BUILDING INTELLIGENT RECOMMENDATION SYSTEMS

WORKSHOP DETAILS

- Instructor: Tanya Khanna
- Email: tk759@scarletmail.rutgers.edu
- Course Materials: Github Link - https://github.com/Tanya-Khanna/Data-Science-Workshop---Spring-2025---NBL-
- Workshop Recordings: https://libguides.rutgers.edu/datascience/python

WORKSHOPS SCHEDULE

Introduction to Python Programming	February 3, 2025; 2 - 3:30 PM
Mastering Data Analysis: Pandas and Numpy	February 10, 2025; 2 - 3:30 PM
Introduction to Tableau: Visualizing Data Made Easy	February 17, 2025; 2 - 3:30 PM
Introduction to Machine Learning: Supervised Learning	February 24, 2025; 2 - 3:30 PM
Introduction to Machine Learning: Unsupervised Learning	March 3, 2025; 2 - 3:30 PM
Data-Driven Decision Making: A/B Testing and Statistical Hypothesis Testing	March 10, 2025; 2 - 3:30 PM
Demystifying Generative AI	March 24, 2025; 2 - 3:30 PM
Large Language Models: From Theory to Implementation	March 31, 2025; 2 - 3:30 PM
Generative AI Applications with AI Agents	April 7, 2025; 2 - 3:30 PM
Building Intelligent Recommendation Systems	April 14, 2025; 2 - 3:30 PM

https://libcal.rutgers.edu/calendar/nblworkshops?cid=4537&t=d&d=0000-00-00&cal=4537&inc=0

WORKSHOP AGENDA

- Introduction to Recommender Systems
- Why Are They So Important for Businesses?
- How Recommender Systems Learn What You Like
- Types of RecSys
- RecSys Algorithms
- CineMate: Movie Recommendation App

INTRODUCTION TO RECOMMENDER SYSTEMS

Imagine this:

- You’re listening to a song on Spotify—and the next track that plays is exactly your music taste.
- You’re watching a show on Netflix—and it suggests another series you instantly want to binge.
- You’re shopping on Amazon—and it shows the perfect product that you didn’t even know you needed.

Sounds like magic? It’s not. That’s the power of **recommender systems.**

📈 The Problem: Too Many Choices

In today's world, we're surrounded by an overwhelming amount of content:

- 🎥 YouTube: 500 hours of video are uploaded every minute. That's 82 years' worth of videos in just 1 hour!
- 🎵 Spotify: Over 100 million songs and podcasts are available.
- 🛒 Amazon: You can buy more than 350 million products.
- 📺 Netflix: To watch all its shows 24/7, you'd need over 10 years. Watching just 4 hours/day? That'll take you 61 years!
- 👥 Facebook: Over 3 billion people use it each month—you can't possibly connect with everyone.
- 💼 LinkedIn: Around 140 job applications are submitted every second. That's millions of jobs and people looking—every day.

So, how do you find the right song, the perfect product, or the best job without spending hours searching?

🤖 The Solution: Recommendation Systems

Recommender systems do the searching for you.
They act like your digital assistant, figuring out what you like based on what you:

- Listen to
- Watch
- Click
- Search
- Buy
- Like or dislike

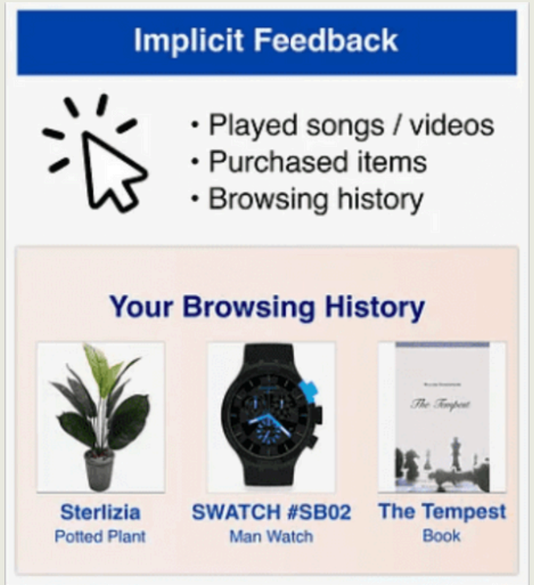
By understanding your tastes and behaviors, they help narrow down millions of options to just the few that matter most to you. They remember what you did before (your history). They look at what people like you are doing (others with similar taste). Then, they predict what you might enjoy next.

💼 Why Are They So Important for Businesses?

Recommender systems are incredibly important for businesses because they play a key role in keeping users engaged, increasing sales, and building long-term customer loyalty. On platforms like Netflix and YouTube, personalized recommendations encourage users to stay longer by constantly suggesting content they'll likely enjoy. For e-commerce giants like Amazon, recommender systems boost sales by showcasing products tailored to individual preferences, increasing the chances of purchase. Music platforms like Spotify keep users coming back by curating playlists and suggesting new songs that match their taste. In short, better recommendations lead to happier users—and happier users drive more business.

Better recommendations = happier users = more business.

HOW RECOMMENDER SYSTEMS LEARN WHAT YOU LIKE



They rely on your feedback—what you watch, click, buy, or rate.

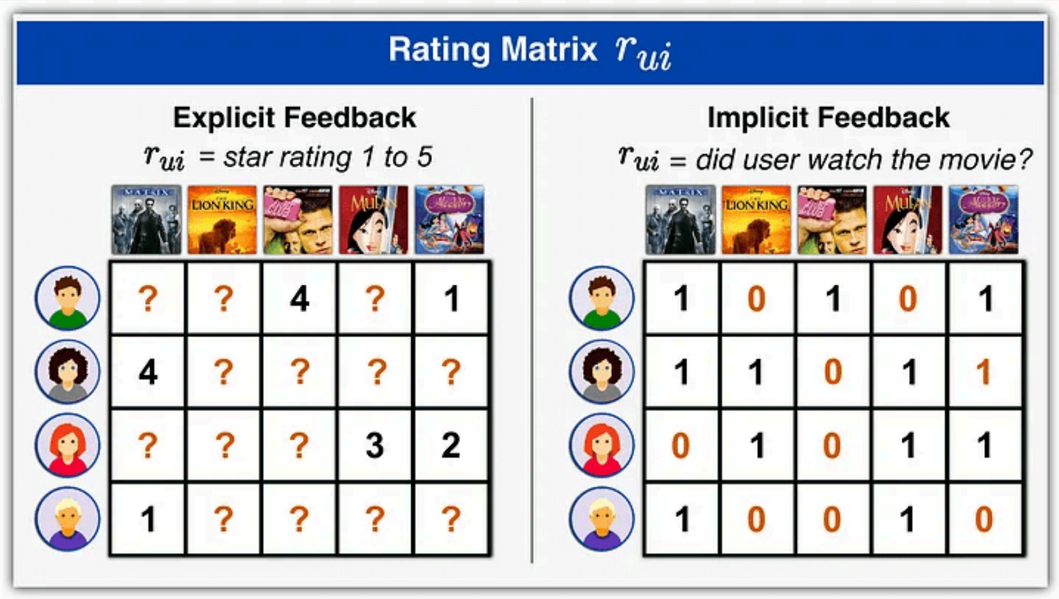
Explicit Feedback (You tell the system what you like)

