Configuration Manual

Large Language Model Driven User Cold Start Recommendations

MSc Research Project
Data Analytics

Shubham Dalvi
Student ID: x23268051

School of Computing
National College of Ireland

Supervisor: Teerath Kumar Menghwar

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Shubham Dalvi
Student ID:	x23268051
Programme:	Data Analytics
Year:	2024-2025
Module:	MSc Research Project
Supervisor:	Teerath Kumar Menghwar
Submission Due Date:	9/15/2025
Project Title:	Large Language Model Driven User Cold Start Recommendations
Word Count:	2196
Page Count:	20

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	September 15, 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Contents

1 Introduction

This configuration manual provides comprehensive setup and deployment instructions for the LLM-Enhanced Recommendation System research project. The project addresses the cold start problem in recommendation systems using Large Language Models (LLMs), Retrieval Augmented Generation (RAG), and intelligent questionnaire-based preference elicitation.

The system implements a full-stack web application with React frontend and FastAPI backend, featuring both content-based and collaborative filtering algorithms enhanced by OpenAI's GPT-3.5-turbo model. This manual details the complete environment setup, software dependencies, hardware requirements, and step-by-step implementation procedures necessary to reproduce the research results.

1.1 Research Context

The cold-start user problem is a long-standing issue in recommendation systems, where new users receive poor recommendations due to lack of interaction history. This project addresses this challenge through:

- Intelligent questionnaire-based preference elicitation
- LLM-enhanced genre weight generation
- Hybrid recommendation approaches combining content-based and collaborative filtering
- Comprehensive evaluation using real MovieLens user data

1.2 Document Structure

This manual is organized as follows:

- **System Overview**: Architecture, core features, dataset description, and project structure
- **Requirements**: Hardware and software specifications
- **Environment Setup**: Step-by-step installation guide
- **Implementation**: Detailed configuration procedures and deployment steps
- **API Documentation**: RESTful endpoint specifications and usage examples
- **References**: Documentation and academic references

2 System Overview

2.1 Project Architecture

The recommendation system consists of four main components:

- **Frontend Application**: React-based web interface with Tailwind CSS

- **Backend API**: FastAPI REST service with machine learning algorithms
- **Recommendation Engines**: Content-based and Collaborative filtering implementations
- **Evaluation Framework**: Comprehensive offline evaluation using MovieLens dataset

2.2 Core Features

- 18-question preference elicitation questionnaire
- LLM-enhanced genre weight generation using OpenAI GPT-3.5-turbo
- Content-based filtering for cold-start users
- User-based collaborative filtering with Pearson correlation
- Real-time recommendation generation
- Comprehensive evaluation metrics (MAE, RMSE, Precision@K, Recall@K, MRR, MAP, NDCG)

2.3 Technology Stack

2.3.1 Frontend Technologies

- **React 18.2.0**: Modern JavaScript library for building user interfaces
- **Tailwind CSS 3.3.6**: Utility-first CSS framework
- **React Router DOM 6.8.1**: Client-side routing
- **Axios 1.6.2**: HTTP client for API communication
- **Lucide React**: Modern icon library

2.3.2 Backend Technologies

- **FastAPI**: High-performance web framework for building APIs
- **Python 3.8+**: Core programming language
- **OpenAI GPT-3.5-turbo**: Large Language Model integration
- **Pandas & NumPy**: Data manipulation and numerical computing
- **Scikit-learn**: Machine learning algorithms and evaluation metrics

2.4 Data Flow Architecture

1. **User Registration**: New users complete the 18-question preference questionnaire
2. **Preference Processing**: LLM analyzes responses to generate normalized genre weights
3. **Content-Based Recommendations**: Initial recommendations based on genre preferences
4. **User Interaction**: Users rate recommended movies to build interaction history
5. **Collaborative Filtering**: Enhanced recommendations using user similarity
6. **Continuous Learning**: System adapts as more interaction data becomes available

2.5 Dataset Description

The recommendation system utilizes the MovieLens dataset, a widely-used benchmark dataset in recommendation system research. The dataset contains movie ratings and metadata collected from the MovieLens website (`https://movielens.org/`), a movie recommendation service operated by GroupLens Research at the University of Minnesota.

