Desing Document

- Table of Contents

1. List of Contributors

2. Task Matrix

3. Design Document

1. System Overview

2. Implementation Details

3. Use Case Support in Design

4. Design Decisions

5. GitHub Commit Requirement

- List of Contributors

1. İbrahim Eren Yılmaz

2. Sencer Ali Şahin

3. Mete Oktar

4. Efe Arda Uzunova

- Task Matrix

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Task | İbrahim Eren Yılmaz | Sencer Ali Şahin | Mete Oktar | Efe Arda Uzunova |
| System Architecture & Technology Stack |  |  |  |  |
| Implementation (UI + Suggestor Classes) |  |  |  |  |
| Use Case Descriptions & Diagrams |  |  |  |  |
| Design Decisions & Comparisons |  |  |  |  |
| QA Strategy & Test Plan |  |  |  |  |
| Final Integration & Documentation |  |  |  |  |

**1. System Overview**

**Brief Project Description**

Our project is a **movie recommendation system** that allows the user to:

• Input and rate several movies (four, in our example).

• Specify a fifth movie for which similar recommendations are desired.

• Generate an Excel file containing recommended films (based on content + user preference).

The primary goal is to **offer personalized movie recommendations** using both **content-based** and **collaborative filtering** elements (via an SVD model).

**System Architecture**

We utilize a **layered architecture**:

• **UI Layer**: A Tkinter-based GUI (in UI.py) for user inputs.

• **Core Logic / Recommender Layer**: The Suggestor class handles data loading, preprocessing, content-similarity computation, and the SVD-based collaborative filtering logic.

• **Data Layer**: CSV files and pickled DataFrames (.pkl) provide data about movies (metadata, credits, keywords) and user ratings.

UI.py

(Tkinter GUI)

CSV & PKL Data Files

(movies, ratings, etc.)

Suggestor.py

(Hybrid Logic)

**Technology Stack**

1. **Python 3.x**

2. **Tkinter** for the GUI.

3. **Pandas** for data manipulation.

4. **Scikit-Learn** (CountVectorizer, cosine\_similarity).

5. **Surprise** library (for SVD collaborative filtering).

6. **CSV/Excel** file operations (basic I/O).

**2. Implementation Details**

**Codebase Structure**

├── archive/

│ ├── credits.csv

│ ├── keywords.csv

│ ├── links\_small.csv

│ ├── movies\_metadata.csv

│ ├── ratings\_small.csv

│ ├── smd.pkl

│ ├── ratings.pkl

│ └── userRatings.csv

├── UI.py

├── Suggestor.py

└── Main.py (or equivalent entry point)

• **UI.py**: Defines the UI class (Tkinter-based) and a run\_ui() function that launches the window, collects user input, and returns it in a dictionary.

• **Suggestor.py**: Defines the Suggestor class, which loads data, processes it, applies the content-based filtering (CountVectorizer + cosine similarity), and also trains an SVD model to personalize recommendations.

• **Main.py** (example name): Ties everything together. We import run\_ui and Suggestor, then run them. After the user input is obtained, we get a final list of recommended movies and export it to an Excel file.

**Key Implementations**

1. **Data Loading**

• Checks if pickled data (smd.pkl, ratings.pkl) exists. Otherwise, it reads from CSV, merges relevant columns, converts data types, etc.

• The code merges **credits**, **keywords**, and **movies\_metadata** on the id.

2. **Content-Based Filtering**

• **CountVectorizer** is applied to a “soup” column, which is a combination of keywords, cast, director, and genres.

• We compute **cosine similarity** over this matrix to get top similar movies.

3. **Collaborative Filtering (SVD)**

• We create a small user rating file (userRatings.csv) using the user’s input from the GUI.

• We load it with **Surprise**’s Dataset.load\_from\_df, build a training set, and fit an SVD model.

• For each candidate movie from the content-based similarity, we predict the user’s rating and sort by the predicted score.

4. **Hybrid Recommendation**

• The hybrid(userId, title) method in Suggestor first retrieves the top-N similar movies via cosine similarity.

• It then re-ranks these movies by the SVD-predicted rating to incorporate the user’s preference.