- You rate a product (like 4 stars out of 5)
- You click thumbs-up or thumbs-down on a video
- You write a review after watching or using something
- 🔍 Clear and accurate, but…
- 😅 Most people don’t take the time to give this kind of feedback

Implicit Feedback (The system watches what you do)

- You watch a video
- You listen to a song
- You buy something online
- 📈 Easier to collect and happens all the time
- ⚠️ But it’s less accurate—maybe you bought something for a friend, not yourself
- Still, it’s the most common type of feedback used today

HOW RECOMMENDER SYSTEMS LEARN WHAT YOU LIKE



User-Item Matrix

- The system builds a big table (called a user-item matrix)
- 📊 Rows = users, Columns = items (movies, songs, products)
- Each box shows if you rated, watched, clicked, or bought something
- Most boxes are empty because we don’t interact with everything
- That’s called a sparse matrix
- The goal is to predict the missing pieces and recommend the right stuff

TYPES OF RECSYS

Content-Based Recommendation System

This system recommends items similar to what you've liked before. It looks at the features of the items (like genre, actors, or category) and matches them to your preferences. If you liked Batman Begins and The Dark Knight, the system might suggest The Dark Knight Rises because it's in the same genre and has the same actors.

- Utility Matrix: A table that shows how each user interacts with each item. For example, it shows whether you liked or rated a movie. It helps the system learn what types of items each user prefers.
- Creating Item Profiles: Every item (movie, product, etc.) is described by a profile that includes: Genre, director, cast, year (for movies); Brand, color, material (for clothes or products)

These features are turned into numbers using a method called TF-IDF. TF (Term Frequency): How often a word appears in one item's description. IDF (Inverse Document Frequency): How unique that word is across all items. Common words like "the" get low scores; rare words like "gothic" get high scores. TF-IDF = TF × IDF: Helps the system figure out which words best describe an item.

- Creating User Profiles: Built from the user's past interactions (watched movies, liked products, etc.). It forms a summary of the user's preferences. If you mostly watch action and thriller movies, your profile will reflect that.

How Recommendations Are Made

1. Cosine Similarity
   - Compares how similar the user profile and item profile are.
   - If the two are very similar (small angle), it's a good match.
2. Classification Models
   - Machine learning models (like decision trees) predict if you'll like an item.
   - Based on characteristics of both you and the item.

- When you search for a movie on Netflix or a photo on Google, you often see similar options pop up. These are shown using similarity search based on your input. The system compares the "features" of what you searched for and finds other items that are close in meaning or appearance.

TYPES OF RECSYS

Collaborative Filtering Recommendation System

Instead of looking at item features (like genre or brand), CF looks at user behavior. It uses feedback from many users (ratings, clicks, purchases) to predict what you might like. If people with similar tastes to you liked something, the system thinks you might like it too.

1️⃣ Memory-Based Collaborative Filtering

- Works directly with the user-item interaction matrix
- Simple and easy to understand
- Doesn't need a machine learning model
- Best for small datasets
- Struggles with large and sparse data

Types:

- User-Based CF
  - Finds users similar to you and recommends what they liked
  - If User A and User B both liked Batman Begins and Justice League, and B also liked Avengers, A might like Avengers too.
- Item-Based CF
  - Looks at items you've liked and finds similar items to recommend
  - If you liked Movie 1 and Movie 2, and Movie 2 is similar to Movie 3, the system will recommend Movie 3.

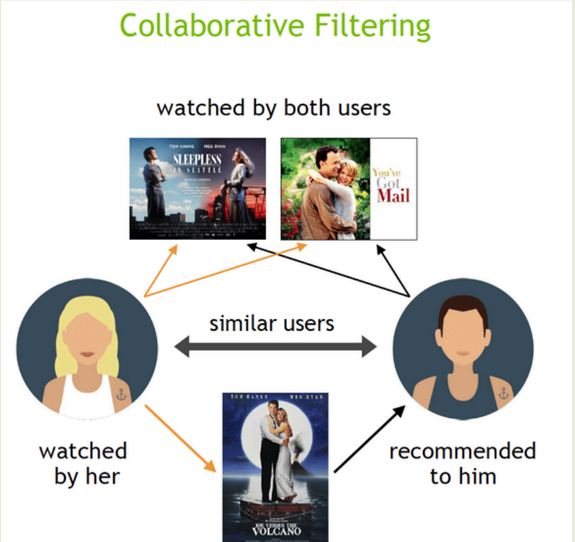
2️⃣ Model-Based Collaborative Filtering

- Uses machine learning models to find patterns
- Learns hidden (latent) features from data that users/items have in common
- Works well with large datasets and can be very accurate

Common Techniques:

- Matrix Factorization (breaks down large matrices into smaller ones to learn patterns)
- SVD (Singular Value Decomposition)
- Clustering
- Deep Learning models (e.g., Neural Collaborative Filtering)

Model-based CF works best with explicit feedback (like ratings), but it can also be adapted for implicit feedback (clicks, views).



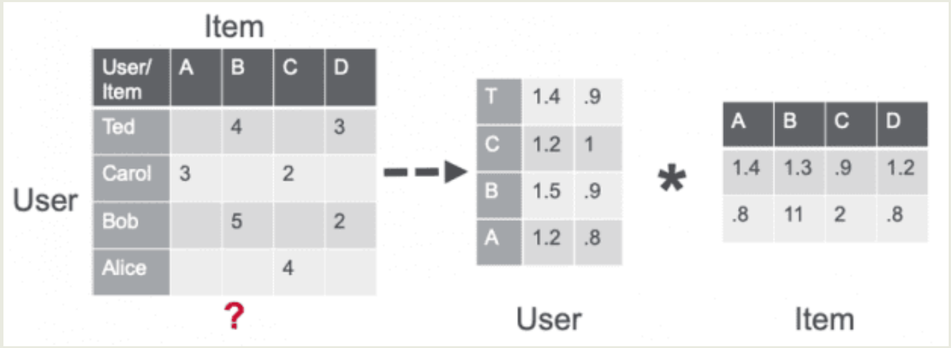
RECSYS ALGORITHMS

Matrix Factorization

It's a way to predict missing ratings (like "?") in a user-item table (e.g., who liked which movie). It breaks a large matrix (user vs. item) into two smaller ones:

- One for users (what kinds of items they like)
- One for items (what type of users like them)

These small vectors capture hidden factors (like genre preferences, brand loyalty, etc.) We multiply these two small matrices to fill in the blanks in the original table.