2.5.1 Dataset Characteristics

- **Dataset Version**: MovieLens Latest Small Dataset
- **Scale**: Approximately 100,000 ratings and 3,600 tag applications
- **Movies**: 9,000+ movies with release dates from 1902 to 2018
- **Users**: 600+ users who have rated at least 20 movies
- **Rating Scale**: 5-star rating system (0.5 to 5.0 stars)
- **Data Format**: CSV files containing ratings, movies, tags, and links

2.5.2 Data Files

- **ratings.csv**: User ratings with timestamps
- **movies.csv**: Movie metadata including titles and genres
- **tags.csv**: User-generated movie tags
- **links.csv**: Movie identifiers for IMDB and TMDB

The dataset is used for both training the recommendation algorithms and conducting comprehensive offline evaluation of system performance.

2.6 Project Directory Structure

```
project-root/
|-- recommendation-system/
|   |-- frontend/                    # React frontend application
|   |   |-- src/
|   |   |   |-- components/          # React components
|   |   |   |-- pages/               # Application pages
|   |   |   |-- context/             # React context providers
|   |   |   |-- hooks/               # Custom React hooks
|   |   |   |-- utils/               # Utility functions
|   |   |-- public/                  # Static assets
|   |   |-- package.json             # Frontend dependencies
|   |   |-- tailwind.config.js       # Tailwind CSS configuration
|   |-- backend/                     # FastAPI backend
|       |-- app/
|       |   |-- main.py              # Main FastAPI application
|       |-- collaborative_filtering.py
|       |-- content_based_filtering.py
|       |-- movie_rating_service.py
|       |-- requirements.txt         # Backend dependencies
|       |-- data/                    # Data storage directory
|-- evaluating-recsys-offline/       # Evaluation framework
|   |-- comprehensive_movielens_evaluation.py # Core-Evaluations
|   |-- strategy_comparison_evaluator.py      # multistrategy comparison
|   |-- enhanced_user_profile_creator.py      # User Profile creator
|   |-- improved_profile_generator.py         # supporting profiling
    strategies
|   |-- enhanced_user_profile/       # Enhanced user profiles (JSON)
|-- ml-latest-small/                 # MovieLens dataset
|   |-- ratings.csv                  # User ratings with timestamps
|   |-- movies.csv                   # Movie metadata and genres
|   |-- tags.csv                     # User-generated movie tags
|   |-- links.csv                    # Movie identifiers (IMDB, TMDB)
|   |-- README.txt                   # Dataset documentation
```

3 Hardware and Software Requirements

3.1 System Configurations

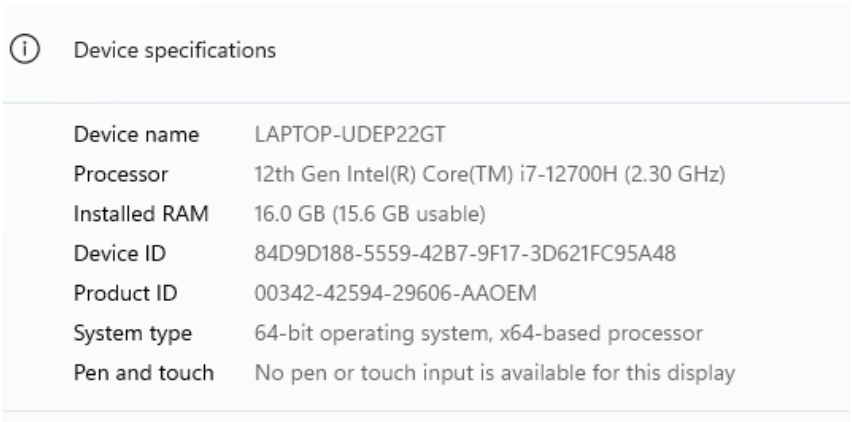

Figure 1: System Configuration

Component	**Specification**
Storage	1.5 TB
Graphics Card 1	NVIDIA RTX 4060, 8 GB
Graphics Card 2	Intel Iris(R) Xe Graphics
RAM	16.0 GB (3200 MHz)
Processor	12th Gen Intel(R) Core(TM) i7-12700H @ 2.30 GHz
Operating System	Windows 11 (Build 10.0.26100)

