Movie Recommendation System - Complete Project Documentation

Project Name: CinemaScope - AI-Powered Movie Recommendation System

Course: CSET240 - Probability

Technology Stack: Python, Streamlit, K-Nearest Neighbors (KNN), BeautifulSoup, OMDB API

Date: November 2025

Table of Contents

---

1. Project Overview

1.1 Introduction

CinemaScope is an intelligent movie recommendation system that uses machine learning algorithms to suggest movies based on user preferences. The system leverages the K-Nearest Neighbors (KNN) algorithm to find similar movies based on genre preferences and IMDB ratings.

1.2 Key Features

- **Movie-Based Recommendations:** Get recommendations similar to a movie you love
- **Genre-Based Recommendations:** Discover movies based on your favorite genres

- **High-Quality Movie Posters:** Automatic fetching and display of movie posters
- **Real-Time IMDB Ratings:** Display accurate ratings for all recommendations
- **Detailed Movie Information:** Plot summaries, cast information, and movie details
- **Modern UI/UX:** Dark cinematic theme with gold accents and smooth animations
- **Dual Data Sources:** OMDB API integration with IMDB scraping fallback

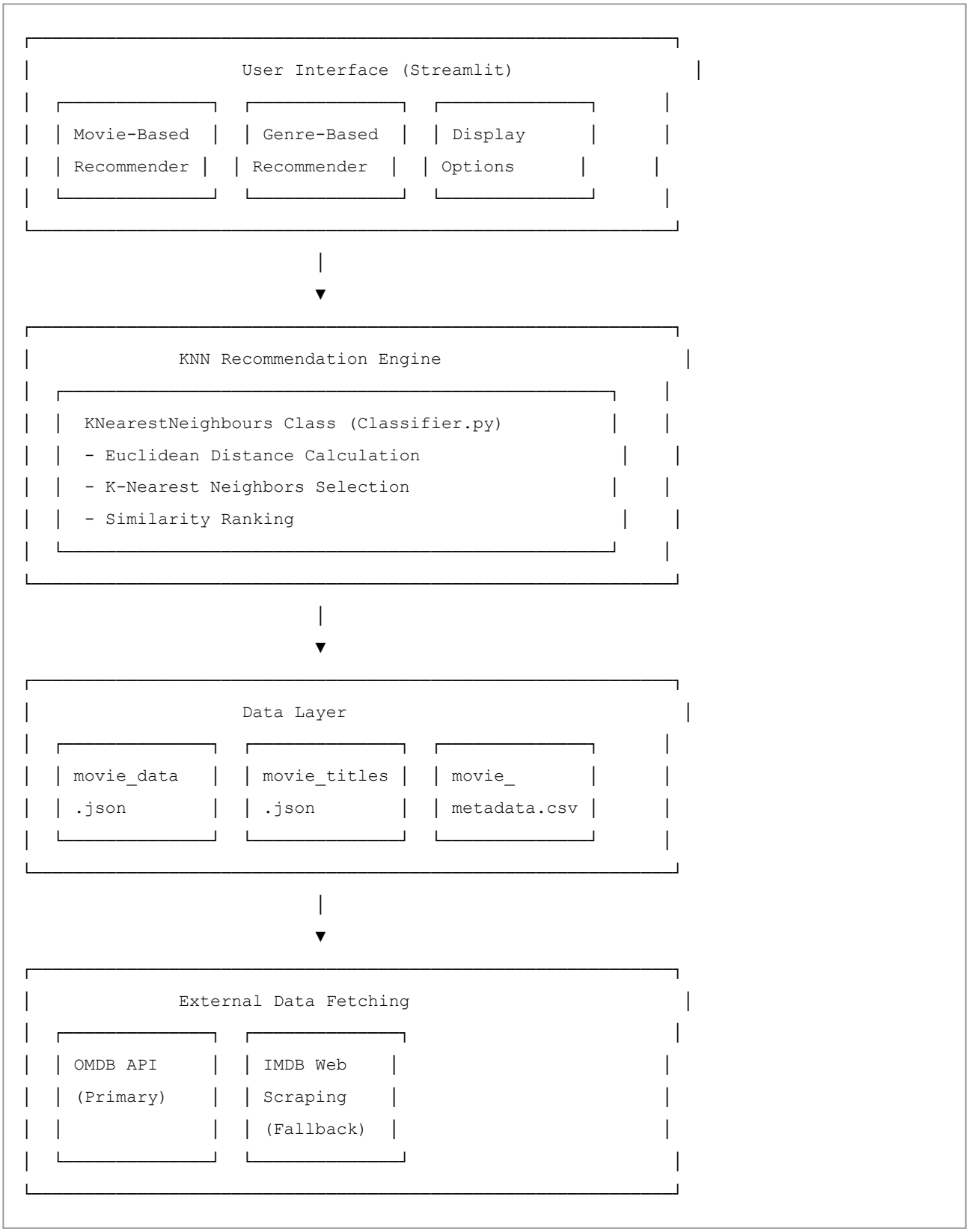
1.3 Technology Stack

- **Frontend Framework:** Streamlit (Python web framework)
- **Machine Learning:** K-Nearest Neighbors (KNN) algorithm
- **Data Processing:** Pandas, NumPy
- **Web Scraping:** BeautifulSoup4, Requests
- **Image Processing:** PIL/Pillow
- **Data Storage:** JSON files
- **External APIs:** OMDB API (optional)

---

2. System Architecture

2.1 High-Level Architecture



2.2 Data Flow

1. **User Input:** User selects a movie or genres
2. **Feature Vector Creation:** System creates a feature vector based on genres and IMDB score
3. **KNN Processing:** KNN algorithm finds K most similar movies
4. **Data Enrichment:** System fetches posters and details from OMDB/IMDB

5. **Display:** Results are displayed in formatted movie cards

---

3. Directory Structure

```
Movie_Recommender_Improved/
│
├── App.py                          # Main Streamlit application
├── Classifier.py                   # KNN algorithm implementation
├── Movie_Data_Processing.ipynb     # Data preprocessing notebook