On the left, you see a user-item rating matrix with some missing values (like Alice missing a rating for Item B). The matrix is split into: A user feature matrix e.g., Ted (T), Carol (C), etc. have 2 feature values each.

An item feature matrix (middle-right) e.g., Items A–D also have their own 2 features.

These matrices are multiplied to reconstruct the original matrix and predict the missing rating (the red "?").

For Explicit Feedback (like ratings)

- Users give star ratings (e.g., 1 to 5 stars for a movie).
- We use a method called Probabilistic Matrix Factorization:
  - The system learns user and item vectors
  - It predicts the ratings using a linear model
  - Loss function: mean squared error (MSE) between predicted and real ratings

Optimization methods used:

  - Stochastic Gradient Descent (SGD): learns step-by-step
  - Alternating Least Squares (ALS): solves equations in blocks — faster in parallel

For Implicit Feedback (like views, clicks, watch time)

- No ratings given; only actions observed (e.g., user watched 30% of a movie).
- Matrix factorization here uses:
  - how many times (or how long) user u interacted with item i
  - confidence in the interaction
  - 1 if item was interacted with, else 0
- The model learns from these to estimate preferences
- Loss function = weighted squared error, weighted by confidence

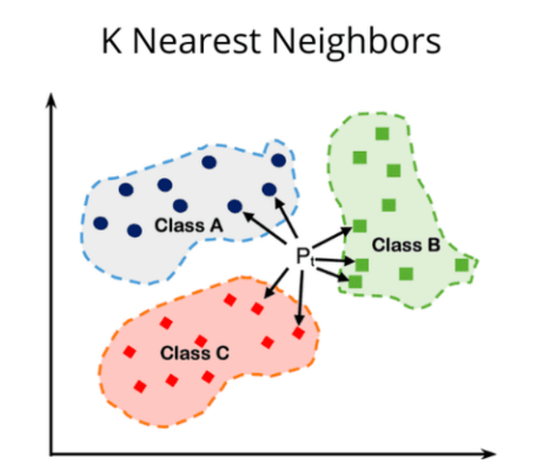
Hybrid Approach: SVD++

- Combines explicit + implicit feedback. For example:
  - You rated Inception 5 stars (explicit)
  - You watched Interstellar twice but didn't rate it (implicit)
- SVD++ improves accuracy by considering both types. Adds the influence of all items the user interacted with (even without rating)

RECSYS ALGORITHMS

K-Nearest Neighbors (KNN)

KNN is like saying, "Let me recommend things that people similar to you liked." It works by finding the closest matches (or “neighbors”) to a user or item using similarity scores.



Each dot, square, and diamond represents a user or item belonging to different groups or categories: 🔵 Class A, 🟩 Class B, 🔴 Class C
The point labeled P is a new user or item the system doesn’t know much about yet.
The arrows show the system finding P’s k-nearest neighbors — the closest points (users/items) based on distance.
Based on those neighbors, P will be classified (or recommended) to the most common group nearby—likely Class C in this case.

Compare Users or Items; Look at movie ratings, product views, or song likes; Use similarity measures like: Euclidean distance (how far two users are on a graph) and Cosine similarity (how aligned their preferences are); Find Neighbors (k); Pick the k most similar users or k most similar items; If k = 3, pick the top 3 closest matches; Make Recommendations.

- If using User-based KNN:
  - Recommend items liked by similar users that you haven’t seen.
  - If Alice likes what Bob likes, suggest Bob’s favorites to Alice
- If using Item-based KNN:
  - Recommend similar items to the ones you already liked.
  - If you liked Inception, suggest movies rated similarly like Interstellar

- ✅ KNN is easy to understand and good for small datasets.
- ❌ But it struggles with huge datasets (too slow) and sparse data (lots of missing ratings).
- ✅ Matrix Factorization (like SVD) handles those challenges better by learning hidden patterns from user behavior.

RECSYS ALGORITHMS

Logistic Regression

It’s a simple and powerful model used to predict probabilities—especially useful for:

- Click-through rate (CTR) prediction
- Will a user click/buy/watch something or not?

The prediction is a number between 0 and 1 (0 = no interest, 1 = strong interest)

- It works well for binary outcomes: clicked/not clicked, liked/not liked
- It can use a lot of extra information, like:
  - 👤 User details (age, gender, location)
  - 🛍️ Item features (category, price, popularity)
  - 🕒 Context (day of the week, device used)
- Helps solve the cold start problem by using these side features
- (i.e., how to recommend when there's no user history yet)

How Does the Model Learn?

- It uses a loss function with:
  - A logistic loss term (to learn how wrong the prediction was)
  - A regularization term (to prevent overfitting)
- The loss is minimized during training using optimization techniques

Try to make the predicted value close to the real one (0 or 1), but don’t overcomplicate the model.

⚠️ Limitation

- Logistic Regression looks at each feature on its own.
- It doesn’t automatically understand feature combinations (like: “young users + expensive products” patterns).

🔧 Solutions: Adding Feature Interactions

- Poly2 (Degree-2 Polynomial Model)
  - Adds a term for every feature pair
  - Learns combinations like:
  - Gender=Male + Category=Shoes
- Decision Trees + Logistic Regression
  - A tree is first used to split and group features
  - The result is passed to Logistic Regression for final prediction
  - Gives the model non-linear power without making it too complex

✅ Why It's Still Popular

- Fast, interpretable, easy to train
- Works well with lots of side info
- Great for baseline models and large-scale CTR prediction

RECSYS ALGORITHMS

Deep Learning (DL)

- Traditional models work well—but deep learning gets better with more data.
- DL models handle complex patterns in user behavior and item features.
- Used by major companies like Airbnb, Facebook, Google, and YouTube.
- Improves both accuracy and personalization at scale.

🔁 Life Cycle of a Deep Learning Recommender System

🏋️ 1. Training Phase

- The model learns from past user-item interactions (e.g., clicks, views, ratings).
- Learns:
  - Patterns in user behavior
  - Item features (e.g., genre, price, category)
- Output: a trained model that understands what kind of items users prefer.

