicates that this is a new user information line:
- Use map(int, line.strip().split('|')) to extract the user ID and store it in the variable user.

b) If the current line does not contain the character |, it means this is an item to be predicted for the user:
- Use int(line.strip()) to extract Item ID.
- Add (user, item) pairs to the test_cases list.

```
    def
load_testing_data(test_file_
path): test_cases = []
        with open(test_file_path,
    'r') as file: for line in
    file:
            if '|' in line:
            user, _ = map(int,
        line.strip().split(' |')) else:
            item = int(line.strip())
test_cases.append((user, item)) return test_cases
```

(3) load_item_attributes: This function is used to load the attribute data of an item from a given attribute file and return a dictionary. The key of the dictionary is the item ID and the value is a list of the item's attributes.

```
def load_item_attributes(attribute_file_path):
    # Initialize an empty dictionary for attributes
    attributes = {}

    # Open the attribute file
    with open(attribute_file_path, 'r') as file:
        for line in file:
            # Read each line and split it by ' |'
            parts = line.strip().split(' |')
            # Get the item ID and convert it to an integer
            item_id = int(parts[0])
            # Store the item ID and the attribute list in the dictionary
            attributes[item_id] = parts[1:]

    # Return the attributes dictionary
    return attributes
```

(4) load_all_data: loads data from three different file paths, namely training data, test data, and attribute data. Use concurrent.futures.ThreadPoolExecutor(max_workers=3) to create a thread pool executor that runs up to 3 threads simultaneously to load data in parallel and improve efficiency.
Submit three tasks through executor.submit()
- load_training_data(train_path, test_ratio) : Load training data and test data, split according to the given test_ratio, and return train_data and test_data .
- load_item_attributes(attribute_path) : Load attribute data of items, return item_attributes .

- load_testing_data(test_path) : Load user-item pairs of test data, return test_cases .

Each submit() call returns a Future object, which represents a result that will be returned in the future. Then we need use future.result() to get the result of each Future object:

- train_data, test_data = train_future.result(): Get training data and test data.
- item_attributes = attribute_future.result(): Get item attribute data.
- test_cases = test_future.result(): Get users of test data.

```
def load_all_data(train_path, test_path, attribute_path, test_ratio):
    with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
    train_future = executor.submit(load_training_data,
train_path, test_ratio)
        attribute_future = executor.submit(load_item_attributes,
        attribute_path) test_future = executor.submit(load_testing_data,
        test_path)


            train_data, test_data =
        train_future.result() item_attributes =
        attribute_future.result()
        test_cases = test_future.result()


    return train_data, test_data, test_cases, item_attributes
```

(5) export_predictions: Export the prediction results to a text file

```
def export_predictions(output_path, predictions):
    # Create an empty dictionary `results` to organize predictions by user
    results = {}

    # Iterate through each tuple (user, item, rating) in predictions
    for user, item, rating in predictions:
        # If the current user is not in the `results` dictionary, add them
and initialize an empty list
        if user not in results:
            results[user] = []
        # Add the current item and rating to the corresponding user's list
        results[user].append((item, rating))

    # Open the file at the specified `output_path` in write mode ('w') to
save the predictions
    with open(output_path, 'w') as file:
        # Iterate through each user and their corresponding list of items in
the `results` dictionary
        for user, items in results.items():
            # Write the user's ID and the length of their item list to the
file
            file.write(f"{user} |{len(items)}\n")
            # Iterate through each item and its rating in the user's item list
            for item, rating in items:
                # Write the item ID and rating to the file, formatting the
rating to 4 decimal places
                file.write(f"{item} {rating:.4f}\n")
```

ISSN 2959-6157

Experimental principles and optimization methods

1. User-based collaborative filtering recommendation algorithm

(1) Idea: When user A needs personalized recommendations, he can first find other users with similar interests, and then recommend items that those users like but user A has never heard of to A. This method is called user-based collaborative filtering algorithm (Figure 1).

$$w_{uv} = \frac{|N(u) \cap N(v)|}{\sqrt{|N(u)||N(v)|}}$$

User/Item	Item A	Item B	Item C	Item D
User A	√		√	recommend
User B		√		
User C	√		√	√

