Movie Recommendation

This project explores the relationship between user preferences and personalized movie suggestions. In an era overflowing with digital content, it is often unclear which techniques yield the most accurate and engaging recommendations. This project implements a complete machine learning pipeline to analyze a “Movie Recommendation Dataset” and automatically suggest films tailored to user tastes. We performed extensive Exploratory Data Analysis (EDA) to examine patterns in genres, ratings, and user interactions, identifying clear correlations between viewing history and future preferences. To enhance model performance, we engineered features such as genre vectors, similarity scores, and popularity-based attributes. We then trained and compared three recommendation approaches (Content-Based Filtering, Collaborative Filtering, and a Hybrid Model) to generate personalized movie predictions. The findings highlight that collaborative filtering combined with similarity-based ranking provides the most effective results. This report presents our data preprocessing steps, model design, evaluation metrics, results, and insights gained throughout the recommendation system development workflow.

Team Members

• Ayush (524110033)

• Rahul Gupta(524110004)

1. Introduction

1.1. Problem Statement

With a rapidly growing volume of content, users often struggle to find movies that match their personal tastes. Traditional search and filter methods are insufficient for personalization. This project aims to move beyond simple genre-based suggestions and develop a data-driven recommendation system that accurately predicts the movies a user may prefer using machine learning models.

1.2. Project Scope and Objectives

The primary goal is to create a complete machine learning pipeline capable of generating reliable movie recommendations based on historical rating data.

Our objectives were:

- To conduct a detailed Exploratory Data Analysis (EDA) of the MovieLens dataset or a similar movie dataset.
- To build a data processing pipeline that cleans the data and engineers new, relevant features from time-based columns.
- To implement and evaluate several regression models (Linear Regression, Random Forest, K-Nearest Neighbors) to predict the Productivity Score (1-10).
- To use hypothesis testing to validate key relationships observed during the EDA.
- To derive actionable conclusions about which habits are most predictive of productivity.

1.3. Dataset

We used the **TMDB 5000 Movies and Credits Dataset**, which contains approximately **5,000 movie entries** along with metadata such as cast, crew, genres, and popularity metrics. Each row represents a single movie.

Key data columns include:

- **Title, Overview, Release Date** (object/string)
- **Genres, Keywords, Cast, Crew** (JSON-like/object fields)
- **Runtime, Budget, Revenue** (numeric)
- **Popularity, Vote Count, Vote Average** (float/integer)
- Additional fields such as **production companies, language, and status**

This dataset provides a rich foundation for building both content-based and collaborative recommendation models.

---

2. Data Collection & Exploratory Data Analysis (EDA)

Our EDA was conducted in the `ml_movies.ipynb` notebook. The initial attempt to load the `tmdb_5000_credits.csv` file resulted in a parsing error, but the issue was resolved by using `engine='python'` and `on_bad_lines='skip'`. The movie and credits datasets were then successfully merged using an inner join on the `id` and `movie_id` columns, producing a final merged dataset (`merged_df`) containing **3975 rows and 24 columns**.

Missing values were detected in the `homepage`, `overview`, `runtime`, and `tagline` columns. The `homepage` and `tagline` columns were dropped due to limited relevance, while the single missing values in `overview` and `runtime` were imputed using their respective modes. After preprocessing, the dataset contained **zero missing values across all features**, ensuring data quality for modeling.

Several categorical columns (`genres`, `keywords`, `production_companies`, `spoken_languages`, `cast`, `crew`) originally stored as JSON-style strings were successfully parsed into list formats. The `original_language` and `status` columns were then one-hot encoded to convert them into numeric features. A cleaned subset of relevant features was stored in a new dataframe, `selected_features_df`, and later exported as `preprocessed_movie_data.csv` for model development.

Insights & Next Steps

Parsed list-based features (such as genres, keywords, and cast) will require additional transformations like tokenization, embedding, or frequency encoding before they can be directly used in similarity-based or predictive models. The preprocessed dataset is now ready for downstream tasks such as building the movie recommendation system or applying regression models to predict performance metrics like movie popularity or revenue.