🧠 2. Inference Phase

- The model makes real-time predictions for new recommendations.
- Involves 3 key steps:
  a. Candidate Generation – Find items similar to what the user might like
  b. Candidate Ranking – Score and sort those items by likelihood of interest
  c. Filtering – Remove irrelevant or already seen items
- The best-ranked items are recommended to the user instantly.

How It Works (Step-by-Step)

1. Collect user-item interaction data (e.g., ratings, clicks)
2. Learn embeddings: represent users and items as vectors in high-dimensional space (Embeddings turn categorical info (like user ID, item ID, location) into dense numeric vectors; Similar users/items have similar embedding vectors; Learned during training and used in scoring new recommendations; Final step: compute the dot product between user and item embeddings; Apply sigmoid activation to convert score into a probability (0 to 1)
3. Train the model to predict the likelihood a user interacts with an item
4. For new recommendations:
   - Extract user context (recent history, demographics)
   - Feed it into the model
   - Get personalized suggestions using learned embeddings

- **Deep learning models can personalize at massive scale; Capture both short-term signals (recent views) and long-term preferences; Provide more relevant, real-time recommendations**

POPULAR DL MODELS FOR RECOMMENDATION

Wide & Deep Model (Google)

- A hybrid model that combines:
  - A wide (linear) model → memorizes patterns like "user from City X buys Brand Y"
  - A deep (neural net) model → learns complex relationships using embeddings
- Wide component:
  - Fast, simple, good at remembering frequent, straightforward patterns
- Deep component:
  - Uses embedding layers for categorical features (like user ID, product ID)
  - Captures general patterns and can handle new or rare combinations

➕ Final prediction:

- Combine the outputs of both models - This captures both short-term trends (wide) and long-term preferences (deep)

Why It’s Useful

- Solves the memorization vs generalization problem
- Powers systems like the Google Play app store recommendations

POPULAR DL MODELS FOR RECOMMENDATION

DLRM: Deep Learning Recommendation Model (Meta/Facebook)

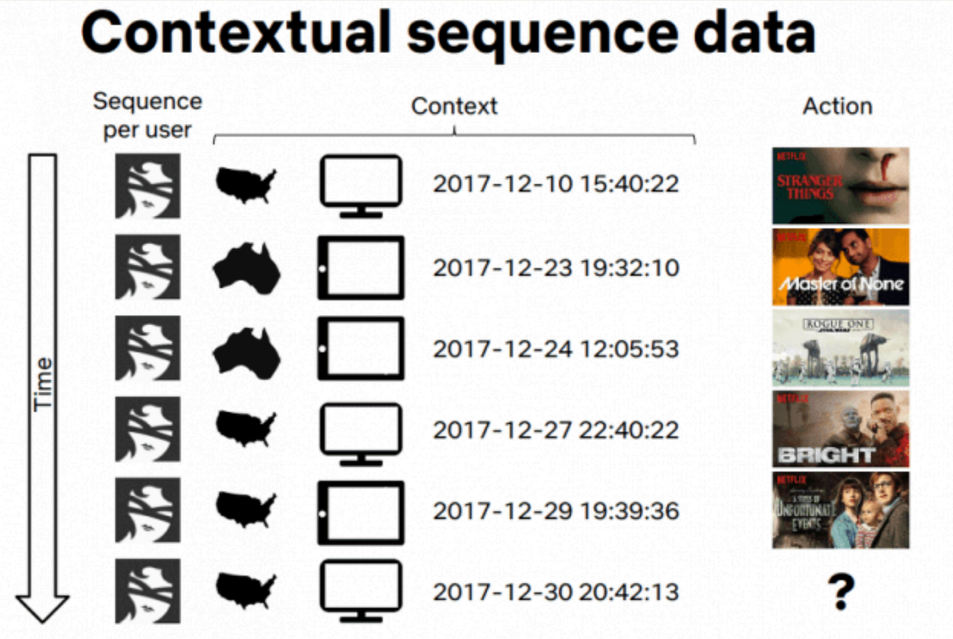
A high-performance model made for large-scale recommendation tasks

- Works with both:
  - Categorical features (e.g., item ID, gender, region)
  - Numerical features (e.g., price, time spent, age)

1. Embedding Layer:
   - Converts categorical features into dense vectors
2. Bottom MLP:
   - Processes numerical features
3. Feature Interaction Layer:
   - Takes dot products between pairs of embeddings to capture interactions
4. Top MLP:
   - Combines everything and makes the final prediction

Why It’s Useful: Efficient at handling huge datasets (like those used by Facebook Ads or Marketplace)

CONTEXT FILTERING IN RECOMMENDER SYSTEMS



- It’s about making better recommendations by understanding the situation around user actions.
- The system looks at when, where, and how users interact (e.g., device, time, location).
- This helps recommend items that fit the moment.

📍 Context Matters: Netflix Example

- Netflix tracks:
  - 🎯 What you watch (e.g., Stranger Things)
  - 💻 Device used (e.g., computer)
  - 🌍 Location (e.g., U.S.)
  - 📅 Date & time (e.g., Dec 10, 2017, evening)
- With this data, Netflix can:
  - Recommend action shows in the evening if that’s your trend
  - Adjust recommendations based on device type or day of week
- The system uses all this context to predict what you’ll watch next (marked by a “?” in the table).

SESSION-BASED VS SEQUENCE-BASED RECOMMENDATIONS

🛍️ Session-Based Recommender Systems

- Focus only on what the user is doing right now in their current session.
- Don't need long-term user history.
- Useful when: Users are anonymous; Behavior is short and goal-specific (like browsing 5–6 products)

📈 Sequence-Based Recommender Systems

- Consider the order of all past interactions (across sessions).
- Model long-term behavior and evolving preferences.
- Useful when: User history is available and meaningful; Order of interactions matters (e.g., binge-watching)

These models embed user actions like words in a sentence. The system learns the “meaning” of interactions over time.

Analogy to NLP (Natural Language Processing)

- Just like words in a sentence are embedded and processed to understand meaning...
- User actions (clicks, scrolls, views) are embedded and processed to understand behavior.

Why Use Context Filtering + DL Models?

- More accurate and personalized recommendations
- Handle anonymous sessions or long-term sequences
- Adapt to real-time signals like time of day or device used
- Boost engagement with timely, relevant suggestions

RECSYS ALGORITHMS
Large Language Models

🚀 How Did We Get Here?

