The Project Progress Updates 1

Achievements

➢ Algorithm

Following the initial screening phase, the proposed plan focuses on designing a robust framework that integrates multiple algorithms for search and promotion functionalities. The objective is to enhance user experience by delivering more accurate, comprehensive, and personalized recommendations. The framework combines a tag-based recommendation algorithm and a collaborative filtering algorithm, leveraging their complementary strengths to address diverse user needs. Below, we outline the key components, implementation strategies, and integration approach.

1. Tag-Based Recommendation Algorithm

The tag-based recommendation system aims to capture explicit user preferences and facilitate initial matching, particularly for new users. At the point of registration, a structured tag input interface will be introduced, comprising two main categories:

- **Self-Description Tags**: Examples include “love to travel,” “anime enthusiast,” or “fitness expert,” reflecting users’ personal traits or identities.
- **Interests and Preferences Tags**: Examples such as “want to meet funny people,” “like to discuss technology,” or “prefer quiet personalities” indicate desired connections or content.

To ensure scalability and consistency, a predefined library of commonly used tags will be provided, supplemented by an option for users to input custom tags. Custom inputs will undergo review or clustering (e.g., using K-Means or DBSCAN) to prevent excessive fragmentation. Tag similarity between users will be calculated using established metrics such as Cosine Similarity or the Jaccard Similarity Coefficient. For instance, if User A has tags {Travel, Food, Movies} and User B has {Travel, Music, Photography}, the Jaccard similarity is computed as $|A \cap B| / |A \cup B| = 1/6$.

To enhance precision, users will be prompted to prioritize their tags during onboarding, assigning higher weights to critical preferences (e.g., “must like pets” could carry double the weight of standard tags). In the initial stage, recommendations will be generated by filtering the database for the Top-N users with the highest tag similarity scores. As user behavior data accumulates (e.g., browsing history, likes), tag weights will be dynamically updated, and implicit tags (e.g., “sports enthusiast” for frequent sports content viewers) will be inferred and added.

For efficient computation, user tags will be converted into vector representations, such as One-Hot encoding or TF-IDF, enabling similarity calculations. Additionally, clustering techniques like K-Means or DBSCAN will be employed to group users with similar interests, reducing the computational load of pairwise matching and identifying latent interest communities.

2. Collaborative Filtering Algorithm

While the tag-based approach provides a solid foundation, it lacks the depth to capture dynamic preferences or behavioral nuances. The collaborative filtering algorithm addresses this by analyzing user interactions, including explicit actions (e.g., likes, comments, chat frequency, browsing duration) and implicit feedback (e.g., skipping recommended items as negative signals). Two complementary strategies will be implemented:

- **User-Based Collaborative Filtering**: This method identifies users with similar behavioral patterns. For example, if User A and User B exhibit comparable interactions (e.g., liking or engaging with Users C and D), their interests are likely aligned. Behavioral similarity will be measured using Pearson’s Correlation Coefficient or Cosine Similarity, enabling recommendations of items User B enjoys but User A has not yet encountered.
- **Item-Based Collaborative Filtering**: This approach focuses on object similarity. If User A likes User X, and User X shares behavioral patterns with User Y (e.g., both are frequently liked by the same group), User Y may be recommended to User A.

Implementation will involve constructing a user-object interaction matrix, followed by similarity calculations between objects to recommend related items. To uncover latent features, matrix decomposition techniques such as Singular Value Decomposition (SVD) or Alternating Least Squares (ALS) will be applied. Alternatively, neural network-based methods (e.g., Autoencoders or Embeddings) will transform user behavior into low-dimensional vectors, improving recommendation accuracy and scalability.