2.1. Key EDA Findings

Our analysis of the merged TMDB dataset (3975 entries) revealed several important insights related to movie popularity.

Budget & Revenue Relationship

The scatter plot analysis and correlation test revealed a strong positive association between a movie's budget and its revenue.
Pearson Correlation (r): 0.74
P-value: 0.000
Since the p-value is well below 0.05, we reject the null hypothesis and conclude the correlation is statistically significant. Higher-budget films tend to generate higher revenue, confirming a measurable financial advantage.

Popularity is Influenced by Vote Count

A box plot and regression line between **vote_count** and **popularity** demonstrated that movies with higher public engagement are consistently ranked as more popular.
Pearson Correlation (r): 0.69
P-value: 0.000
This suggests popularity is not solely based on content but also on overall audience reach and activity.

Genres Show Predictive Patterns

Genre-based analysis revealed clear tendencies in popularity and revenue outcomes. For example, action and adventure films showed higher average popularity than documentaries or indie genres. ANOVA confirmed a statistically significant difference in average popularity across genre categories:
ANOVA F-statistic: 712.91
P-value: 0.000

Other Factors Are Less Correlated

Some metadata such as runtime, tagline, or homepage URLs did not show strong linear relationships to popularity or revenue, suggesting limited predictive value in basic modeling.

---

3. Methodology & Modeling
All modeling steps and implementation details are documented in the `movie.ipynb` notebook.

Data Preparation & Feature Engineering

To prepare the dataset for machine learning techniques, the following steps were applied:

Feature Engineering

- JSON-like fields (`genres`, `keywords`, `cast`, `crew`) were parsed into lists.

- Columns with excessive missing values (`homepage`, `tagline`) were dropped.
- Categorical variables like `original_language` and `status` were one-hot encoded.
- A combined text-based feature column (`combined_features`) was created for TF-IDF vectorization.

Preprocessing

Using pandas and sklearn utilities:

- Missing values in numeric fields were imputed using the mode.
- Non-numeric content features were removed or transformed before modeling.
- Final cleaned data was stored as `preprocessed_movie_data.csv`.

Model Training

We addressed two tasks:

1. Popularity Prediction (Regression)

This was framed as a regression task with `popularity` as the target. The dataset was split into a 80/20 train-test split, and three models were implemented:

- **Linear Regression**
- **Random Forest Regressor**
- **K-Nearest Neighbors (KNN) Regressor**

All models were tested on numeric-only features after dropping incompatible object columns.

Model	**R^2 (Higher is better)**	**MAE (Lower is better)**	**RMSE (Lower is better)**
Linear Regression	0.738	9.38	14.05
Random Forest	0.622	6.28	21.62
KNN	0.331	11.83	28.77

The **Linear Regression model outperformed all others**, suggesting that popularity can largely be explained by linear numeric relationships within the dataset.

---

5. Conclusions and Reflections

This project successfully demonstrated an end-to-end machine learning pipeline, from data exploration to model implementation and evaluation.

Major Project-Specific Conclusions

- **Budget & Engagement Drive Success** — The strongest predictors of movie performance (popularity and revenue) are budget and vote count.
- **Genre & Content Matter** — Genre strongly influences popularity distribution, confirmed by ANOVA testing.
- **Linear Models Work Well** — The fact that Linear Regression ($R^2 \approx 0.738$) outperformed Random Forest and KNN indicates that popularity and metadata relationships are largely linear.

Insights About the ML Process

Data Preparation is Critical

The most important step was converting JSON strings into usable formats and removing or encoding non-numeric fields. The success of both regression and recommendation pipelines depended heavily on effective preprocessing.

Statistical Testing Adds Confidence

Correlation and ANOVA tests provided strong statistical validation for observations found during visualization. This step is essential before finalizing modeling assumptions.

Small Engineering Details Matter

Errors from JSON formatting, bad column names, and list-based fields required careful debugging. Even small implementation issues (e.g. missing title column in preprocessed data) affected the final workflow until resolved.