- Traditional Deep Neural Networks (DNNs) have already transformed recommender systems by:
  - RNNs / LSTMs / GRUs – great for sequential behavior (e.g., product clicks, viewing history)
  - GNNs (Graph Neural Networks) – used for data that looks like a network, such as users and friends
  - BERT (for reviews) – to extract meaning from user-written text

⚠ But There Are Some Limitations

- DNN-based RecSys models are:
  - Not good at understanding deep text (e.g., reviews, item descriptions, queries)
  - Usually built for one specific task (e.g., rating prediction or product ranking)
  - Struggle with complex recommendations like:
    - Building travel plans
    - Suggesting bundles
    - Handling natural conversations
  - Poor generalization across tasks without retraining
  - Doesn't offer explanations for their recommendations

💡 Enter LLMs (Large Language Models) like ChatGPT, BERT, Gemini

- Trained on huge amounts of text → understand language deeply (Understands product descriptions, reviews, and user queries)
- Can generate and explain recommendations in natural language
- Support multi-tasking (e.g., recommend + explain + summarize)
- Allow conversation-based recommendations (like a shopping assistant!)
- Handle complex reasoning using:
  - In-context learning – Can make personalized recommendations from a short prompt
  - Chain-of-thought – reason step-by-step (e.g., "You like A, so maybe B and C will interest you")

RECSYS ALGORITHMS
Large Language Models

🧠 How LLMs Improve Recommendations

- Conversational Interaction: You can ask, "What should I watch tonight?" and model tailors suggestions based on mood, past preferences, etc.
- Enhanced Explainability: Instead of just showing a product, the model can say: “This backpack is similar to the one you liked last week and is great for hiking.”
- Dynamic Personalization: Can reason through user goals, not just past clicks.
- Multi-step Planning: Useful in recommending travel plans, study routines, or multi-item bundles.

Current LLM-Based RecSys Techniques

- RAG (Retrieval-Augmented Generation): Combines LLMs with a database to give factual, grounded recommendations
- Prompt Engineering: Asking the right question to steer the LLM toward better recommendations
- Fine-tuned LLMs: Models adapted for e-commerce, travel, or music recommendations
- Chain-of-Thought (CoT) Reasoning: Helps with planning or recommendations that require logic and steps

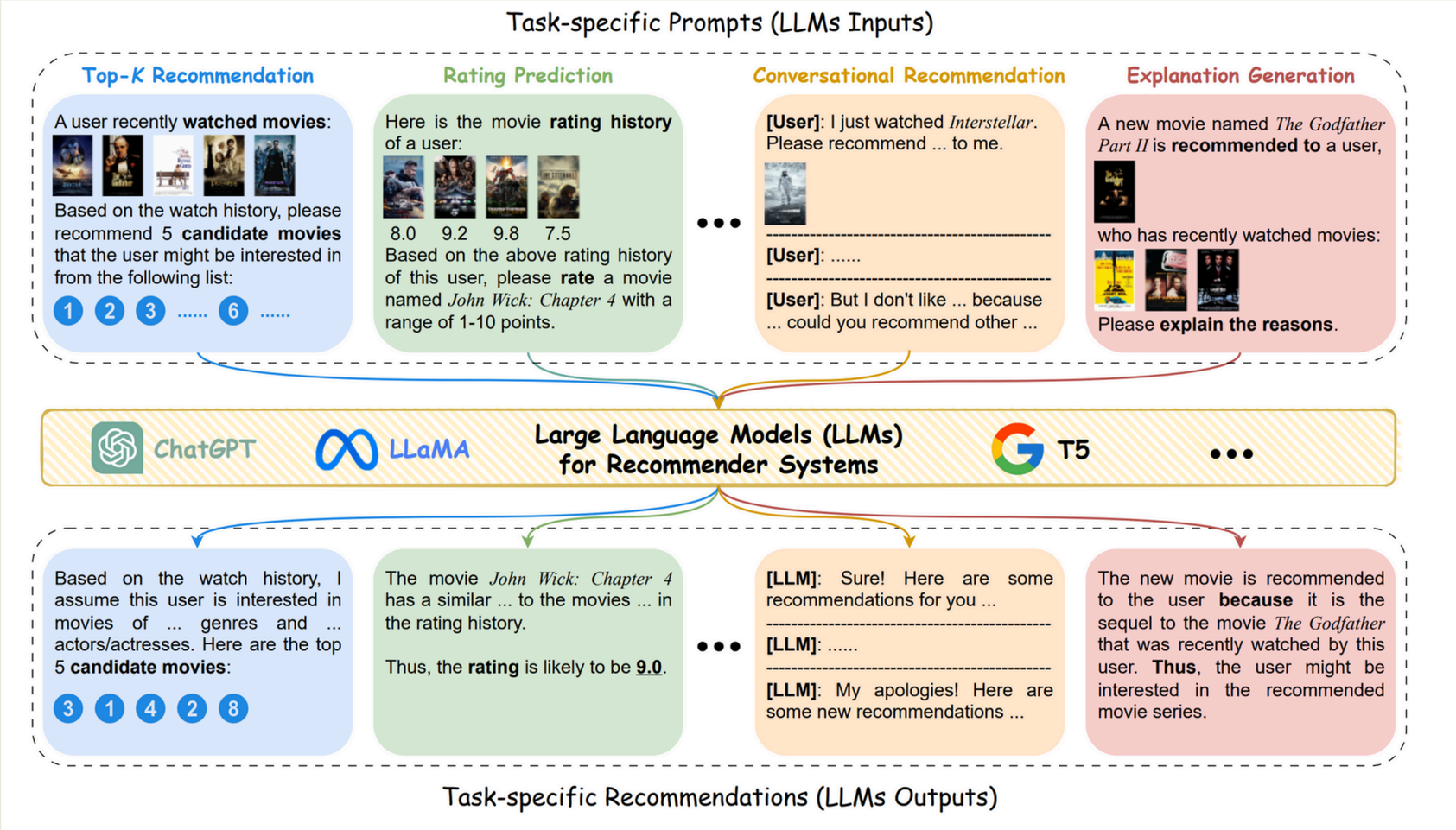
Travel App using an LLM: User says: “I want a 3-day beach vacation in Thailand under $500.” The LLM understands preferences, budget, timeline → suggests places, flights, hotels, AND builds an itinerary.