├── OMDB_API_SETUP.md               # OMDB API setup instructions
├── PROJECT_DOCUMENTATION.md        # This documentation file
│
├── Data/
│   ├── movie_data.json             # Processed movie feature vectors
│   ├── movie_titles.json           # Movie titles and IMDB links
│   └── movie_metadata.csv          # Raw movie dataset
│
├── meta/
│   └── logo.jpg                    # Application logo
│
└── __pycache__/                    # Python cache files
```

3.1 File Descriptions

App.py

The main application file containing:

- Streamlit UI components
- Movie recommendation logic
- Poster and information fetching functions
- UI styling and theming
- User interaction handlers

Classifier.py

Contains the KNearestNeighbours class:

- Euclidean distance calculation
- KNN algorithm implementation
- Similarity computation

Movie_Data_Processing.ipynb

Jupyter notebook for data preprocessing:

- Loading raw CSV data
- Extracting genres
- Creating feature vectors
- Generating JSON data files

Data Files

- **movie_data.json:** Contains feature vectors for each movie (genre flags + IMDB score)
- **movie_titles.json:** Contains movie titles, indices, and IMDB links
- **movie_metadata.csv:** Original dataset with 5043 movies and 28 features

---

4. Data Processing Pipeline

4.1 Data Source

The project uses the **IMDB 5000 Movie Dataset**, which contains:

- 5,043 movies
- 28 features including genres, ratings, directors, actors, etc.
- IMDB links for each movie

4.2 Data Preprocessing Steps

Step 1: Load Raw Data

```
data = pd.read_csv('./Data/movie_metadata.csv')
df = data[['genres', 'movie_title', 'imdb_score', 'movie_imdb_link']].copy()
```

Step 2: Extract Genres

```
genres_all_movies = [df.loc[i]['genres'].split('|') for i in df.index]
genres = sorted(list(set([item for sublist in genres_all_movies for item in sublist])))
# Result: 26 unique genres
```

Step 3: Create Feature Vectors

For each movie:

1. Create a binary vector for genres (1 if movie has genre, 0 otherwise)
2. Append IMDB score to the vector
3. Result: Vector of length 27 (26 genres + 1 IMDB score)

Example:

```
# Movie: "Avatar" with genres: Action, Adventure, Fantasy, Sci-Fi
# IMDB Score: 7.9
# Feature Vector: [1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, ..., 0, 0, 1, 0, ..., 7.9]
```