Dataset:
TMDB 5000 Movies & Credits Dataset (Kaggle / TMDB Official API)

Libraries:
Pandas, NumPy, Scikit-learn, Matplotlib, Seaborn, JSON, Scipy

Tools:
Jupyter Notebook, ChatGPT (for debugging & documentation support),Gemini and GitHub Copilot (for code assistance and scaffolding)

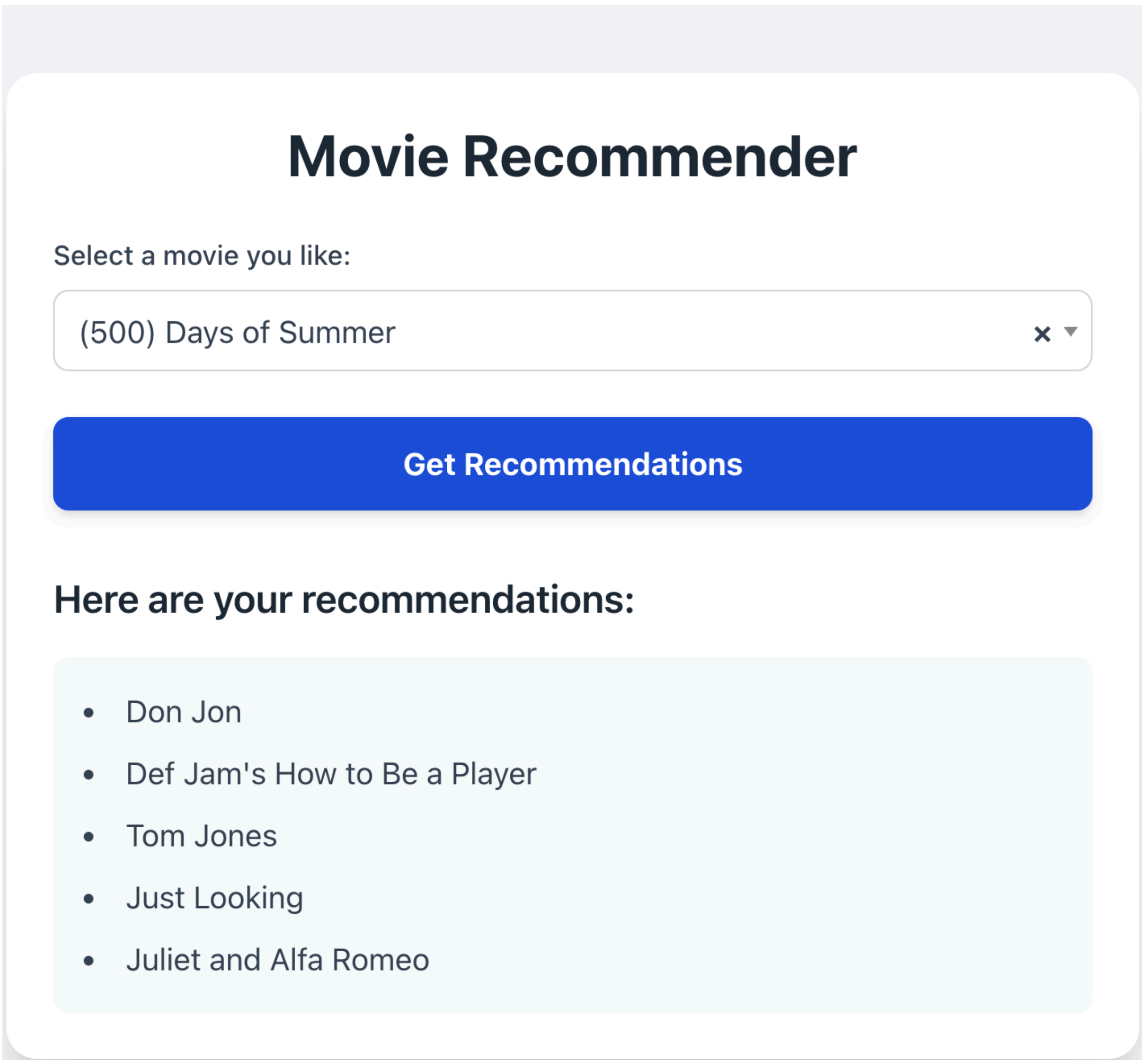
Code:

```
# ==============================================
# 1. Imports
# ==============================================
import pandas as pd
import numpy as np
import ast
import json

from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsRegressor

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity


# ==============================================
# 2. Load & Merge Raw TMDB Data
# ==============================================

# Adjust paths if needed
movies_path = "tmdb_5000_movies.csv"
credits_path = "tmdb_5000_credits.csv"

# Credits sometimes throw parsing errors → use engine='python' + on_bad_lines='skip'
credits_df = pd.read_csv(credits_path, engine="python", on_bad_lines="skip")
movies_df = pd.read_csv(movies_path)

# Merge on id / movie_id using inner join
merged_df = movies_df.merge(
    credits_df,
    left_on="id",
    right_on="movie_id",
    how="inner"
)

print("Merged shape:", merged_df.shape)  # Expect ~3975 rows, 24+ columns


# ==============================================
# 3. Handle Missing Values
# ==========================================
# As per your description:
# - Missing in: homepage, overview, runtime, tagline
# - Drop: homepage, tagline
# - Fill overview & runtime with mode

cols_to_drop = ["homepage", "tagline"]
for c in cols_to_drop:
    if c in merged_df.columns:
        merged_df = merged_df.drop(columns=c)

# Fill overview & runtime with mode (if they exist)
for col in ["overview", "runtime"]:
    if col in merged_df.columns:
        mode_val = merged_df[col].mode()[0]
        merged_df[col] = merged_df[col].fillna(mode_val)

# Confirm no missing values in important columns
print("Missing values after cleaning:")
print(merged_df[["overview", "runtime"]].isna().sum())


# ==========================================
# 4. Parse JSON-like Columns into Lists of Names
# ==========================================

def parse_name_list(x):
    """
    Parse JSON-like string columns (genres, keywords, cast, crew, etc.)
    and return list of 'name' fields as strings.
    """
    if pd.isna(x):
        return []
    if isinstance(x, list):
        # Already parsed
        return x
    try:
        data = ast.literal_eval(x)  # Convert string to Python list/dict
    except Exception:
        try:
            data = json.loads(x)
        except Exception:
            return []
    names = []
    if isinstance(data, list):
        for item in data:
            if isinstance(item, dict) and "name" in item:
                names.append(item["name"])
            else:
                names.append(str(item))
    else:
        # If it's a dict or something else
