FREE SAMPLE EDITION

Recommender Systems

From Ratings to Retrieval — Theory and Python, Built from Scratch

Yahya Laraki

Preface + Chapters 1–2 · The complete book has 13 chapters

Contents

Preface

Somebody picked the song that played after yours on the radio last night. Somebody picked the movies your streaming service suggested this weekend. Somebody decided which products land at the top of the search results you see, which posts appear first in your feed, and which books show up next to the one you just bought.

That “somebody” was, most likely, a recommender system.

Recommender systems are among the most widely deployed machine learning systems on the planet. They shape what billions of people watch, buy, read, and listen to, every single day. And yet they are strangely under-taught. If you wanted to learn how they work, you had two main doors in, and both of them were locked.

Door one was the research literature. Papers that introduce beautiful ideas but assume you already speak the field’s dialect — they talk about *matrices* (rectangular tables of numbers), *loss functions* (a score that tells an algorithm how wrong it is), and *negative sampling* (showing an algorithm counterexamples so it learns faster) as if you’d been raised on them. If you’re coming in fresh, a paper can feel less like a map and more like a wall.

Door two was the library documentation. Open-source libraries — free, shared toolkits that other people built so you don’t have to start from scratch — are wonderful things. But their docs tend to assume you already understand the theory. They’ll tell you *how* to call a function. They won’t tell you *why* you’d want to call it, or what the function is actually doing to your data, or how to tell whether the result is any good.

This book is the bridge between those two doors.

Why this book exists

Every idea in this book is paired with code you can actually run. Not fragments copied from documentation. Not pseudocode that looks nice and explains nothing. Real, runnable Python, built around a real dataset, so you can see each concept working — and break it, tweak it, and watch what changes.

That pairing is the philosophy: theory without code leaves you reciting definitions you don’t believe; code without theory leaves you copy-pasting incantations you don’t understand. Put them together and something clicks — you run a nearest-neighbors model and *see* the ratings table rearrange itself, change one line and watch your score move. The abstract becomes concrete, and the concrete becomes yours.

Who this book is for

This book assumes you know basic Python — variables, loops, functions, lists — and high-school math: algebra, a little geometry, maybe a hazy memory of what a logarithm is. It also assumes you have a vague idea of machine learning. You've probably heard that a model can learn from examples, even if you've never trained one yourself.

That is the entire prerequisite list, and I mean it.

Here is what this book does *not* assume: no calculus beyond basic intuition. No statistics. No prior expertise in machine learning. No familiarity with recommendation jargon — every term you need is defined the first time it appears, in plain language, and then reused until it feels like an old friend.

If you're a student meeting this field for the first time, a self-taught developer adding recommendations to your toolkit, or an engineer on whom a recommendation feature just landed — you're in the right place.

How to use this book

The chapters are meant to be read in order. Each one builds on the last. Chapter 1 asks what a recommender system even is. Chapter 2 turns that into a concrete problem you can measure. Chapter 3 shows how to measure it honestly. And from there, chapter by chapter, you climb from the simplest approaches to the most modern ones. Skipping ahead is possible — each chapter is self-contained enough to survive it — but the climb is easier if you don't.

The single most important instruction in this book: run the code. Typing it yourself beats reading it — the slowness is where the learning happens. When you type the line that computes a *cosine similarity* (how alike two things are) between two movies and see the answer on your own screen, you own that idea in a way no reading gives you. Every listing runs as-is on a normal laptop.

Each chapter ends with three exercises. Do them — not because I'm grading you, but because they're where you discover which parts you only *thought* you understood. Getting stumped isn't failure; it's the exercise pointing you back to the section worth re-reading.

And at the end of each chapter you'll find two small sections: **Key takeaways**, a summary of the ideas that matter most, and **Key terms**, the chapter's vocabulary. Treat them as revision tools for when your confidence wobbles ten chapters in.

Conventions

A few things to know before you begin.

All the code is Python. Not a mix of languages, not snippets translated across frameworks — one language, start to finish, so your attention stays on the ideas instead of the syntax.