3. Integration of Tag-Based and Collaborative Filtering Algorithms

The fusion of these algorithms is designed to balance explicit user input with inferred preferences, adapting to users’ lifecycle stages. For new users with limited behavioral data, the system will prioritize tag-based recommendations to quickly match them with similar individuals based on stated interests. As behavioral data accumulates, collaborative filtering will assume a more dominant role, leveraging interaction patterns to uncover deeper preferences. The tag-based approach will continue to serve as a corrective mechanism, ensuring explicit interests are not overlooked.

The integration will employ a linear weighting formula to combine recommendation scores:

Score = α * Label Similarity + (1-α) * Collaborative Filtering Score

Here, α represents the weight assigned to the tag-based component. For new users, α will be set close to 1, emphasizing tag similarity. As usage time and data volume increase, α will gradually decrease, shifting reliance toward collaborative filtering. The exact adjustment of α will be fine-tuned based on user engagement metrics and A/B testing results.

To ensure compatibility, both tag and behavioral data will be unified into vector representations. Tag embeddings will be trained using techniques like Word2Vec, while behavioral embeddings will be derived from interaction data. Operationally, tag-based matching will be updated daily via batch processing, while collaborative filtering will support real-time adjustments through online computation, ensuring responsiveness to user activity.

➢ Software Development

During this time, the main tasks of software development are Services Separation, API and Database Design.

1. **User Service**

- **Responsibilities:** User lifecycle management, credential management, user profile maintenance
- **Core functions:** registration/login/logout, authentication and authorization,
- **Technical features:** Spring Security + JWT
- **API:**

```
POST /api/v1/auth/register - User register
POST /api/v1/auth/login - User login
POST /api/v1/auth/logout - user logout
POST /api/v1/auth/refresh-token - Refresh token
GET /api/v1/users/{id} - Get user information
PUT /api/v1/users/{id} - Update user profile
GET /api/v1/users/settings - Get user settings
PUT /api/v1/users/settings - Update user settings
```

- **MySQL tables:**

```
CREATE TABLE users (
  id BIGINT PRIMARY KEY AUTO_INCREMENT,
  username VARCHAR(50) UNIQUE NOT NULL,
  email VARCHAR(100) UNIQUE NOT NULL,
  phone VARCHAR(20),
  password_hash VARCHAR(255) NOT NULL,
  account_status ENUM('ACTIVE', 'INACTIVE', 'SUSPENDED', 'DELETED') NOT NULL,
  created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);

CREATE TABLE user_settings (
  id BIGINT PRIMARY KEY AUTO_INCREMENT,
  user_id BIGINT NOT NULL,
  notification_email BOOLEAN DEFAULT TRUE,
  notification_push BOOLEAN DEFAULT TRUE,
  notification_sms BOOLEAN DEFAULT FALSE,
  privacy_level ENUM('PUBLIC', 'PRIVATE', 'FRIENDS_ONLY') DEFAULT 'PUBLIC',
  display_online_status BOOLEAN DEFAULT TRUE,
  display_last_active BOOLEAN DEFAULT TRUE,
  created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  FOREIGN KEY (user_id) REFERENCES users(id)
);
```

- **MongoDB:**

```
user_preferences {
  _id: ObjectId,
  user_id: Long,
  age_range: { min: Number, max: Number },
  distance_preference: Number,
  gender_preference: [String],
  interests: [String],
  important_traits: [String],
  deal_breakers: [String],
  matching_priorities: Object,
  created_at: Date,
  updated_at: Date
}
```

2. **Matching Service**
   - **Responsibilities:** Intelligent recommendation engine, matching algorithm execution.
   - **Core functions:** recommendations based on user portraits (collaborative filtering + deep learning), dynamic weight adjustment (behavioral data feedback), matching pool management.
   - **API:**

```
GET /api/v1/matches/recommendations - Get recommended user list
GET /api/v1/matches - Get successful matching
DELETE /api/v1/matches/{matchId} - remove matching
GET /api/v1/matches/compatibility/{targetUserId} - Get compatibility score
```