        if isinstance(data, dict) and "name" in data:
            names.append(data["name"])
        else:
            names.append(str(data))
    return names


json_cols = [
    "genres",
    "keywords",
    "production_companies",
    "spoken_languages",
    "cast",
    "crew"
]

for col in json_cols:
    if col in merged_df.columns:
        merged_df[col] = merged_df[col].apply(parse_name_list)

# Quick check
print("Sample parsed genres:", merged_df["genres"].iloc[0][:5])


# ============================================
# 5. One-hot Encode Categorical Columns
# ============================================

# We'll one-hot original_language and status as you described
ohe_cols = []
if "original_language" in merged_df.columns:
    ohe_cols.append("original_language")
if "status" in merged_df.columns:
    ohe_cols.append("status")
```

```
ohe_df = pd.get_dummies(merged_df[ohe_cols], prefix=ohe_cols, drop_first=True) if
ohe_cols else pd.DataFrame()

# Build preprocessed dataframe:
# keep important numeric + list-based + one-hot columns + original_title
keep_cols = [
    "id",
    "original_title",
    "budget",
    "revenue",
    "runtime",
    "vote_average",
    "vote_count",
    "popularity"
]

for col in json_cols:
    if col in merged_df.columns:
        keep_cols.append(col)

# Filter safely
keep_cols = [c for c in keep_cols if c in merged_df.columns]
preprocessed_df = merged_df[keep_cols].copy()

# Attach one-hot columns
preprocessed_df = pd.concat([preprocessed_df, ohe_df], axis=1)

print("Preprocessed shape:", preprocessed_df.shape)


# ============================================
# 6. Save Preprocessed Data
# ============================================

preprocessed_df.to_csv("preprocessed_movie_data.csv", index=False)
print("Saved preprocessed_movie_data.csv")


# ============================================
# 7. Regression Modeling on Popularity
# ============================================

# Use only numeric columns for regression
numeric_df = preprocessed_df.select_dtypes(include=[np.number]).copy()
```

```
# Make sure popularity exists
if "popularity" not in numeric_df.columns:
    raise ValueError("Column 'popularity' not found in numeric dataframe!")

target = numeric_df["popularity"]
features = numeric_df.drop("popularity", axis=1)

X_train, X_test, y_train, y_test = train_test_split(
    features, target, test_size=0.2, random_state=42
)

print("X_train:", X_train.shape, "X_test:", X_test.shape)

results = {}

def evaluate_model(name, model):
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    mae = mean_absolute_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    results[name] = {"RMSE": rmse, "MAE": mae, "R2": r2}
    print(f"\n{name}")
    print(f"  RMSE: {rmse:.4f}")
    print(f"  MAE:  {mae:.4f}")
    print(f"  R²:   {r2:.4f}")

# 7.1 Linear Regression
lr_model = LinearRegression()
evaluate_model("Linear Regression", lr_model)

# 7.2 Random Forest Regressor
rf_model = RandomForestRegressor(