The book's running example is the MovieLens-100K dataset: one hundred thousand movie ratings from real users, made public for exactly this kind of learning. It's small enough that most listings finish in seconds on a laptop. One exception, flagged in place: the SASRec model in Chapter 11 trains for minutes on a CPU (start with the chapter's reduced epoch count to check the pipeline, then run the full version). (Appendix A covers environment setup; Appendix B is a library cheat sheet.)

Every chapter follows the same shape. It opens with a short "What you'll learn" box, so you know where you're headed. Then the main body, theory and code side by side. Then three exercises. Then the key takeaways and key terms for revision. Once you've read one chapter, you know the rhythm of all thirteen.

An honesty note

The field moves fast: new papers weekly, libraries evolving, "state of the art" a moving target. The facts here were verified in September 2026 — no hype, no inflated claims, no invented numbers — but treat anything time-sensitive (library versions, benchmark scores, current leaders) as a snapshot, not a monument. When in doubt, check the sources the book points to.

One caution that trips up beginners and veterans alike: benchmark numbers across papers are *not* comparable. Different teams use different splits, filtering rules, and evaluation setups, so a higher reported number in one paper doesn't necessarily mean a better method. Read them as each paper's own story, not a league table.

And finally: this is an introduction, not a research survey. My goal is the ideas that last — the core intuitions behind collaborative filtering, factorization, ranking, and evaluation that have survived every wave of new technique. For the cutting edge, the book points to the literature. Foundations first; the frontier will still be there when you're ready.

What's ahead

The climb is ordered: Chapter 1 sets the scene (what recommenders are and where they appear), Chapter 2 turns "recommend things" into a problem a computer can solve, Chapter 3 asks how you know whether a recommender is any good, and then you build — content-

based filtering (4), nearest neighbors (5), matrix factorization (6), implicit feedback (7), hybrids with LightFM (8), learning to rank (9), deep learning (10), sequential recommendation (11), graphs and LLMs (12) — closing with recommenders in production (13).

Along the way, you'll write code for every idea on the same dataset, with the same conventions. By the end, you'll have built a recommender, evaluated it honestly, and understood its limits — a combination rarer than it should be, and more valuable.

So: open a terminal, type along, and let's begin.

Chapter 1 · What Is a Recommender System?

What you'll learn

- Why recommender systems exist and why they matter
- The two landmarks that shaped the field: Amazon's item-to-item collaborative filtering and the Netflix Prize
- The three main families of recommenders: collaborative filtering, content-based filtering, and hybrid methods
- What the rest of this book covers, chapter by chapter

The overload problem

Walk into a library with fifty million books and no shelves, no sections, no librarian: roughly the internet without help. A streaming service carries tens of thousands of titles, an online store hundreds of millions of products, a music app more songs than a lifetime holds. The good news is that everything is available. The bad news is also that everything is available.

This is **decision overload**: too many options, so people give up or settle for whatever's in front of them. A **recommender system** narrows a huge set of options down to a short list for a particular person — the row of suggestions "for you" when you open an app.

The effect is hard to overstate. Recommenders quietly shape what billions of people watch, buy, read, and listen to — which headlines you notice, which songs you hear next, which films you watch on a Friday night. For businesses, good recommendations mean engagement and sales; for users, less scrolling and more enjoying. The machinery is invisible, but its fingerprints are everywhere.

What makes the problem hard: nobody tells the system what they want. You never hand over a list of your tastes; it infers your preferences from traces — what you clicked, watched, bought, rated. Turning those traces into good suggestions is this book's subject.

Second: the data is *sparse*. Most users interact with a tiny fraction of the catalog — a few dozen movies out of thousands, a handful of products out of millions — so the system usually judges an item it has almost no direct evidence about from you. Almost every technique in this book is, at bottom, a clever way of filling in those blanks: borrowing signal from similar users, item attributes, or dataset-wide patterns.

Third: tastes are not static. What you want on a Saturday night differs from what you want on a Monday morning, and what you loved five years ago may leave you cold today. Recommenders must keep learning — a theme running from Chapter 3's evaluation to Chapter 13's production systems.