**Component Interfaces**

* **UI Class (from UI.py)**

class UI:

def \_\_init\_\_(self):

# Creates the Tkinter window and frames

# Collects up to 4 rated films + 1 target film

# ...

def submit\_data(self):

# Gathers all user inputs (film titles + ratings)

# Stores them in self.user\_data

# Destroys the window

• **run\_ui()** function: Launches the UI, returns a dictionary with keys like {"input\_1": "Gladiator", "rating\_1": 5, ...}.

* **Suggestor Class (from Suggestor.py)**

class Suggestor:

def \_\_init\_\_(self, input):

# Loads CSV/PKL data or merges from scratch

# Sets up CountVectorizer and obtains cosine\_sim

# Trains SVD model using user CSV data

def get\_films\_alike(self):

# Returns the title that the user wants recommendations for

def hybrid(self, userId, title):

# 1) Retrieves top N similar movies from cosine similarity

# 2) Uses SVD to predict rating for each, sorts descending

# 3) Returns top recommendations

• Primary methods include:

• **Data cleaning** (clean\_data, get\_list, get\_director, etc.).

• **create\_soup**: Creates the combined text used for CountVectorizer.

• **hybrid**: Hybrid recommendation logic.

**Visual Interfaces**

The user interface is quite simple—a **Tkinter** form with:

• Four rows, each collecting a movie title and a rating (0–5).

• A fifth row to collect a “target” movie for which the user wants similar recommendations.

• A **Submit** button that triggers submit\_data().

**3. Use Case Support in Design**

We selected **4 important use cases** that align with the functional requirements of a basic recommendation system:

1. **UC1: User inputs and rates a set of known movies**

• The user opens the application, sees four fields for movie titles and ratings, and provides them.

• *Requirement:* System must collect user preferences.

2. **UC2: User specifies a target movie**

• The user enters a fifth movie title for which they want recommendations.

• *Requirement:* System must accept a “target” film.

3. **UC3: System calculates recommended movies**

• On submit, the system merges content-based and collaborative filtering approaches.

• *Requirement:* Provide personalized recommendations based on user taste and movie similarity.

4. **UC4: System outputs the recommendation list**

• The system displays or exports the recommended titles, e.g., to Excel (deneme\_gelismis.xlsx).

• *Requirement:* Provide an accessible recommendation result file or direct output.

**Requirement Mapping**

|  |  |
| --- | --- |
| **Functional Requirement** | **Use Case Supported** |
| Allow user to enter rated movies | UC1 |
| Permit user to select a “target” movie for similarity | UC2 |
| Combine user ratings with content-based similarity | UC3 |
| Output recommendations in a structured format | UC4 |

**How the System Architecture Supports Each Use Case**

• **UC1 & UC2**: The **UI Layer** (UI.py) directly addresses these, providing text boxes and combo boxes for ratings.

• **UC3**: The **Recommender Layer** (Suggestor) has the logic for combining SVD predictions with cosine similarity.

• **UC4**: **Data Layer** writes output to Excel or CSV, fulfilling the final user-facing requirement.

**4. Design Decisions**

**Technology Comparisons**

1. **GUI Framework Choice**

• **Tkinter** vs. PyQt/Pyside.

• Chosen **Tkinter** because it is built into Python’s standard library and is simpler for small-scale forms.

2. **Recommendation Algorithm**

• **Content-Based (TF-IDF or CountVectorizer)** vs. purely Collaborative approaches.

• Chosen a **hybrid** approach because combining user preference with content similarity often yields better accuracy with fewer user ratings.

3. **Machine Learning Library**

• **Surprise** vs. building an SVD from scratch (e.g., scikit-learn’s truncated SVD or an ALS approach).

• Chosen **Surprise** for its simplicity and direct rating-prediction APIs.

**Decision Justifications**

• **Tkinter** fits quick prototyping; no extra dependencies needed.

• **Hybrid approach** addresses the “cold start” problem for the user’s short rating list.

• **Surprise** drastically simplifies building and training an SVD model for collaborative filtering.

**5. GitHub Commit Requirement**

All code (UI, Suggestor, test scripts, readme files, etc.) must be committed to GitHub.

• We recommend separate commits, with clear messages like “Implement UI for user input (#3)” or “Add SVD training logic (#7).”

• The alternative code comparisons (e.g., different ML approaches) should also be committed, possibly on separate branches or shown in example code