3.2 Software Dependencies

3.2.1 Frontend Dependencies (Node.js/React)

Package	**Version**	**Purpose**
React	^18.2.0	Core frontend framework
React Router DOM	^6.8.1	Client-side routing
Axios	^1.6.2	HTTP client for API calls
Tailwind CSS	^3.3.6	Utility-first CSS framework
Lucide React	^0.294.0	Icon library
PostCSS	^8.4.32	CSS post-processor
Autoprefixer	^10.4.16	CSS vendor prefixing
React Scripts	^5.0.1	Build and development tools

3.2.2 Backend Dependencies (Python)

Package	Version	Purpose
FastAPI	Latest	Web framework for API development
Uvicorn[standard]	Latest	ASGI server for FastAPI
Pydantic	Latest	Data validation using Python type annotations
OpenAI	Latest	LLM integration for genre weight generation
Pandas	Latest	Data manipulation and analysis
NumPy	Latest	Numerical computing
Python-multipart	Latest	File upload support
Python-dotenv	Latest	Environment variable management
HTTPX	Latest	Async HTTP client

3.2.3 Additional Python Libraries for Evaluation

Package	Version	Purpose
Scikit-learn	Latest	Machine learning algorithms and metrics
Matplotlib	Latest	Data visualization
Seaborn	Latest	Statistical data visualization
Jupyter Notebook	Latest	Interactive development environment

3.3 Prerequisites

Before starting the implementation, ensure the following software is installed:

- **Node.js** (version 16.0 or higher)
- **Python** (version 3.8 or higher)
- **Git** for version control
- **Code Editor** (VS Code recommended)
- **OpenAI API Key** for LLM integration

3.4 Minimum System Requirements

Component	Minimum Requirement
RAM	8 GB (16 GB recommended)
Storage	5 GB free space
Processor	Dual-core 2.0 GHz (Quad-core recommended)
Internet	Stable broadband connection
Browser	Chrome 90+, Firefox 90+, Safari 14+, Edge 90+

4 Environment Setup

4.1 Backend Setup

4.1.1 1. Create Python Virtual Environment

```
# Navigate to backend directory
cd phase2/backend

# Create virtual environment
python -m venv recommendation_env

# Activate virtual environment
# On Windows:
recommendation_env\Scripts\activate
# On macOS/Linux:
source recommendation_env/bin/activate
```

4.1.2 2. Install Backend Dependencies

```
# Install required packages
pip install fastapi uvicorn[standard] pydantic python-multipart
pip install python-dotenv openai pandas numpy httpx
pip install scikit-learn matplotlib seaborn jupyter
```

4.1.3 3. Environment Configuration

Create a `.env` file in the backend directory:

```
# OpenAI API Configuration
OPENAI_API_KEY=your_openai_api_key_here

# Application Configuration
DEBUG=True
CORS_ORIGINS=http://localhost:3000

# Data Configuration
MOVIELENS_DATA_PATH=../../ml-latest-small/
UI_USERS_FILE=data/ui_users.json
USER_RESPONSES_FILE=data/user_responses.json
```

4.1.4 4. Start Backend Server

```
# Run FastAPI server
uvicorn app.main:app --reload --host 0.0.0.0 --port 8000
```

The backend API will be available at `http://localhost:8000`

4.2 Frontend Setup

4.2.1 1. Install Node.js Dependencies

```
# Navigate to frontend directory
cd phase2/frontend

# Install dependencies
npm install
```

4.2.2 2. Configure Environment

Create a `.env` file in the frontend directory:

```
# API Configuration
REACT_APP_API_BASE_URL=http://localhost:8000
```

4.2.3 3. Start Development Server

```
# Start React development server
npm start
```

The frontend application will be available at `http://localhost:3000`

4.3 Dataset Configuration

4.3.1 MovieLens Dataset Setup

1. Download the MovieLens 100K dataset from `https://grouplens.org/datasets/movielens/`
2. Extract the dataset to the `ml-latest-small/` directory
3. Verify the following files exist:
    - `movies.csv` - Movie metadata
    - `ratings.csv` - User ratings
    - `tags.csv` - User-generated tags
    - `links.csv` - Movie links to external databases

4.3.2 Data Processing

The system automatically processes the MovieLens data during initialization:

- Filters movies with at least 5 ratings
- Filters users with at least 5 ratings
- Normalizes genre information
- Creates recommendation matrices

4.4 Development Environment Verification

4.4.1 Backend Health Check

```
# Test backend API
curl http://localhost:8000/health
```

Expected response:

```
{
  "status": "healthy",
  "timestamp": "2024-01-01T12:00:00Z"
}
```

4.4.2 Frontend Accessibility

1. Open browser and navigate to `http://localhost:3000`
2. Verify the questionnaire page loads correctly
3. Test API connectivity by submitting a sample questionnaire
4. Confirm recommendation generation functionality

5 Implementation

5.1 Core Implementation Components

5.1.1 Questionnaire-Based Preference Elicitation

The system implements an 18-question preference elicitation framework that captures multidimensional user preferences:

- **Pacing and Visual Preferences**: Questions about movie tempo and visual importance
- **Character Complexity**: Preferences for character development and moral complexity
- **Thematic Content**: Mood preferences (uplifting vs. dark content)
- **Narrative Structure**: Linear vs. complex storytelling preferences
- **Temporal Preferences**: Contemporary vs. classic film preferences
- **Social Viewing Patterns**: Individual vs. group viewing behaviors

5.1.2 LLM-Enhanced Genre Weight Generation

The system employs OpenAI's GPT-3.5-turbo model for intelligent preference analysis:

```
SYSTEM_PROMPT = """
You are a movie recommendation expert who understands how to weight different movie genres based on user preferences.

Given information about a user's:
1. Explicitly preferred genres
2. Explicitly disliked genres
3. Detailed questionnaire responses about their movie preferences

Analyze ALL these factors TOGETHER to determine appropriate weights for ALL MovieLens genres.

CRITICAL INSTRUCTION: While the user has explicitly selected some genres as "preferred" and others as "disliked",
DO NOT automatically assign highest weights to preferred genres. Instead, analyze the questionnaire responses
to determine which genres TRULY match the user's taste profile. The questionnaire responses may reveal interests
in genres the user didn't explicitly select, or disinterest in genres they did select.

For example, if a user selects "Documentary" as preferred, but their questionnaire responses show they prefer
fast-paced action, visually impressive effects, fantasy worlds, and minimal dialogue - they likely would enjoy
Action and Fantasy genres MORE than Documentary, despite their explicit selection.

Provide genre weights as decimal values between 0 and 1, where the weights sum to 1.0.
You MUST include weights for ALL 19 MovieLens genres:

The 19 MovieLens genres are:
Action, Adventure, Animation, Children, Comedy, Crime, Documentary, Drama, Fantasy, Film-Noir,
Horror, IMAX, Musical, Mystery, Romance, Sci-Fi, Thriller, War, Western

For example, a complete response might look like:
{
  "Action": 0.15, "Adventure": 0.12, "Animation": 0.03, "Children": 0.02, "Comedy": 0.08,
  "Crime": 0.09, "Documentary": 0.04, "Drama": 0.12, "Fantasy": 0.06, "Film-Noir": 0.03,
  "Horror": 0.02, "IMAX": 0.02, "Musical": 0.01, "Mystery": 0.07, "Romance": 0.04,
  "Sci-Fi": 0.05, "Thriller": 0.09, "War": 0.03, "Western": 0.03
}

Guidelines:
- Consider questionnaire answers as the PRIMARY factor in weight determination
- Analyze responses to infer genres that would best match the user's taste
- Use explicit genre selections as a secondary consideration
- Do NOT automatically give preferred genres the highest weights
- Ensure the weights reflect a holistic analysis with emphasis on questionnaire answers
"""
```

Figure 2: System-Prompt

5.2 Content-Based Filtering Implementation

5.2.1 Data Sources and Features

The content-based recommender uses the following components:

- **Movie Data**: Pre-processed MovieLens dataset with one-hot encoded genres
- **Genre Columns**: Binary features for each of the 19 MovieLens genres (e.g., genre_Action, genre_Comedy)
- **User Genre Weights**: LLM-generated normalized weights from questionnaire responses
- **Movie Metadata**: Title, genres, and other movie information