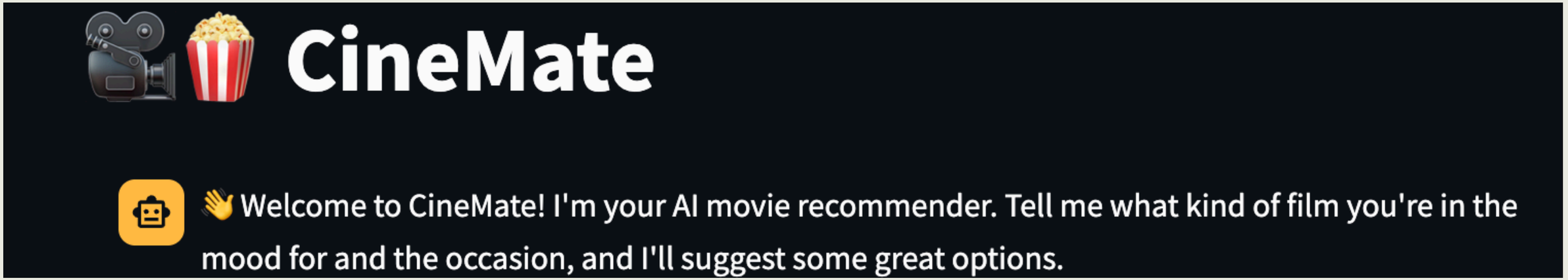
Who’s Using LLMs in RecSys?

- OpenAI: ChatGPT plugins for shopping and travel
- Google: Gemini for travel, entertainment, and contextual suggestions
- Amazon: Personalized shopping assistants powered by LLMs
- Spotify: AI DJs using language models to recommend music and explain choices
- Companies are exploring hybrid models that combine: Traditional ranking + LLM-based reasoning; Pre-filtered candidates + LLM-generated explanations

RECSYS ALGORITHMS

Large Language Models





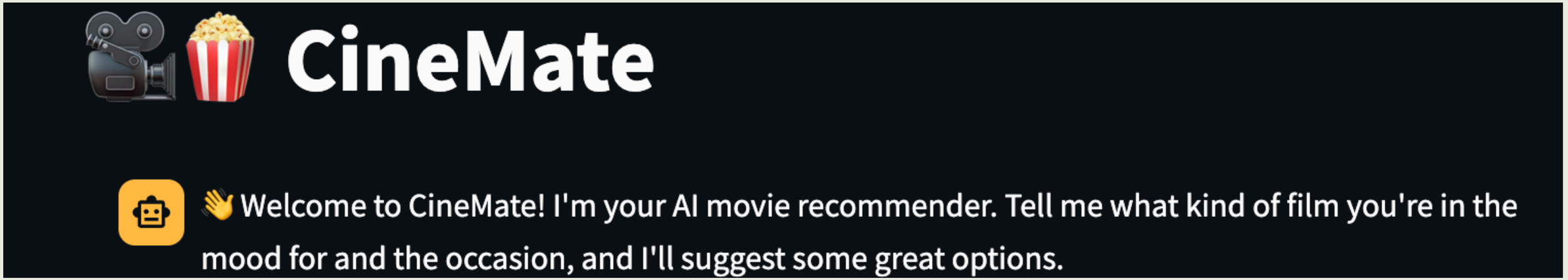
🚀 What Are We Building?

We’re building a smart movie recommender where users simply describe the kind of movie they want (e.g., "sci-fi adventure for a movie night with friend") — and the app responds with movies, along with a friendly AI-generated recommendation.
It's powered by:

- natural language understanding
- vector-based semantic search
- generative AI for summaries
- and a friendly user interface

🛠️ Tools & Why We Use Them

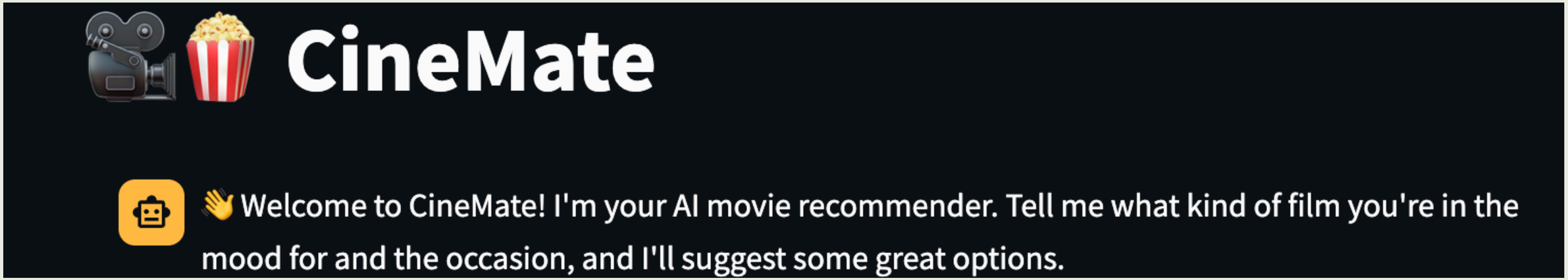
- Streamlit is an open-source Python framework that helps you build interactive web apps with just a few lines of code.
- Perfect for anyone building AI apps without web dev hassle.
- Why we use it: To build the app’s interactive interface for search and chat
- Benefits:
  - Requires minimal code to create powerful UIs
  - Built-in support for chat, images, sliders, and buttons
  - Seamlessly integrates with Python code and AI APIs
  - Ideal for rapid prototyping and sharing AI demos



🛠️**Tools & Why We Use Them**

- Weaviate is an AI-native database designed to help you build amazing, scalable, and production-grade AI-powered applications. It doesn't just store text — it understands it.
- It lets you:
  - Search using meaning, not just exact words
  - Store your own data (like movies) as objects
  - Use powerful AI models like Cohere to turn that data into embeddings (vectors)
- Benefits:
  - Supports vector-based (semantic) queries
  - Fast retrieval of relevant results based on meaning, not just keywords
  - Integrates with Cohere for automatic text vectorization
  - Ideal for Retrieval-Augmented Generation (RAG) pipelines

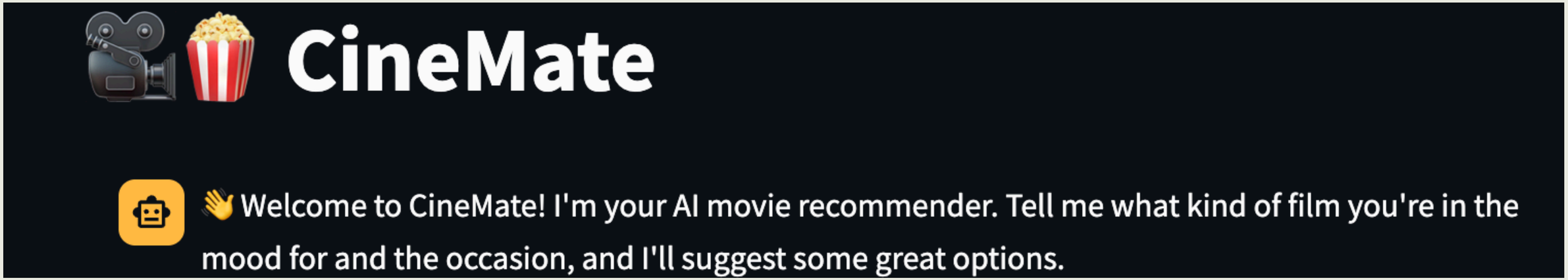
- The Streamlit–Weaviate Connection
  - To connect Weaviate with your app easily, we use the st-weaviate-connection package. It simplifies how you:
  - Connect to Weaviate Cloud
  - Run keyword/semantic/hybrid queries
  - Perform RAG (Retrieval-Augmented Generation) using Cohere directly from the database