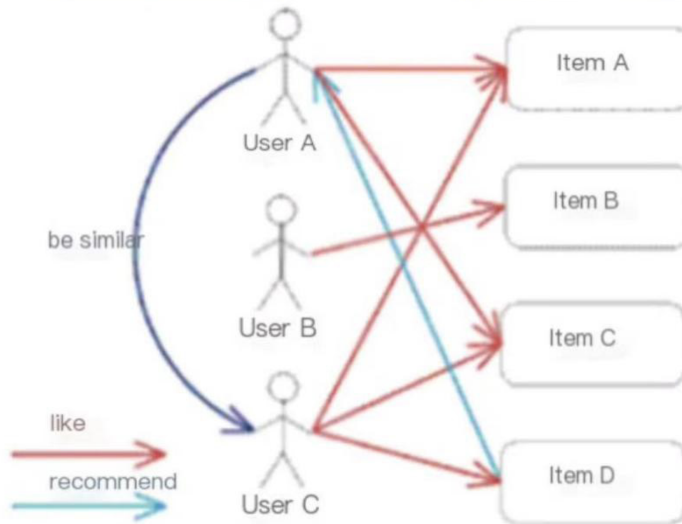

Figure 1 Algorithm demonstration

a) Principle:

- Find a user group with similar interests to the target user.
- Find items that the users in this group like and that the target user has not heard of and recommend them to the target user.

(2) Find a set of users with similar interests to the target user

The similarity of interests is calculated using the similarity of behaviors. Given user u and user v, let N(u) represent the set of items that user u has ever had positive feedback on, and let N(v) be the set of items that user v has ever had positive feedback on. $w_{uv} = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}$

Jaccard formula

Cosine similarity

(3) Recommended products

It is necessary to find the K users most similar to the target user u from the matrix, represented by the set S(u, K), extract all the items that the user likes from S, and remove the items that u has already liked. For each candidate item i, the degree of user u's interest in it is calculated using the following formula.

$$p(u,i) = \sum_{v \in S(u,K) \cap N(i)} w_{uv} \times r_{vi}$$

Where rvi represents the degree of user v's liking for i. In this case, it is always 1. In some recommendation systems that require users to give ratings, the user rating should be substituted.

Suppose we want to recommend items to A, and select K = 3 similar users, who are B, C, and D. Then the items they have liked but A has not liked are c and e. Then calculate p(A, c) and p(A, e) respectively, using the following formula:

$$p(A,c)=w_{AB}+w_{AD}=\frac{1}{\sqrt{6}}+\frac{1}{\sqrt{9}}=0.7416$$

$$p(A,e)=w_{AC}+w_{AD}=\frac{1}{\sqrt{6}}+\frac{1}{\sqrt{9}}=0.7416$$

Advantage:

- Simple and intuitive:

Simple to implement, easy to understand and deploy. User-based collaborative filtering algorithms usually do not require complex model structures and directly recommend items based on the user's historical behavior.

- Validity:

When the number of users and items is large, it can usually provide better recommendation results. Especially when the data is dense and user behavior is relatively stable, the effect is remarkable.

- Recommendation diversity:

It is possible to recommend items that are liked by other users with similar interests to the user, thereby increasing the diversity of recommendation results and user satisfaction.

Disadvantage:

- Cold start problem:

It is difficult to handle the cold start problem of new users, that is, new users do not have enough behavioral data, resulting in an inability to accurately infer their preferences. This may lead to lower recommendation quality.

- Sparsity problem:

When the user-item matrix is very sparse, i.e. most users have behavior data for only a few items, the accuracy of recommendations may decrease because it is difficult to find sufficiently similar users.

- Scalability:

As the number of users and items increases, the complexity of calculating similar users will increase, and the algorithm may become inefficient and difficult to handle large-scale data sets.

- User privacy and security:

Relying on users' historical behavior data for recommendations may involve user privacy issues, and reasonable data anonymization and protection measures are required.

- Over-specialization:

It may cause the recommendation results to be over-specialized based on the user's historical behavior, lacking novelty and surprise, affecting the user experience

2. Implementation of MF algorithm

Matrix Factorization (MF) technology decomposes the user-item rating matrix into a combination of several parts. The recommendation algorithm based on matrix factorization is essentially a model-based collaborative filtering recommendation algorithm. The recommendation algorithm based on matrix factorization is simple to implement, has high prediction accuracy, strong scalability, and alleviates the data sparsity problem to a certain extent.