5.2.2 Recommendation Algorithm

The system uses a dot product approach rather than cosine similarity:

```python
def calculate_recommendations(self, user_genre_weights: pd.Series) -> pd.DataFrame:
    """Calculate recommendation scores for all movies using user's genre preferences"""
    try:
        # Movie-genre matrix
        movie_genre_matrix = self.movies_df[self.genre_columns]

        # Calculate recommendation scores using dot product
        recommendation_scores = movie_genre_matrix.dot(user_genre_weights)

        # Add scores to movies dataframe
        recommendations_df = self.movies_df.copy()
        recommendations_df['recommendation_score'] = recommendation_scores

        # Sort by recommendation score in descending order
        recommendations_df = recommendations_df.sort_values('recommendation_score', ascending=False)

        return recommendations_df
    except Exception as e:
        logger.error(f"Error calculating recommendations: {e}")
        return pd.DataFrame()
```

Figure 3: Generating content based filtering recommendations

5.3 Collaborative Filtering Implementation

5.3.1 User-Based Collaborative Filtering

The system implements user-based collaborative filtering with the following approach:

1. **Data Preparation**: Load MovieLens dataset and frontend user ratings
2. **Data Filtering**: Include movies with 100+ ratings and users with 5+ ratings
3. **Rating Normalization**: Subtract user mean from all ratings for bias removal
4. **User Similarity**: Calculate Pearson correlation between normalized ratings
5. **Neighborhood Selection**: Find users above similarity threshold
6. **Recommendation Generation**: Predict ratings using weighted similarity scores
7. **Movie Filtering**: Exclude movies already rated by target user
8. **Recommendation Ranking**: Sort by predicted rating scores and return top-N

5.3.2 Pearson Correlation Implementation

```python
def calculate_user_similarity(self, user1_ratings: Dict[int, float], user2_ratings: Dict[int, float]) -> float:
    """
    try:
        # Find common movies rated by both users
        common_movies = set(user1_ratings.keys()) & set(user2_ratings.keys())

        # Require minimum overlap for meaningful similarity
        if len(common_movies) < 4:  # User's preference: increased for better statistical validity
            return 0.0

        # Get ratings for common movies
        user1_common_ratings = [user1_ratings[movie_id] for movie_id in common_movies]
        user2_common_ratings = [user2_ratings[movie_id] for movie_id in common_movies]

        # Guard against degenerate cases: if either user's normalized ratings have (near-)zero
        # variance across common movies, Pearson correlation becomes undefined/unstable.
        # Treat such users as having uniform preferences and return a small similarity.
        user1_std = np.std(user1_common_ratings)
        user2_std = np.std(user2_common_ratings)
        eps = 1e-8
        if user1_std < eps or user2_std < eps:
            logger.debug("Near-zero variance in normalized ratings; returning small similarity (0.1)")
            return 0.1  # Small, non-zero similarity for uniform-preference users

        # For normalized data, Pearson correlation is the standard approach
        correlation, p_value = pearsonr(user1_common_ratings, user2_common_ratings)

        # Handle NaN values
        if np.isnan(correlation):
            return 0.0

        # Log debugging info for high correlations (this is now more expected with normalization)
        if correlation > 0.9:
            logger.debug(f"High correlation: {correlation:.3f} with {len(common_movies)} common movies")
            logger.debug(f"Std devs - User1: {user1_std:.3f}, User2: {user2_std:.3f}")

        return correlation

    except Exception as e:
        logger.error(f"Error calculating user similarity: {str(e)}")
        return 0.0
```