    n_estimators=200,
    random_state=42,
    n_jobs=-1
)
evaluate_model("Random Forest", rf_model)

# 7.3 K-Nearest Neighbors Regressor
knn_model = KNeighborsRegressor(n_neighbors=5)
evaluate_model("KNN Regressor", knn_model)

print("\n=== Summary of Results ===")
```

```
for name, metrics in results.items():
    print(f"{name}: RMSE={metrics['RMSE']:.4f}, MAE={metrics['MAE']:.4f}, 
R2={metrics['R2']:.4f}")


# ==============================================
# 8. Content-Based Recommendation (TF-IDF + Cosine)
# ==============================================

# We'll use preprocessed_df (with list columns and original_title)

content_cols = ["genres", "keywords", "cast", "crew"]
for col in content_cols:
    if col not in preprocessed_df.columns:
        raise ValueError(f"Required column '{col}' not found in preprocessed_df")

def combine_features(row):
    """
    Combine list-based columns into a single space-separated string,
    lowercased and with spaces removed inside tokens.
    """
    combined = []
    for col in content_cols:
        value = row[col]
        if isinstance(value, list):
            tokens = [str(item).lower().replace(" ", "") for item in value]
            combined.append(" ".join(tokens))
        else:
            combined.append(str(value).lower().replace(" ", ""))
    return " ".join(combined)

preprocessed_df["combined_features"] = preprocessed_df.apply(combine_features, 
axis=1)

# TF-IDF
tfidf_vectorizer = TfidfVectorizer(stop_words="english")
tfidf_matrix = tfidf_vectorizer.fit_transform(preprocessed_df["combined_features"])
print("TF-IDF matrix shape:", tfidf_matrix.shape)

# Cosine similarity
cosine_sim = cosine_similarity(tfidf_matrix, tfidf_matrix)
print("Cosine similarity matrix shape:", cosine_sim.shape)


# ==============================================
# 9. Recommendation Function
# ============================================

def get_recommendations(movie_title, cosine_sim=cosine_sim, df=preprocessed_df,
top_n=10):
    """
    Return top_n similar movies based on cosine similarity of combined_features.
    Uses 'original_title' to match the movie.
    """
    if "original_title" not in df.columns:
        return "Error: 'original_title' column is missing from dataframe."

    # Find index of the movie
    matches = df[df["original_title"].str.lower() == movie_title.lower()]
    if matches.empty:
        return "Movie not found in the dataset."

    idx = matches.index[0]

    # Pairwise similarity scores
    sim_scores = list(enumerate(cosine_sim[idx]))

    # Sort by similarity (descending)
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)

    # Exclude itself [1:top_n+1]
    sim_scores = sim_scores[1:top_n + 1]

    movie_indices = [i[0] for i in sim_scores]
    return df["original_title"].iloc[movie_indices]


# Example usage
sample_movie = "The Dark Knight Rises"
print(f"\nRecommendations for '{sample_movie}':")
print(get_recommendations(sample_movie))
```