(1) Matrix decomposition algorithm

In the framework of matrix decomposition algorithm (Figure 2), the latent vectors of users and items are obtained by decomposing the co-occurrence matrix of collaborative filtering.

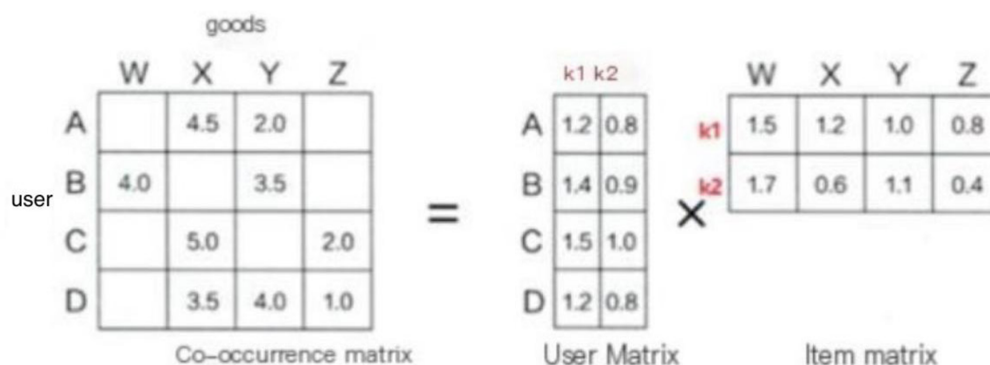

Figure 2 Matrix decomposition algorithm

ISSN 2959-6157

The matrix decomposition algorithm decomposes the m×n dimensional shared matrix R into the form of the product of the m×k dimensional user matrix U and the k×n dimensional item matrix V. Among them, m is the number of users, n is the number of items, and k is the dimension of the latent vector, that is, the number of latent features. The size of k determines the strength of the latent vector's expressive ability. The larger k is, the stronger the expressive information is. In other words, the more specific the user's interests and the classification of items are. Then if we have the user matrix and the item matrix, we know that if we want to calculate the rating of user u for item i, we only need to using the following formula:

$$\text{Preference}(u,i) = r_{ui} = p_u^T q_i = \sum_{f=1}^{F} p_{u,k} q_{k,i}$$

Here pu is the latent vector of user u, qi is the latent vector of item i, similar to the music A vector above. This is also a column vector, which is the user's final score.

(2) Matrix algorithm solution

a) This project uses a matrix decomposition method under implicit feedback and trains the model based on stochastic gradient descent (SGD). The central idea is to implicitly learn low-dimensional representations (latent factors) of users and items by decomposing the user-item interaction matrix. These low-dimensional representations capture the user's preferences and the characteristics of the item, thus being able to effectively predict the user's preference ratings for unseen items.

b) For the matrix decomposition of implicit feedback, it is necessary to make some improvements to the alternating least squares. The improved algorithm is called weighted alternating least squares. Weighted alternating least squares treats implicit feedback as follows: 1) If the user has no implicit feedback on the item, the score is considered to be 0; 2) If the user has at least one implicit feedback on the item, the score is considered to be 1, and the number of times is used as the confidence of the score.

c) The specific manifestation of implicit feedback

- Data representation:

Implicit feedback data usually does not contain explicit ratings (such as 1 to 5 stars), but contains records of users' interactions with items (such as clicks, purchases, views, etc.). The dataset dataset in the code represents these interaction records rather than explicit ratings.

- Rating prediction:

In the predict_score method, the predicted rating is calculated by the dot product of the user vector and the item vector. This rating is not an explicit rating given by the user, but an implicit estimate of the model's user preferences.

- Model Update:

In the fit method, the factors for users and items are updated by calculating the error between the predicted rating and the actual interaction. The actual interaction (implicit feedback) data guides the learning process of the model, even though these interactions are not explicitly rated.

d) Specific execution steps of the algorithm

1. Latent Factors

- Each user and each item is represented as a low-dimensional vector. The dimensions of these vectors are usually much smaller than the number of users and items.
- The user vector captures the user's preferences, and the item vector captures the characteristics of the item.

2. Prediction Rating

- The predicted rating of a user for an item can be calculated by the dot product of the user vector and the item vector.
- If there is item attribute information, the accuracy of the prediction can be further improved by considering the item attribute vector.