Figure 4: Pearson Correlation implmentation

5.3.3 Recommendation Generation Algorithm

After finding similar users, the system generates recommendations using weighted collaborative filtering:

```
def generate_recommendations(target_ratings, similar_users,
    n_recommendations):
    """
    Generate movie recommendations using weighted collaborative
        filtering
    """

    # Step 1: Find candidate movies
    candidate_movies = {}
    target_rated_movies = set(target_ratings.keys())

    for user_id, similarity_score in similar_users:
        user_ratings = get_user_ratings(user_id)
```

```
        # Find movies rated by similar user but not by target user
        for movie_id, rating in user_ratings:
            if movie_id not in target_rated_movies:
                candidate_movies[movie_id].append((rating,
                    similarity_score))

    # Step 2: Calculate weighted ratings for candidate movies
    movie_scores = []
    for movie_id, rating_similarity_pairs in candidate_movies:

        # Weighted average: Sum(rating * similarity) / Sum(similarity)
         weighted_sum = sum(rating * similarity for rating, similarity
            in rating_similarity_pairs)
        similarity_sum = sum(similarity for _, similarity in
           rating_similarity_pairs)

        weighted_rating = weighted_sum / similarity_sum
        movie_scores.append((movie_id, weighted_rating))

    # Step 3: Rank and select top recommendations
    movie_scores.sort(by=weighted_rating, descending=True)
    top_movies = movie_scores[:n_recommendations]

    # Step 4: Enrich with movie details (title, genres)
    recommendations = []
    for movie_id, weighted_rating in top_movies:
        movie_info = get_movie_details(movie_id)
        recommendations.append({
            'movieId': movie_id,
            'title': movie_info.title,
            'genres': movie_info.genres,
            'weighted_rating': weighted_rating
        })

    return recommendations
```

5.4 Data Integration and Management

5.4.1 User Data Management

The system maintains separate data stores for:

- **UI Users**: Frontend user registrations and preferences
- **User Responses**: Questionnaire responses and genre weights
- **Movie Ratings**: User-movie interaction data
- **MovieLens Data**: Original dataset for collaborative signals

5.4.2 Real-time Processing

- **On-demand Recommendations**: No pre-computed caching
- **Dynamic User Profiles**: Updates with new ratings

- **Adaptive Algorithms**: Switches between content-based and collaborative filtering
- **Performance Optimization**: Vectorized operations for large datasets

6 API Documentation

6.1 Core API Endpoints

The backend provides the following RESTful API endpoints:

Method	Endpoint	Description
Frontend-Integrated API Endpoints		
GET	/api/health	API health check endpoint
GET	/api/genres	Get all available MovieLens genres
GET	/api/questions	Get all 18 preference questions with options
GET	/api/frequency-options	Get watching frequency options
POST	/api/validate-genres	Validate genre selection for conflicts
POST	/api/questionnaire	Process questionnaire responses and generate LLM-based genre weights

6.2 Usage Overview

The API follows RESTful principles and returns JSON responses. The system supports both content-based and collaborative filtering recommendation approaches, with user preferences collected through an intelligent questionnaire system enhanced by LLM processing.

All endpoints accept and return JSON-formatted data, making integration straightforward for client applications. The recommendation system provides flexibility through multiple filtering algorithms while maintaining simplicity in the API interface.

6.3 API Implementation

6.3.1 FastAPI Service Architecture

The backend implements a RESTful API with the following structure:

```
# FastAPI application setup
app = FastAPI(
    title="MovieLens␣User␣Categorization␣API",
    description="API␣for␣movie␣preference␣questionnaire␣and␣genre␣weight
       ␣generation",
    version="1.0.0"
)
```

```
# Configure CORS for frontend integration
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

6.3.2 Request/Response Models

The API uses Pydantic models for data validation:

```
class QuestionnaireRequest(BaseModel):
    preferred_genres: List[str]
    disliked_genres: Optional[List[str]] = []
    q1: str  # Movie pacing preference
    q2: str  # Character depth and complexity preference
    q3: str  # Visual style and special effects importance
    q4: str  # Overall themes or moods preference
    q5: str  # Preferred setting or era
    q6: str  # Romantic subplots or themes preference
    q7: str  # Dark, tragic, or disturbing elements preference
    q8: str  # Primary focus of movie narrative
    q9: str  # Importance of humor in movies
    q10: str  # Horror movies preference
    q11: str  # Crime or mystery plots preference
    q12: str  # Science-fiction or fantasy movies interest
    q13: str  # Action, adventure, and thriller films interest
    q14: str  # War movies or Westerns preference
    q15: str  # Documentary films preference
    q16: str  # Animated movies or children's/family films preference
    q17: str  # Musicals preference
    q18: str  # IMAX or big-screen experience preference
    watching_frequency: str

class QuestionnaireResponse(BaseModel):
    success: bool
    genre_weights: Optional[Dict[str, float]] = None
    error: Optional[str] = None
    user_id: Optional[str] = None
    raw_llm_response: Optional[str] = None

class CollaborativeFilteringRequest(BaseModel):
    n_similar_users: int = Field(default=10, ge=1, le=50)
    similarity_threshold: float = Field(default=0.3, ge=0.0, le=1.0)
    n_recommendations: int = Field(default=10, ge=1, le=50)

class GenreSelectionRequest(BaseModel):
    preferred_genres: List[str]
    disliked_genres: List[str]