A quick map of the field

Before we go further, the vocabulary every chapter uses — the book's common language:

- A **user** is written u, and the set of all users is written U. You write $u \in U$ to mean "user u, drawn from the set of users U."
- An **item** — a movie, a product, a song, anything that can be recommended — is written i, and the set of all items is I. So $i \in I$ means "item i, drawn from the set of items I."
- The **actual rating or interaction** between user u and item i is written r_{ui}. If you gave a movie four stars, then r_ui = 4.
- The **predicted rating or score** for that user–item pair is written $\hat{r}_{ui}$ (read "r-hat"). This is the system's guess at what you would think of the item.
- $R_u(K)$ is the **top-K ranked list** for user u: the K items the system ranks highest for that user — the actual list of recommendations it shows.
- $rel(u)$ is the set of **held-out relevant items** for user u: items we know the user likes, but which we keep hidden from the system during training so we can later test whether the system can recover them.

You do not need to memorize these now; they will appear again and again, and each chapter will remind you.

Two landmarks in the field's history

Recommenders date to the early 1990s, but two events turned the field from research curiosity into one of machine learning's most important applied areas. They give you the skeleton of the whole story.

Amazon: item-to-item collaborative filtering

The first landmark is a 2003 paper by Greg Linden, Brent Smith, and Jeremy York, "Amazon.com Recommendations: Item-to-Item Collaborative Filtering" (*IEEE Internet Computing*), describing how Amazon generated its "customers who bought this also bought" recommendations.

To understand why the paper mattered, you need a little context. The natural approach is **collaborative filtering**: recommend from the behavior of *other* people. The classic version,

user-based collaborative filtering, compares users — "people similar to you also liked this." The problem was scale: comparing every user to every other user at request time is far too slow with millions of users, and Amazon needed recommendations in fractions of a second.

The paper flipped the comparison: compare *items* instead of users. For every item, find which other items tend to be bought by the same people. If you bought X, the system looks up the items most similar to X and recommends those. The expensive item–item similarity computation happens *offline*, ahead of time, so a page load needs only a quick lookup.

That shift — precomputed item–item similarities instead of on-the-fly user comparisons — made collaborative filtering scale to millions of users and items: the canonical example of **item-based** collaborative filtering, and the template for large-scale recommenders for years.

The Netflix Prize

The second landmark is the **Netflix Prize** (2006–2009): $1 million to anyone who could improve Netflix's rating-prediction error by 10%, measured by **RMSE** (root mean squared error — roughly how far predicted ratings were from actual ratings, averaged and square-rooted). Netflix released roughly 100 million ratings from about 480,000 users across roughly 18,000 movies — real-world data at a scale researchers had never seen.

Thousands of teams competed, and the contest drove a revolution in **latent-factor models** (Chapter 6), which describe each user and item with a short list of learned hidden traits — like how much a user likes action movies — and predict ratings from those. Winning entries also pioneered sophisticated **ensembles** (combinations of many models), teaching the field that combining models beats hunting for one perfect one.

But the prize has a second half: the field's founding privacy lesson. Netflix's released ratings were supposedly anonymized, but in 2008 Arvind Narayanan and Vitaly Shmatikov showed they could be de-anonymized by matching against *public* IMDb ratings. A distinctive IMDb profile under a real name could locate the same pattern in the Netflix data — attaching a name to the "anonymous" record, along with ratings the person might not have wanted public. A sequel prize, announced in 2009, was cancelled in March 2010 after a privacy lawsuit and a U.S. Federal Trade Commission inquiry.

The lesson endures: a dataset that looks anonymous can leak identities once combined with other public data. Nearly every chapter of this book touches collecting, storing, or sharing user behavior data, and the Netflix Prize de-anonymization is the cautionary tale underneath — privacy is part of the engineering, not an afterthought.

The three families

Almost every method in this book belongs to one of three families. The names are standard across the field — learn them well.

Collaborative filtering. Recommendations come from the behavior of similar users or items — the "wisdom of the crowd." **User-based** compares users: "people like you also enjoyed this." **Item-based** compares items: "people who bought this also bought that" — Amazon's 2003 approach. It's powerful because it needs no information about *what* the items are, only who interacted with what. That's also its weakness: it can't recommend a brand-new item nobody has touched, and it struggles with brand-new users — the **cold start** problem (Chapter 4). Subject of Chapter 5, Nearest Neighbors.