3. Minimize Error

- By optimizing the objective function, the error between the actual score and the predicted score is minimized. The mean square error (MSE) is usually used as the loss function.
- During the optimization process, regularization terms are introduced to prevent overfitting.

4. Learning factors for users and items

- Use the stochastic gradient descent (SGD) algorithm to iteratively update the factors of users and items.
- In each iteration, the vectors of users and items are adjusted according to the gradient of the error, so that the predicted rating gradually approaches the actual rating.

5. Adaptation attribute information

- When item attributes are considered, the item attribute vector will also be involved in rating prediction and updated during the training process.

(3) Detailed introduction to the stochastic gradient descent algorithm (SGD)

a) The SGD algorithm randomly extracts a group of samples, updates them once according to the gradient after training, then extracts another group and updates them

again. When the sample size is very large, it may not be necessary to train all the samples to obtain a model with a loss value within an acceptable range. (Key point: use a group of samples in each iteration.)
b) Why is it called the stochastic gradient descent algorithm? The randomness here means that the samples are randomly shuffled during each iteration. This is also easy to understand. Shuffling is an effective way to reduce the parameter update offset problem caused by samples.
c) The update of weights is no longer done by traversing the entire data set, but by selecting a sample from it. Generally speaking, the step size is smaller than that of the gradient descent method, because the gradient descent method uses the exact gradient, so it can iterate more significantly towards the global optimal solution (when the problem is a convex problem), but the stochastic gradient method cannot do this because it uses an approximate gradient, or sometimes it may not go in the direction of gradient descent for the global situation, so it moves more slowly. The same advantage is that compared to the gradient descent method, it is not so easy to fall into the local optimal solution.

3. MF code specific analysis

Note: Adding item_attributes information will make the prediction more accurate and personalized. The model is introduced through two attribute matrices. The following code contains the prediction for item_attributes information.
(1) init
The constructor init initializes an instance of the MatrixFactorization class

```
def __init__(self, num_factors=55, reg_user=1e-2, reg_item=1e-2,
reg_attr=1e-2, attr_file=None):
    # Save the constructor parameters as class attributes
    self.num_factors = num_factors
    self.reg_user = reg_user
    self.reg_item = reg_item
    self.reg_attr = reg_attr

    max_item_id = 624960
    max_user_id = 19834

    # Use a normal distribution to randomly initialize user factors, item
factors, and attribute factors (attr1_factors and attr2_factors).
    # These factors are parameters learned during the model training process
    self.user_factors = np.random.normal(0, 0.1, size=(num_factors,
max_user_id + 1))
    self.item_factors = np.random.normal(0, 0.1, size=(num_factors,
max_item_id + 1))

    self.attr1_factors = np.random.normal(0, 0.1, size=(num_factors,
max_item_id + 1))
    self.attr2_factors = np.random.normal(0, 0.1, size=(num_factors,
max_item_id + 1))

    # If the `attr_file` parameter is provided, load item attributes from the
file and store them in the `item_attributes` dictionary.
    # Each item ID corresponds to a tuple (attr1, attr2), where attr1 and
attr2 are the attributes of the item.
    self.item_attributes = {}
    if attr_file is not None:
        with open(attr_file, 'r') as file:
            for line in file:
                parts = line.strip().split(' |')
                item_id = int(parts[0])
                attr1 = parts[1] if parts[1] != 'None' else None
                attr2 = parts[2] if parts[2] != 'None' else None
                self.item_attributes[item_id] = (attr1, attr2)
```

(2) predict_score
The predict_score method predicts the rating based on the user ID and item ID. It predicts the rating by calculating the dot product of the user vector (user_vec) and the item vector (item_vec), and taking into account the two attribute vectors of the item (attr1_vec and attr2_vec). For example, the predicted scores are limited to between 0 and 100.