Step 4: Generate JSON Files

```
# movie_data.json: List of feature vectors
# movie_titles.json: List of tuples (title, index, IMDB_link)
```

4.3 Data Structure

movie_data.json:

```
[
  [1, 1, 0, 0, ..., 7.9],  // Movie 0 feature vector
  [1, 1, 0, 0, ..., 7.1],  // Movie 1 feature vector
  ...
]
```

movie_titles.json:

```
[
  ["Avatar", 0, "http://www.imdb.com/title/tt0499549/"],
  ["Pirates of the Caribbean: At World's End", 1, "http://..."],
  ...
]
```

---

5. Core Components

5.1 KNearestNeighbours Class (Classifier.py)

Class Structure

```
class KNearestNeighbours:
    def __init__(self, data, target, test_point, k):
        self.data = data               # Training data (all movies)
        self.target = target           # Target labels (not used in this implementation)
        self.test_point = test_point    # Query movie feature vector
        self.k = k                     # Number of neighbors
        self.distances = []            # Calculated distances
        self.indices = []             # Indices of K nearest neighbors
```

Key Methods

1. Euclidean Distance Calculation

```
@staticmethod
def dist(p1, p2):
    """Calculate Euclidean distance between two points"""
    return np.linalg.norm(np.array(p1) - np.array(p2))
```

2. KNN Algorithm

```
def fit(self):
    # Calculate distances to all movies
    self.distances = [(self.dist(self.test_point, point), i)
                      for point, i in zip(self.data, range(len(self.data)))]

    # Sort by distance
    sorted_li = sorted(self.distances, key=itemgetter(0))

    # Get K nearest neighbors
    self.indices = [index for (val, index) in sorted_li[:self.k]]
```

5.2 Recommendation Functions (App.py)

KNN_Movie_Recommender

```
def KNN_Movie_Recommender(test_point, k):
    """Generate movie recommendations using KNN algorithm"""
    target = [0 for item in movie_titles]
    model = KNearestNeighbours(data, target, test_point, k=k)
    model.fit()
    table = []
    for i in model.indices:
        table.append([movie_titles[i][0],      # Movie title
                      movie_titles[i][2],      # IMDB link
                      data[i][-1]])            # IMDB rating
    return table
```

Movie-Based Recommendation

1. User selects a movie
2. Extract feature vector for selected movie
3. Find K+1 nearest neighbors (K+1 to exclude the selected movie itself)
4. Remove the selected movie from results
5. Return top K recommendations

Genre-Based Recommendation

1. User selects genres and minimum IMDB score
2. Create feature vector: [1 for selected genres, 0 for others, IMDB_score]
3. Find K nearest neighbors
4. Return recommendations

---

6. User Interface

6.1 Design Theme

The application features a **dark cinematic theme** with:

- **Color Scheme:**
  - Background: Dark blue gradient (#0f0c29 → #302b63 → #24243e)
  - Accent: Gold (#ffd700) with gradient effects
  - Text: White with varying opacity
  - Cards: Dark with gold borders
- **Typography:**
  - Headers: Playfair Display (serif, bold)
  - Body: Inter (sans-serif, modern)

- **Effects:**
    - Hover animations
    - Gradient backgrounds
    - Box shadows
    - Smooth transitions

6.2 Main Components

Header Section

- Application title: "CinemaScope"
- Subtitle: "Discover Your Next Favorite Movie with AI-Powered Recommendations"
- Animated gradient background

Sidebar

- Dataset statistics (total movies)
- Key features list
- OMDB API status indicator

Main Content Area

- Recommendation type selector
- Input controls (movie selector / genre selector)
- Display options (Show Posters / Text Only)
- Recommendation results

Movie Cards

Each recommendation displays:

- Movie poster (if enabled)
- Movie title with ranking
- Plot summary
- Cast information
- IMDB rating badge
- Link to IMDB page

6.3 User Interaction Flow

```
1. User selects recommendation type
   ├── Movie-based
   │   ├── Select movie from dropdown
   │   ├── Choose display option
   │   ├── Set number of recommendations
   │   └── Click "Get Recommendations"
   │
   └── Genre-based
       ├── Select one or more genres
       ├── Set minimum IMDB score
       ├── Choose display option
       ├── Set number of recommendations
       └── Click "Get Recommendations"

2. System processes request
   ├── Creates feature vector
   ├── Runs KNN algorithm
   └── Fetches movie details

3. Results displayed
   ├── Progress bar shows loading
   ├── Movie cards appear one by one
   └── Posters and details load asynchronously
```

---

7. Algorithm Implementation

7.1 K-Nearest Neighbors (KNN) Algorithm

Algorithm Overview

KNN is a non-parametric, instance-based learning algorithm used for classification and regression. In this project, it's used for finding similar movies.