Content-based filtering. Recommendations come from the *attributes* of the items themselves: items carry features (genres, actors, keywords), the system builds a profile of your tastes from items you liked, then finds similar ones. If you watch a lot of action movies, you get more action movies. Subject of Chapter 4, Content-Based Filtering.

Hybrid methods. These combine collaborative and content-based approaches — the crowd's wisdom plus the items' attributes — hedging against each one's weaknesses. The workhorses of many real systems. Subject of Chapter 8, Hybrid Models with LightFM.

Beyond these three families, the book goes further into modern territory. Chapter 10, Deep Learning for Recommendation, covers what neural networks — flexible models inspired loosely by the brain's structure — bring to the table. Chapter 11, Sequential Recommendation, covers systems that pay attention to the *order* of your actions: what you clicked yesterday matters for what you should see today. Chapter 12, Graphs and LLMs, covers two frontiers — graph methods, which treat users and items as connected networks, and large language models, which read and write text about items and users. And Chapter 13, Recommenders in Production, tackles the engineering reality of running all of this at scale: speed, reliability, and keeping the system fresh as tastes change.

Your roadmap

The full arc, one sentence per chapter and appendix:

- **Chapter 2, Framing the Problem** — turns "recommend things" into precise tasks: what's predicted, what data is used, how explicit ratings differ from implicit signals like clicks.
- **Chapter 3, Evaluation** — measures whether recommendations are any good, using held-out data and metrics built on *rel(u)* and *R_u(K)*, before a system ever touches real users.
- **Chapter 4, Content-Based Filtering** — builds recommenders from item attributes like genres and keywords, and confronts the cold start: what to recommend when you know nothing about the user or item.
- **Chapter 5, Nearest Neighbors** — implements collaborative filtering: find similar users or items and let their behavior decide.

- **Chapter 6, Matrix Factorization and SVD** — the latent-factor revolution from the Netflix Prize: hidden user and item traits that explain the ratings.
- **Chapter 7, Implicit Feedback** — most systems never see star ratings, only clicks, views, and purchases: how to learn from that one-sided data.
- **Chapter 8, Hybrid Models with LightFM** — combines collaborative and content-based signals in a single model with the LightFM library.
- **Chapter 9, Learning to Rank** — from predicting ratings to the real goal: optimizing the *ordering* of the top-K list directly.
- **Chapter 10, Deep Learning for Recommendation** — neural networks for recommendation: embeddings, deep collaborative models, what they add beyond matrix factorization.
- **Chapter 11, Sequential Recommendation** — treats behavior as a sequence: what you did last week and last hour both shape what you see next.
- **Chapter 12, Graphs and LLMs** — graph neural networks plus large language models: network structure and text understanding for recommendation.
- **Chapter 13, Recommenders in Production** — the engineering of real systems: latency, scale, experimentation, monitoring, and keeping recommendations fresh.
- **Appendix A, Environment Setup** — installing packages and preparing the datasets used throughout the book.
- **Appendix B, Library Cheat Sheet** — quick reference for the libraries used in the chapters: imports and function signatures at a glance.

One note before you dive in: every chapter pairs theory with runnable Python code — except this one. Chapter 1 is deliberately code-free: vocabulary, history, big picture first. From Chapter 2 onward you'll write code that actually recommends things, and by the end you'll have built recommenders from every family here, evaluated them honestly, and understood what makes them work — and risky.

Welcome to the field.

Exercises

1. **Spot the family.** Pick two recommendation features you use regularly (for example, a "recommended for you" row on a streaming service and "frequently bought together" on an online store). For each one, decide whether it is collaborative filtering, content-based filtering, or a hybrid — and write one sentence justifying your choice based on what information the feature seems to use.
2. **Explain the Amazon shift.** In your own words, explain why computing item–item similarities offline made Amazon's recommendations faster at request time than

comparing users on the fly. What does “offline” mean here, and why does doing the work ahead of time help?

3. **The privacy thought experiment.** Netflix released ratings it believed were anonymous. Using the Narayanan–Shmatikov result, explain in two or three sentences how a public IMDb profile could be used to attach a name to an “anonymous” Netflix record. Then name one precaution you would take if you were publishing a similar dataset today.