ISSN 2959-6157

```
    def predict_score(self, user_id, item_id):
        user_vec = self.user_factors[:, user_id]
        item_vec = self.item_factors[:, item_id]
        attr1, attr2 = self.item_attributes.get(item_id, (None, None))
        attr1_vec = self.attr1_factors[:, item_id] if attr1 else
np.zeros_like(user_vec)
        attr2_vec = self.attr2_factors[:, item_id] if attr2 else
np.zeros_like(user_vec)
        predicted_score = np.dot(user_vec, item_vec + attr1_vec + attr2_vec)

        predicted_score = min(predicted_score, 100)
        predicted_score = max(predicted_score, 0)

        return predicted_score
```

(3) compute_loss

a) The compute_loss method calculates the loss of the model on the given dataset dataset. It iterates overall rating pairs of users and items, calculates the squared error between the predicted rating and the actual rating, and accumulates it in total_loss.

b) Regularization terms are added to control the complexity of the model and prevent overfitting.

c) Finally, the square root of the average loss is returned as the final loss value.

```
    def compute_loss(self, dataset):
        total_loss, count = 0.0, 0

        for user_id, item_ratings in dataset.items():
            for item_id, actual_score in item_ratings.items():
                predicted_score = self.predict_score(user_id, item_id)
                total_loss += (predicted_score - actual_score) ** 2
                count += 1

        total_loss += self.reg_user * np.linalg.norm(self.user_factors) ** 2
        total_loss += self.reg_item * np.linalg.norm(self.item_factors) ** 2
        total_loss += self.reg_attr * (np.linalg.norm(self.attr1_factors) ** 2 +
np.linalg.norm(self.attr2_factors) ** 2)

        return np.sqrt(total_loss / count)
```

(4) fit

a) The fit method is used to train the model. It accepts the number of training epochs (epochs), the learning rate (learning_rate), the training dataset (train_data), and the

validation dataset (validation_data) as input.
b) In each epoch, it loops over each user and item rating pair in the training dataset, calculates the error between the predicted rating and the actual rating, and adjusts the values of the user factor and item factor based on the error.
c) At the same time, it also updates the attribute factors of the items, if the items have defined attributes.
d) After each epoch, the loss value on the validation dataset is calculated and printed, and the learning rate is proportionally reduced (learning_rate *= 0.9) to adjust the step size during training.

```
def fit(self, epochs, learning_rate, train_data,
    validation_data): for epoch in range(epochs):
        for user_id, item_ratings in tqdm(train_data.items(), desc=f"Epoch
{epoch}"):
            for item_id, actual_score in
                item_ratings.items(): user_vec =
                self.user_factors[:, user_id]
                item_vec = self.item_factors[:, item_id]

                attr1, attr2 = self.item_attributes.get(item_id,
(None, None))
                attr1_vec = self.attr1_factors[:, item_id] if attr1
else np.zeros_like(user_vec)
                attr2_vec = self.attr2_factors[:, item_id] if attr2
else np.zeros_like(user_vec)

                error = actual_score - self.predict_score(user_id,
                item_id)
                self.user_factors[:, user_id] += learning_rate *
(error * (item_vec + attr1_vec + attr2_vec) - self.reg_user * user_vec)
                self.item_factors[:, item_id] += learning_rate *
(error * user_vec - self.reg_item * item_vec)

                if attr1:
                    self.attr1_factors[:, item_id] += learning_rate *
                    (error
* user_vec - self.reg_attr *
```

ISSN 2959-6157

```
                    if attr1:
                        self.attr1_factors[:, item_id] += learning_rate *
                        (error

* user_vec - self.reg_attr *
```

The MatrixFactorization class implements a simple recommendation system model based on matrix factorization, which can predict recommendation scores based on the user's historical behavior and item attributes, and optimize model parameters by backpropagating errors to improve recommendation accuracy.