Content-Based Filtering Response Models endpoint returns a plain
   dictonary and not a Pydantic Object
```

6.4 API Usage Examples

6.4.1 Starting the API Server

To run the FastAPI server locally:

```
cd recommendation-system/backend/app
uvicorn app.main:app --reload --port 8000
# Server runs on http://localhost:8000
```

The API documentation is available at `http://localhost:8000/docs` (Swagger UI) and `http://localhost:8000/redoc` (ReDoc).

6.4.2 Basic Workflow

1. **Get Available Genres**: `GET /api/genres`
2. **Get Questions**: `GET /api/questions`
3. **Validate Genre Selection**: `POST /api/validate-genres`
4. **Submit Questionnaire**: `POST /api/questionnaire`
5. **Get Content-based Recommendations**: `GET /api/content-based-recommendations`
6. **Get Movies for Rating**: `GET /api/movies-for-rating`
7. **Submit Ratings**: `POST /api/submit-ratings`
8. **Get Collaborative Recommendations**: `POST /api/collaborative-filtering`

6.4.3 Example Request/Response

Submit Questionnaire Example:

```
POST /api/questionnaire
Content-Type: application/json

{
  "preferred_genres": ["Action", "Sci-Fi", "Thriller"],
  "disliked_genres": ["Romance", "Musical"],
  "q1": "Fast-paced, action-packed with constant movement.",
  "q2": "Well-defined characters with some complexity.",
  "q3": "Highly important     I value stunning visuals and effects.",
  "q4": "Exciting, adventurous, and thrilling.",
  "q5": "Futuristic or science-fiction settings.",
  "q6": "I prefer minimal romance     a small side story at most.",
  "q7": "I'm comfortable with dark or disturbing content.",
  "q8": "Action and excitement     I enjoy fights, chases, set pieces.",
  "q9": "Nice to have     I enjoy occasional comic relief.",
  "q10": "I'm okay with mild suspense or thriller elements.",
  "q11": "Sometimes     I enjoy a good mystery or crime thriller.",
  "q12": "Very interested     I actively seek out sci-fi and fantasy.",
  "q13": "High     Adrenaline-pumping action movies are favorites.",
  "q14": "I don't particularly enjoy either war movies or Westerns.",
  "q15": "Rarely     I might watch occasionally, prefer fiction.",
  "q16": "Not usually     I prefer live-action adult-oriented films.",
  "q17": "I dislike musicals     singing usually turns me off.",
```

```
  "q18": "Absolutely     I try to watch blockbusters in IMAX.",
  "watching_frequency": "Frequently (multiple times per week)"
}
```

Response:


```
{
  "success": true,
  "genre_weights": {
    "Action": 0.18,
    "Sci-Fi": 0.15,
    "Thriller": 0.14,
    "Adventure": 0.12,
    "Crime": 0.08,
    "Fantasy": 0.07,
    "Drama": 0.06,
    "Mystery": 0.05,
    "Horror": 0.04,
    "IMAX": 0.04,
    "War": 0.02,
    "Comedy": 0.02,
    "Documentary": 0.01,
    "Film-Noir": 0.01,
    "Animation": 0.01,
    "Children": 0.0,
    "Musical": 0.0,
    "Romance": 0.0,
    "Western": 0.0
  },
  "user_id": "a1b2c3d4-e5f6-7890-1234-567890abcdef"
}
```


7 References

References

[1] Python Software Foundation. Python Documentation. Available at: `https://docs.python.org/`. Accessed: 2024.

[2] OpenAI. OpenAI API Documentation. Available at: `https://platform.openai.com/docs`. Accessed: 2024.

[3] Sebastián Ramirez. FastAPI Documentation. Available at: `https://fastapi.tiangolo.com/`. Accessed: 2024.

[4] F. Maxwell Harper and Joseph A. Konstan. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems, 5(4):1-19, 2015.