How It Works

1. **Feature Representation:**
   - Each movie is represented as a 27-dimensional vector
   - 26 dimensions for genres (binary: 1 or 0)
   - 1 dimension for IMDB score (continuous: 0-10)
2. **Distance Calculation:**
   - Euclidean distance between query movie and all movies in dataset

- Formula: `√(Σ(xi - yi)²)`
- Lower distance = more similar movies

3. **Neighbor Selection:**

   - Sort all movies by distance
   - Select K movies with smallest distances
   - These are the recommendations

Example Calculation

Query Movie: "Avatar"

- Genres: Action, Adventure, Fantasy, Sci-Fi
- IMDB Score: 7.9
- Feature Vector: `[1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, ..., 0, 0, 1, 0, ..., 7.9]`

Comparison Movie: "Pirates of the Caribbean"

- Genres: Action, Adventure, Fantasy
- IMDB Score: 7.1
- Feature Vector: `[1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, ..., 0, 0, 0, 0, ..., 7.1]`

Distance Calculation:

```
Distance = √[(1-1)² + (1-1)² + (0-0)² + ... + (1-0)² + ... + (7.9-7.1)²]
         = √[0 + 0 + 0 + ... + 1 + ... + 0.64]
         = Small distance → Similar movies
```

7.2 Why KNN for Movie Recommendations?

Advantages:

- Simple to understand and implement
- No training phase required
- Works well with high-dimensional data
- Handles multi-label classification (multiple genres)
- Interpretable results

Limitations:

- Computationally expensive for large datasets (O(n) for each query)
- Sensitive to irrelevant features
- Requires feature scaling (handled by binary encoding)

---

8. Data Fetching Mechanisms

8.1 Movie Poster Fetching

The system uses a **multi-tier approach** for fetching movie posters:

Tier 1: OMDB API (Primary)

```
def fetch_from_omdb(imdb_id=None, movie_title=None):
    # Uses OMDB API to fetch poster URL
    # Returns PIL Image object
```

Advantages:

- Reliable and fast
- High-quality images
- Consistent format

Tier 2: IMDB Scraping (Fallback)

Multiple methods tried in order:

1. **JSON-LD Structured Data**
    - Extracts poster from `<script type="application/ld+json">`
    - Most reliable scraping method
2. **Open Graph Meta Tags**
    - Extracts from `<meta property="og:image">`
    - Standard for social media previews
3. **IMDB HTML Selectors**
    - Multiple CSS selectors tried:
        - `div.ipc-media--poster-27x40`
        - `div.poster`
        - `img[data-testid="hero-poster"]`
4. **Regex Pattern Matching**
    - Searches HTML for image URLs
    - Filters for poster-related URLs

8.2 Movie Information Fetching

Similar multi-tier approach for fetching:

- Plot/Story
- Cast information
- Movie title

OMDB API Response Structure

```
{
  "Title": "Avatar",
  "Plot": "A paraplegic marine...",
  "Actors": "Sam Worthington, Zoe Saldana, Sigourney Weaver",
  "Poster": "https://m.media-amazon.com/images/...",
  "imdbRating": "7.9"
}
```

IMDB Scraping Methods

1. JSON-LD structured data
2. Meta description tags
3. Plot summary selectors
4. Cast list extraction
5. Title extraction from multiple sources

8.3 Caching Strategy

Both poster and info fetching use Streamlit's caching:

```
@st.cache_data(ttl=1800, show_spinner=False)  # Cache for 30 minutes
def movie_poster_fetcher(imdb_link, movie_title=None):
    # Function implementation
```

Benefits:

- Reduces API calls
- Faster subsequent loads
- Better user experience

---

9. Installation & Setup

9.1 Prerequisites

- Python 3.7 or higher
- pip (Python package manager)

9.2 Required Packages

Install dependencies:

```
pip install streamlit
pip install pandas
pip install numpy
pip install beautifulsoup4
pip install requests
pip install pillow
```

Or use requirements.txt (if available):

```
pip install -r requirements.txt
```

9.3 Data Files

Ensure the following files exist in the `Data/` directory:

- `movie_data.json`
- `movie_titles.json`
- `movie_metadata.csv` (optional, for reference)

9.4 OMDB API Setup (Optional)

1. Visit http://www.omdbapi.com/apikey.aspx (http://www.omdbapi.com/apikey.aspx)
2. Sign up for free tier (1,000 requests/day)
3. Set environment variable:

```
# Windows PowerShell
$env:OMDB_API_KEY="your_api_key_here"

# Windows CMD
set OMDB_API_KEY=your_api_key_here

# Linux/Mac
export OMDB_API_KEY="your_api_key_here"
```

9.5 Running the Application

```
streamlit run App.py
```

The application will open in your default browser at `http://localhost:8501`

---

10. Usage Guide

10.1 Movie-Based Recommendations

1. **Select Recommendation Type:** Choose "Movie based"
2. **Select a Movie:** Choose from the dropdown (e.g., "Avatar")
3. **Display Options:**
    - "Show Posters" - Displays movie posters (slower)
    - "Text Only" - Faster, no posters
4. **Number of Recommendations:** Use slider (5-20)
5. **Click:** "□ Get Recommendations"
6. **View Results:** Scroll through recommended movies

10.2 Genre-Based Recommendations

1. **Select Recommendation Type:** Choose "Genre based"
2. **Select Genres:** Choose one or more genres (e.g., Action, Sci-Fi)
3. **Set Minimum IMDB Score:** Use slider (1-10, default: 8)
4. **Display Options:** Choose poster display preference
5. **Number of Recommendations:** Set desired count
6. **Click:** "□ Get Recommendations"
7. **View Results:** Movies matching your preferences

10.3 Understanding Results

Each movie card shows:

- **Ranking:** #1, #2, #3, etc.
- **Title:** Movie name
- **Plot:** Story summary (if available)
- **Cast:** Main actors (if available)
- **IMDB Rating:** Star rating badge
- **Poster:** Movie poster image (if enabled)
- **IMDB Link:** Click to view full details

10.4 Tips for Best Results

- **For Movie-Based:** Select movies you genuinely enjoyed
- **For Genre-Based:** Select 2-4 genres for best results
- **IMDB Score:** Higher minimum score = fewer but better quality results
- **Posters:** Enable only if you have good internet connection
- **Number of Recommendations:** 10-15 is optimal for variety

---

11. Technical Details

11.1 Performance Considerations

Optimizations:

- Data caching with `@st.cache_data`
- Lazy loading of posters
- Progress bars for user feedback
- Efficient distance calculations using NumPy

Bottlenecks:

- Web scraping can be slow (mitigated by caching)
- Large dataset processing (5,043 movies)
- Image downloading and resizing

11.2 Error Handling

The system includes comprehensive error handling:

- Network timeouts (15 seconds)
- Missing data fallbacks
- API failures (graceful degradation)
- Image loading errors (placeholder display)

11.3 Code Quality

Best Practices:

- Function modularity
- Clear naming conventions
- Comprehensive comments
- Error handling
- Code reusability

11.4 Security Considerations

- User input sanitization
- Safe HTML rendering
- API key protection (environment variables)
- Request headers (User-Agent) for scraping

---

12. Future Enhancements

12.1 Potential Improvements

1. **Algorithm Enhancements:**
    - Collaborative filtering

    - Content-based filtering hybrid
    - Deep learning models
    - Matrix factorization

2. **Feature Additions:**

    - User accounts and preferences
    - Watchlist functionality
    - Rating system
    - Review integration
    - Trailer embedding

3. **Performance:**

    - Database integration (PostgreSQL/MongoDB)
    - Redis caching
    - Async processing
    - CDN for images

4. **UI/UX:**

    - Dark/light theme toggle
    - Advanced filtering options
    - Sort and filter results
    - Export recommendations