   - **MongoDB:**

```
user_interactions {
  _id: ObjectId,
  user_id: Long,
  target_user_id: Long,
  interaction_type: String, // 'LIKE', 'DISLIKE', 'SUPERLIKE'
  created_at: Date
}

matches {
  _id: ObjectId,
  user1_id: Long,
  user2_id: Long,
  match_score: Number,
  match_factors: Object,
  match_date: Date,
  status: String, // 'ACTIVE', 'INACTIVE', 'BLOCKED'
  last_interaction_date: Date
}

recommendation_history {
  _id: ObjectId,
  user_id: Long,
  recommended_users: [Long],
  factors: Object,
  created_at: Date
}
```

3. **Chat Service**
   - **Responsibilities:** Real-time communication management
   - **Core functions:** webSocket message routing, message storage and state synchronization, illegal message filtering (AI content review)
   - **Technical features:** WebSocket
   - **API:**

```
WebSocket: /ws/chat
GET /api/v1/chat/conversations - get conversation list
GET /api/v1/chat/conversations/{conversationId}/messages - get chatting history
POST /api/v1/chat/conversations/{conversationId}/messages - send a message
PUT /api/v1/chat/messages/{messageId}/read - mark a message as read
```

   - **MongoDB:**

```
conversations {
    _id: ObjectId,
    participants: [Long],
    last_message: Object,
    created_at: Date,
    updated_at: Date
}

messages {
    _id: ObjectId,
    conversation_id: ObjectId,
    sender_id: Long,
    message_type: String, // 'TEXT', 'IMAGE', 'VIDEO', 'AUDIO', 'LOCATION'
    content: String,
    media_url: String,
    read_by: [Long],
    created_at: Date
}
```

4. **Media Service**

- **Responsibilities:** Multimedia content management
- **Core functions:** AI image/video review, CDN distribution management.
- **Technical features:** FFmpeg processing, OSS storage
- **API:**
  ```
  POST /api/v1/media/upload - upload media
  GET /api/v1/media/{mediaId} - get media
  DELETE /api/v1/media/{mediaId} - delete media
  POST /api/v1/media/{mediaId}/moderate - moderate media by administrators
  ```
- **MongoDB:**
  ```
  media_assets {
    _id: ObjectId,
    user_id: Long,
    file_name: String,
    file_type: String,
    file_size: Number,
    url: String,
    thumbnail_url: String,
    dimensions: {
      width: Number,
      height: Number
    },
    moderation_status: String, // 'PENDING', 'APPROVED', 'REJECTED'
    ai_tags: [String],
    created_at: Date
  }

  media_moderation {
    _id: ObjectId,
    media_id: ObjectId,
    moderation_type: String, // 'AUTO', 'MANUAL'
    status: String,
    rejection_reason: String,
    moderator_id: Long,
    moderation_date: Date
  }
  ```

5. **Notification Service**

- **Responsibilities:** Asynchronous message push
- **Core functions:** notification/SMS/email sending
- **Technical features:** MQ peak shaving and valley filling, multi-channel routing
- **API:**
  ```
  GET /api/v1/notifications - get latest notifications
  PUT /api/v1/notifications/{notificationId}/read - mark a notification as read
  ```
- **MongoDB:**

```
notifications {
  _id: ObjectId,
  user_id: Long,
  type: String, // 'MATCH', 'MESSAGE', 'LIKE', 'SYSTEM' , etc.
  title: String,
  content: String,
  related_entity_id: String,
  related_entity_type: String,
  is_read: Boolean,
  created_at: Date
}
```

- **Message Queue:**

```
notification.email - queue for email notification
notification.sms - queue for SMS notification
```

Challenges

➢ Algorithm

1. User-defined tags, particularly custom entries, may introduce semantic noise and inconsistencies. Although clustering and moderation mechanisms will be in place, maintaining a clean, scalable tag taxonomy requires continuous refinement and oversight.
2. Collaborative filtering models heavily rely on sufficient interaction data. In early-stage systems or low-activity segments, data sparsity can hinder the accuracy of similarity measurements and latent feature learning.
3. The hybrid model depends on an effective method to blend tag-based and behavioral recommendations. Designing a dynamic, adaptive weighting mechanism (parameter α) that reflects user lifecycle and engagement metrics is an open challenge.