🛠️**Tools & Why We Use Them**

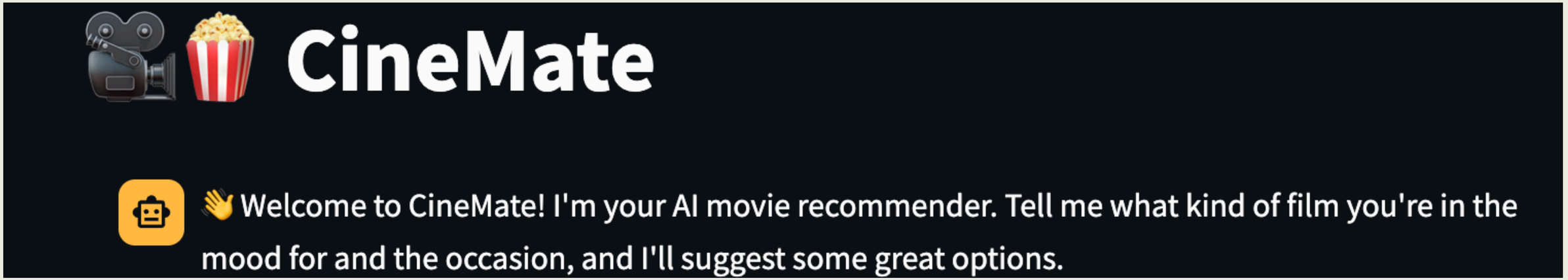
- Cohere offers powerful AI models for:
  - Turning text into embeddings (for meaning-based search)
  - Generating human-like summaries, responses, and recommendations
- In this app, we use Cohere for:
  - Embedding: It turns each movie's title, tagline, and description into a vector (i.e., a "meaning fingerprint")
  - Generation: It summarizes the most relevant search results into a personalized recommendation
- Benefits:
- Captures deep context and meaning of text using embeddings
- Generates natural language recommendations from structured data
- No need to train your own models — use powerful prebuilt AI

- TMDB API: Free movie database API (The Movie Database)
- Why we use it: To fetch movie posters and metadata
- Benefits:
  - Rich collection of movie posters, descriptions, and metadata
  - Free to use for personal/academic projects
  - Adds visual appeal to the app by displaying movie posters



How Will the Recommendation Work?

1. User enters a search: You say: **“I want an animated family film for family viewing”**
2. Streamlit sends the query “animated family film” to Weaviate:
   - If using **Keyword** mode: finds matches by words (matches text using BM25 (based on word frequency) - good for exact phrase matches)
   - If using **Semantic** mode: finds matches by meaning (via vector similarity) (uses vector similarity (meaning-based) - great when the query and data are phrased differently
   - If using **Hybrid**: balances both (combines keyword and semantic - best of both worlds — accurate and intelligent)
3. Weaviate then retrieves the top 10 movies that best match the query, it also filters for release years you chose.
4. These movies are passed to Cohere’s generative model, along with the original query with a prompt like: **“Suggest one to two movies out of the following list, for a family viewing. Give a concise yet fun and positive recommendation.”**
5. Cohere generates a recommendation blurb.
6. Streamlit displays:
   - Poster images (from TMDB)
   - Titles & taglines
   - The recommendation message



🤖 How Are LLMs Generating These Movie Recommendations?

1. The Prompt: "Here's the context, now generate"
You give the LLM (Cohere in this case) two things:

- A user query (e.g., "family viewing")
- A list of movie titles + taglines or summaries retrieved by Weaviate

You wrap this into a prompt like:
"Suggest one to two movies out of the following list, for a family viewing. Give a concise yet fun and positive recommendation."
This is called prompt engineering — you're guiding the model by framing the task clearly.

2. Tokenization: Understanding the Input
The LLM converts everything into tokens — numerical representations of words and subwords.
For example: ["Suggest", "one", "to", "two", "movies", ...] → [1401, 225, 22, 133, 6783, ...]
This allows the model to "understand" relationships between terms like:

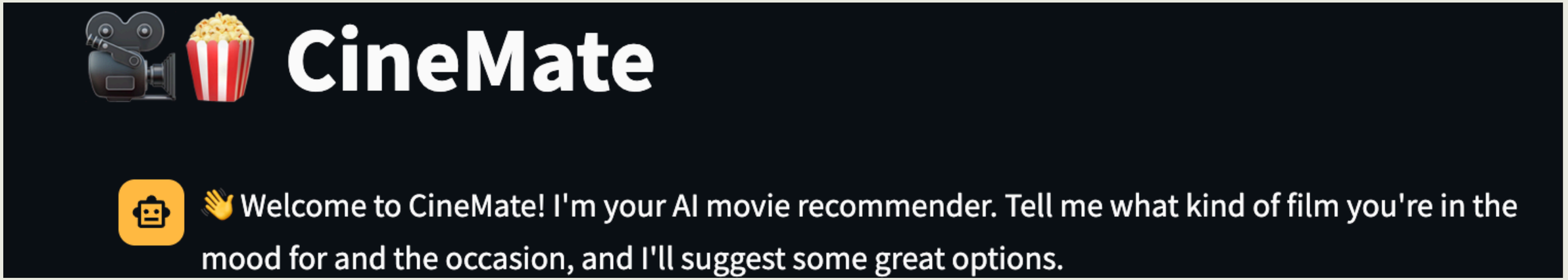
- "family viewing"
- "Pixar" + "animation" + "rated G"
- "fun" + "safe for children"

3. Contextual Attention: Relevance Filtering
The LLM processes the input through multiple attention layers, identifying:

- Which movies are most relevant to the viewing context
- Which titles sound more positive, emotional, suitable

It evaluates tokens not just in isolation but in relation to all others (thanks to self-attention in the transformer architecture).
So if a movie's description includes:
"A heartwarming tale of friendship and magic, perfect for all ages."
...it'll score higher for a "family viewing" context than a dark mystery film.