5. **Data:**

    - Real-time data updates
    - More data sources (TMDB, Rotten Tomatoes)
    - User-generated content
    - Social features

12.2 Scalability

Current limitations:

- In-memory data storage
- Synchronous processing
- Single-user design

Scalability solutions:

- Database backend
- Microservices architecture
- Load balancing
- Distributed computing

Appendix A: Key Functions Reference

A.1 App.py Functions

load_data()

- Loads movie data from JSON files
- Cached for performance
- Returns: (data, movie_titles)

extract_imdb_id(imdb_link)

- Extracts IMDB ID from URL
- Returns: IMDB ID string (e.g., "tt0499549")

fetch_from_omdb(imdb_id, movie_title)

- Fetches movie data from OMDB API
- Returns: JSON data or None

movie_poster_fetcher(imdb_link, movie_title)

- Fetches movie poster image
- Returns: PIL Image object or None

get_movie_info(imdb_link, movie_title)

- Fetches movie information (plot, cast, title)
- Returns: (title, cast, story, rating)

KNN_Movie_Recommender(test_point, k)

- Generates recommendations using KNN
- Returns: List of [title, link, rating] tuples

display_movie_card(movie, link, ratings, index, show_poster)

- Displays formatted movie card
- Handles poster display and information

clean_text(text)

- Cleans and formats text for display
- Returns: Cleaned text string

A.2 Classifier.py Methods

KNearestNeighbours.init(data, target, test_point, k)

- Initializes KNN model

KNearestNeighbours.dist(p1, p2)

- Calculates Euclidean distance
- Static method

KNearestNeighbours.fit()

- Executes KNN algorithm
- Finds K nearest neighbors

Appendix B: Data Format Specifications

B.1 movie_data.json Format

```
[
  [1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 7.9],
  [1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7.1],
  ...
]
```

Structure:

- Array of arrays
- Each inner array: 27 elements
- First 26: Genre flags (0 or 1)
- Last element: IMDB score (float)

B.2 movie_titles.json Format

```
[
  ["Avatar", 0, "http://www.imdb.com/title/tt0499549/"],
  ["Pirates of the Caribbean: At World's End", 1, "http://www.imdb.com/title/tt0449088/"],
  ...
]
```

Structure:

- Array of arrays
- Each inner array: [title (string), index (int), IMDB_link (string)]

B.3 Genre List (26 Genres)

1. Action
2. Adventure

3. Animation
4. Biography
5. Comedy
6. Crime
7. Documentary
8. Drama
9. Family
10. Fantasy
11. Film-Noir
12. Game-Show
13. History
14. Horror
15. Music
16. Musical
17. Mystery
18. News
19. Reality-TV
20. Romance
21. Sci-Fi
22. Short
23. Sport
24. Thriller
25. War
26. Western

Appendix C: Troubleshooting

C.1 Common Issues

Issue: Posters not showing

- **Solution:** Check internet connection, verify OMDB API key, wait for images to load

Issue: Slow recommendations

- **Solution:** Disable posters, reduce number of recommendations, check data file sizes

Issue: No recommendations

- **Solution:** Verify data files exist, check for errors in console, try different input

Issue: API errors

- **Solution:** Verify API key, check rate limits, ensure internet connection

C.2 Debug Mode

Enable debug output by modifying functions to print:

```
print(f"Debug: Fetched {len(table)} recommendations")
print(f"Debug: Poster URL: {poster_url}")
```

Conclusion

The CinemaScope Movie Recommendation System demonstrates a complete implementation of a machine learning-based recommendation system using the K-Nearest Neighbors algorithm. The project showcases:

- **Machine Learning:** KNN algorithm for similarity-based recommendations
- **Web Development:** Modern Streamlit-based user interface
- **Data Processing:** Efficient data preprocessing and storage
- **API Integration:** OMDB API with IMDB scraping fallback
- **User Experience:** Beautiful, responsive UI with rich content

The system successfully provides personalized movie recommendations based on user preferences, combining algorithmic precision with an intuitive user interface.

Document Version: 1.0
Last Updated: November 2025
Author: CSET240 Project Team

This documentation covers all aspects of the Movie Recommendation System project from data processing to user interface, providing a comprehensive guide for understanding, using, and extending the system.