➢ Software Development

1. **Distributed Token Management**
   - **Problem:** JWT token revocation in a multi-service architecture led to potential security risks when users logged out or tokens expired.
   - **Solution:** Implemented a Redis-based token blacklist with TTL synchronization, ensuring immediate token invalidation across all services.
2. **Cross-Database Consistency**
   - **Problem:** User profile updates required synchronization between MySQL (users table) and MongoDB (user_preferences), risking data inconsistency.
   - **Solution:** Introduced a transactional outbox pattern with Kafka events to propagate changes asynchronously, achieving eventual consistency.

Next Plans

➢ Algorithm

1. Convert user tags into machine-readable vectors using One-Hot Encoding or TF-IDF. Each user profile will be represented as a sparse or dense vector depending on tag frequency and weight.
2. Implement Jaccard and Cosine Similarity functions to compare user vectors. Enable batch scoring to identify Top-N similar users from the database.
3. Use unsupervised learning (e.g., K-Means or DBSCAN) to cluster semantically similar custom tags. Apply dimensionality reduction (e.g., PCA or UMAP) for visualization and manual inspection of tag spaces.
4. Build a sparse matrix with users as rows and target users as items. Populate matrix entries based on explicit (likes) and implicit (views, skips) behavior.
5. Implement both user-based and item-based collaborative filtering using Pearson Correlation or Cosine Similarity.

➢ Software Development

1. Specific implementation of user identity authentication system

- **Email verification:** When registering, users need to verify their email address. The system sends a 6-digit verification code (valid for 5 minutes). The registration request must be accompanied by a verification code.
- **Anti-bot mechanism:** IP flow control for multiple registration requests in a short period of time (such as up to 3 times per minute)
- **Risk login detection:** Limit the number of login failures: 5 consecutive failures within 5 minutes, the account will be locked for 30 minutes.
- **JWT (JSON Web Token) authentication:** Access tokens are valid for 15 minutes, and refresh tokens are valid for 7 days (stored in Redis). Token blacklist, that is, when the user actively logs out or the token expires, it is added to the Redis blacklist to prevent theft.

2. Specific implementation of user registration

- **Registration form submission**
  - **Front-end collection:** user name, email/mobile number, password, password confirmation
  - **Basic verification:** field is not empty, email format, password consistency
- **Server-side processing**
  - **Uniqueness check:** check whether the username/email already exists
  - **Password security processing:** strength verification (length, complexity)
  - **Account initialization:** set the default user status (ACTIVE)
- **Generate a unique user ID**
  - Create a default record for associated users
- **Response processing**
  - **Success:** return 201 Created, including basic user information (excluding sensitive data)
  - **Failure:** return a specific error (such as "email already registered")
- **Data storage**
  - MySQL user table records core information
  - MongoDB stores extended attributes (such as preferences)

    - Redis caches newly registered users (anti-duplicate submission)

3. Specific implementation of user login service

- **Credential verification:**
    - Support username/email + password login
    - Account status check: whether it is locked/disabled
- **Session creation:**
    - Record login device/IP information
- **Response processing:**
    - **Success: r**eturn 200 OK, including Access Token and user basic information
    - **Failure:** Return specific reasons (such as "wrong password", remaining number of attempts)

4. Supporting basic services

- **Account activation**
    - Send activation link (including time-limited token) after registration
    - Limited function usage for unactivated accounts
- **Password service**
    - Password reset (via verification email)
    - Password change (need to verify the original password)
- **Session management**
    - Login status query
    - Active logout (token expired)
    - Multi-device session management