Attention enables the model to weigh the importance of each token relative to every other token in the input, helping it understand context, relationships, and relevance. For example, the model learns to associate words like "animated," "heartwarming," and "friendship" as more suitable for a family context, and it can focus on movie descriptions that reflect these traits. By computing these relationships across multiple layers of attention, the model gains a deep understanding of the input, allowing it to craft a personalized, coherent response.



🤖 How Are LLMs Generating These Movie Recommendations?

4. Text Generation: Crafting the Recommendation
Now the model starts generating the recommendation one token at a time.
It does this by:

- Predicting the most likely next token based on previous ones
- Repeating until it forms a fluent, human-like sentence

For example:

- "For your family night, try..."
- "... Coco — a colorful and emotional journey through family traditions."

It's not just picking a canned message — it's writing something new based on the exact data and prompt you gave it.

5. Why It Feels Personalized
Even if two users give similar prompts like:

- "family viewing"
- "movie night with kids"

The LLM:

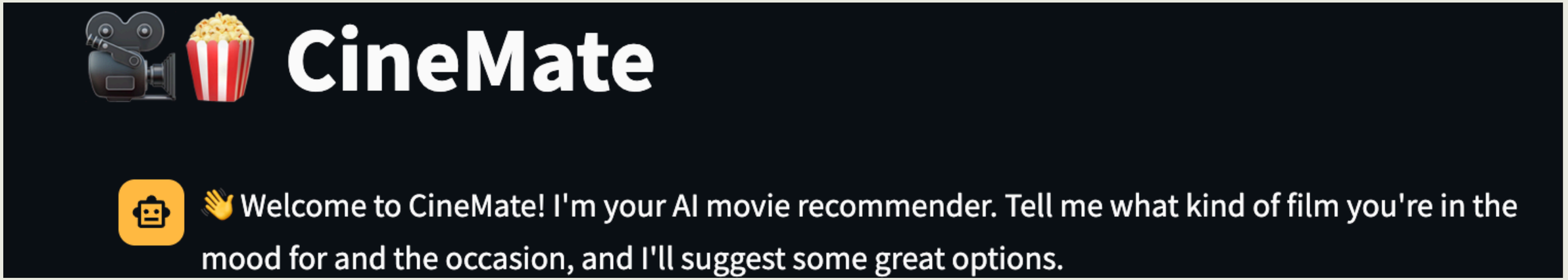
- Re-interprets the semantic intent
- Ranks movie options differently
- Uses temperature and randomness to vary phrasing

So recommendations feel personalized and fresh.

Why This Is Called RAG (Retrieval-Augmented Generation)

- Retrieval → You pull the top 10 movies using Weaviate
- Augmented Generation → You feed those into Cohere's LLM
- The result: smarter, grounded, context-aware outputs

This avoids hallucinations and ensures the LLM recommends movies you actually have.



🗂️ What Does Each File Do?

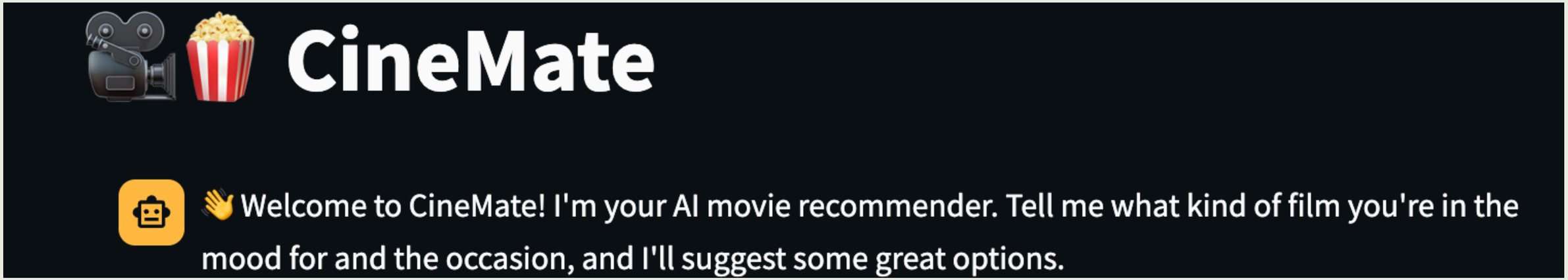
📁 add_data.py
Purpose: Prepares and uploads your movie data into the Weaviate database.
What it does:

- Loads the movie data from 1950_2024_movies_info.json
- For each movie:
  - Extracts details like title, genres, budget etc.
  - Uses the imdb_id to fetch a movie poster from the TMDB API
  - Encodes the poster image as a base64 string

(When we fetch a movie poster image (like a .jpg file from TMDB), it's a binary file — meaning it's made up of raw bytes, not readable text. But to store that image inside a text-based database like Weaviate, or to send it in a JSON payload, we need to convert it into a plain text format. That's where Base64 encoding comes in! It converts binary data (like images) into a string of A-Z, a-z, 0-9, and symbols. That string can be safely stored in a text-based format like JSON. Later, you can decode it back into an image for display.)

  - Uploads the movie and poster into a Weaviate collection called MovieDemo
- Uses the tqdm progress bar to show upload status
- ✅ Run this file once to populate your database.

📁 app.py
Purpose: The main Streamlit application for interacting with users.
What it does:
Connects to the Weaviate database using the st-weaviate-connection
Displays an intuitive sidebar for:
Selecting search mode (Keyword / Semantic / Hybrid)
Choosing a year range
Lets the user enter:
A movie type or genre (e.g., "sci-fi adventure")
An occasion (e.g., "date night")
Performs a search in Weaviate to find top matching movies
Displays:
Posters and titles from the top results
Sends the selected movies and prompt to Cohere's generative model
Cohere returns a personalized movie recommendation summary
✅ This is the interactive frontend of your app.



🚀 How to Run the App

1. *SET UP YOUR PROJECT FOLDER*

Create the following structure:

```
cine-mate/
├── helpers/
│   └── data/
│       └── 1950_2024_movies_info.json  # Your movie dataset
│
├── app.py              # The main Streamlit app
├── add_data.py         # Script to upload movies to Weaviate
├── requirements.txt    # Python packages needed
```

2. GET YOUR API KEYS

You'll need four API keys:

- WEAVIATE_API_KEY: Weaviate Console → create a free cluster.
- WEAVIATE_URL: Comes with your Weaviate cluster
- COHERE_API_KEY: Cohere Dashboard
- TMDB_API_KEY: TMDB Settings

```
pip install -r requirements.txt

python add_data.py

streamlit run app.py
```