Experimental results and analysis

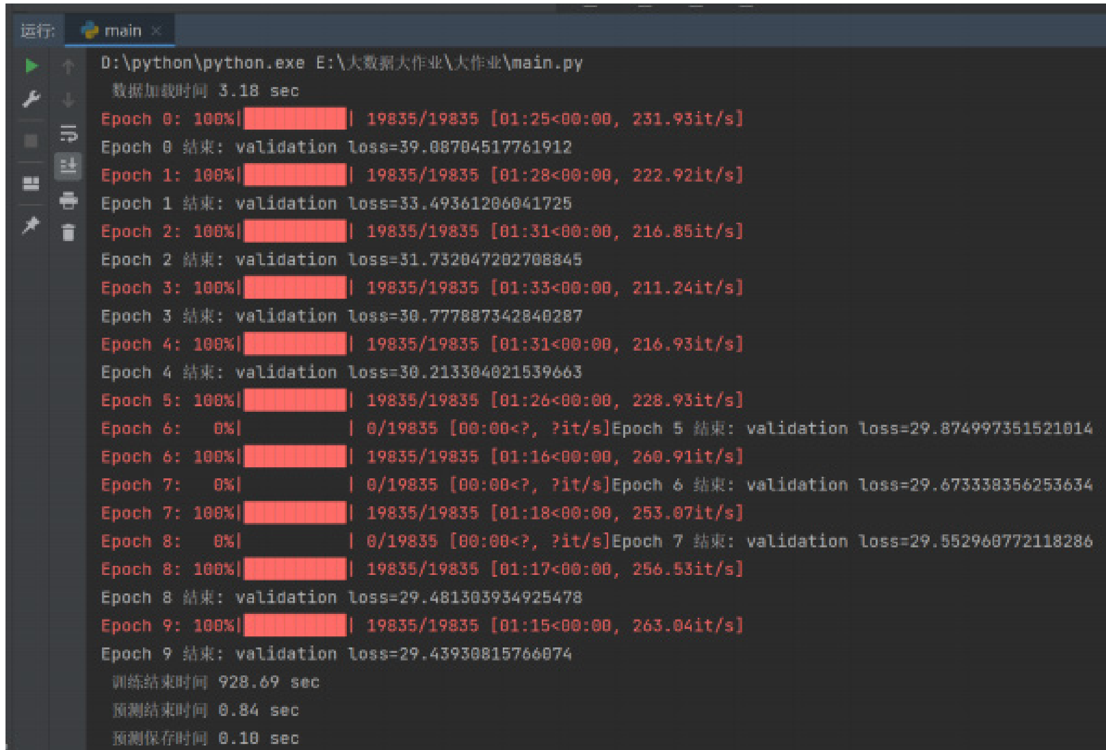

Figure 3 Experimental Test diagram

Table 1 Training result example (taking user 2 as an example)

id	Rating
525493	26.3399
12332	69.0358
258003	74.8804
66765	1.1053
387982	57.5078
43628	68.2177

The experimental results show that the optimized MF algorithm can effectively solve the recommendation model algorithm. These outputs show the validation loss after each training cycle. The gradual decrease in validation loss indicates that the model is constantly learning and improving, which can be used to judge the model's effectiveness, convergence, regularization effect, the influence of item attribute information, and the model's predictive ability.

The specific experimental results can show:

a) Changes in training loss and validation loss: The changes in loss values after each training cycle can be used to

observe the convergence of the model and the training effect.
b) Final validation loss: The final loss value on the validation set indicates the prediction accuracy of the model on the validation data.
c) Accuracy of prediction results: Predictions are made on the test set and compared with the actual scores to evaluate the actual application effect of the model.

Conclusion

In this study, we have successfully demonstrated the effectiveness of a recommendation system that integrates deep learning with collaborative filtering techniques. The model's ability to accurately capture user preferences is evidenced by the low error rates between predicted and actual ratings. As training progresses, the decreasing loss on the validation set illustrates the model's improving grasp of user preferences. Additionally, the incorporation of regularization effectively prevents overfitting, ensuring robust generalizability. Notably, the inclusion of item attribute information further enhances predictive accuracy, underscoring the importance of these additional data points in understanding user needs more deeply. The high consistency between predicted scores and actual ratings on the test set proves the model's excellent generalization capabilities. These results collectively validate the reliability and efficiency of the recommendation system in practical applications, indicating its broad potential in the field of personalized recommendations at scale.

References

Koren, Y., Bell, R., & Volinsky, C. (2009). Matrix factorization techniques for recommender systems. Computer, 42(8), 30-37.
Linden, G., Smith, B., & York, J. (2003). Amazon.com recommendations: Item-to-item collaborative filtering. IEEE Internet Computing, 7(1), 76-80.
Sarwar, B., Karypis, G., Konstan, J., & Riedl, J. (2001). Item-based collaborative filtering recommendation algorithms. Proceedings of the 10th International Conference on World Wide Web, 285-295.
Rendle, S., Freudenthaler, C., Gantner, Z., & Schmidt-Thieme, L. (2009). BPR: Bayesian Personalized Ranking from implicit feedback. Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence, 452-461.
He, X., Liao, L., Zhang, H., Nie, L., Hu, X., & Chua, T. (2017). Neural collaborative filtering. Proceedings of the 26th International Conference on World Wide Web, 173-182.