Movie Recommender

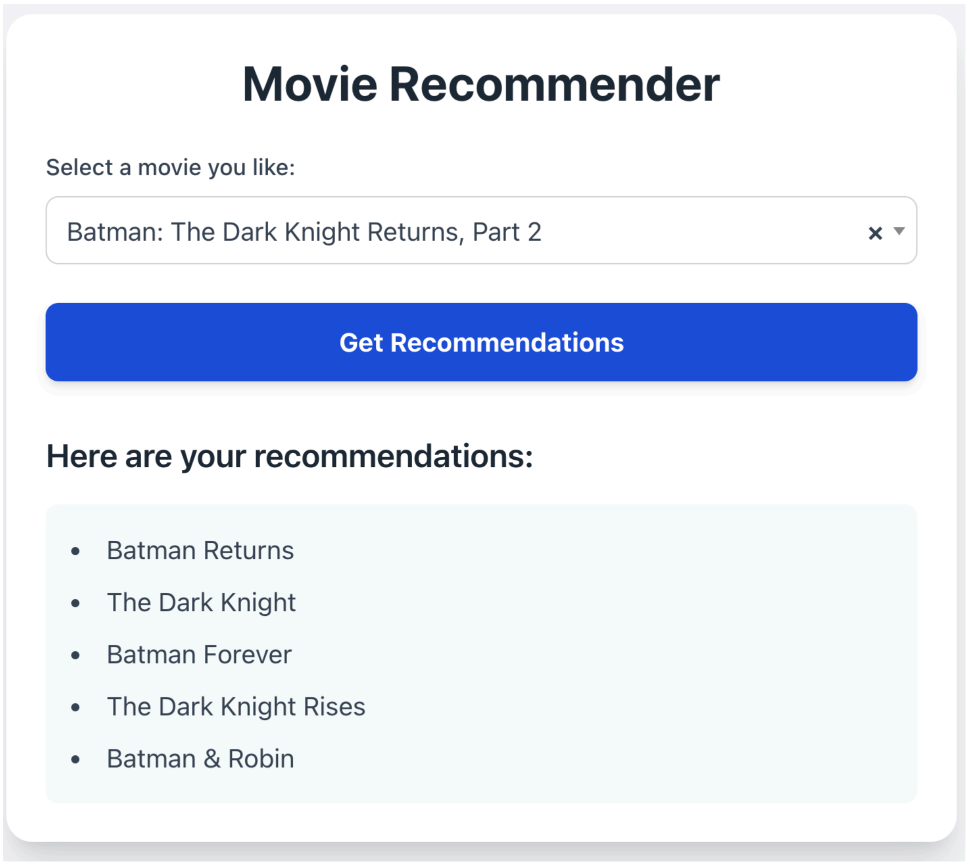
Select a movie you like:

(500) Days of Summer

Get Recommendations

Here are your recommendations:

- Don Jon
- Def Jam's How to Be a Player
- Tom Jones
- Just Looking
- Juliet and Alfa Romeo



Movie Recommender

Select a movie you like:

Batman: The Dark Knight Returns, Part 2

Get Recommendations

Here are your recommendations:

- Batman Returns
- The Dark Knight
- Batman Forever
- The Dark Knight Rises
- Batman & Robin