Key takeaways

- Recommender systems exist to solve decision overload: narrowing huge catalogs down to short, personalized lists of suggestions.
- Collaborative filtering recommends from the behavior of similar users or items — the “wisdom of the crowd” — in user-based and item-based flavors.
- Content-based filtering recommends from item attributes, and hybrid methods combine both approaches.
- Amazon’s 2003 item-to-item paper showed how to make collaborative filtering scale: precompute item–item similarities offline instead of comparing users at request time.
- The Netflix Prize (2006–2009) offered $1M for a 10% RMSE improvement and drove the latent-factor revolution; its sequel was cancelled in 2010 after the prize data was de-anonymized via public IMDb ratings.
- The book’s notation — $u \in U$, $i \in I$, r_{ui}, $\hat{r}_{ui}$, $R_u(K)$, $rel(u)$ — is the shared language every chapter uses.
- Privacy is an engineering concern, not an afterthought: even “anonymized” data can leak identities when combined with public information.

Key terms

- **Recommender system:** software that narrows a large set of options down to a short, personalized list of suggestions for a particular user.
- **Decision overload:** the difficulty of choosing when faced with too many options.
- **User ($u \in U$):** a person receiving recommendations; U is the set of all users.
- **Item ($i \in I$):** anything that can be recommended — a movie, product, song; I is the set of all items.
- **Rating / interaction (r_{ui}):** the actual recorded value for user u and item i, such as a star rating or a click.
- **Predicted rating / score ($\hat{r}_{ui}$):** the system’s estimate of r_{ui} — its guess at what the user would think of the item.

- **Top-K ranked list ($R_u(K)$):** the K items the system ranks highest for user u — the recommendation list it shows.
- **Held-out relevant items ($rel(u)$):** items known to be relevant to user u, hidden from the system during training so they can be used to test it.
- **Collaborative filtering:** recommendations based on the behavior of similar users or items — the "wisdom of the crowd."
- **User-based collaborative filtering:** a collaborative approach that compares users: "people like you also liked this."
- **Item-based collaborative filtering:** a collaborative approach that compares items: "people who bought this also bought that."
- **Content-based filtering:** recommendations based on item attributes (e.g., genres, keywords) matched to a profile of the user's tastes.
- **Hybrid methods:** recommenders that combine collaborative and content-based approaches.
- **Latent-factor model:** a method that describes each user and item with a short list of learned hidden traits and predicts ratings from them.
- **Ensemble:** a combination of multiple models whose predictions are merged, often improving over any single model.
- **RMSE (root mean squared error):** a metric measuring how far predicted ratings are, on average, from actual ratings; lower is better.
- **Cold start:** the problem of recommending for a brand-new user or item with no recorded interactions.
- **De-anonymization:** re-attaching identities to supposedly anonymous data, e.g., by matching rating patterns against public profiles.

Chapter 2 · Framing the Problem

Before you can build a recommender, you need to know what problem you're solving. This chapter turns "suggesting things people like" into precise definitions you can write code against: the two kinds of feedback, the two prediction tasks, the matrix that organizes everything, the cold-start problem, and a tour of MovieLens — the dataset this book runs on.

What you'll learn

- The difference between explicit and implicit feedback, and why implicit is plentiful but tricky
- Rating prediction vs. ranking — and why the product almost always ships a ranked list
- What the user–item matrix is, and what sparsity means (with real numbers from MovieLens)
- The cold-start problem and its standard mitigations
- Which MovieLens editions exist, and how to download and load MovieLens-100K yourself

Explicit vs implicit feedback

Every recommender learns from feedback — evidence about what a user likes — in two flavors.

Explicit feedback is deliberately given: star ratings, thumbs up/down, reviews. The user tells you what they think, in terms you chose. High-quality but scarce — most people watch a movie and never rate it.

Implicit feedback is observed rather than asked for: clicks, page views, purchases, how long someone watched before abandoning a video. Abundant — every visit to your store generates it — but ambiguous. A click is not a compliment. Someone clicked on a video and left after eight seconds: was that interest, or regret? Someone bought a gift for their mother-in-law: does that mean they personally like what they bought? Implicit feedback tells you what people *did*, not what they *felt*, and the two are not the same.

That's the central tension of implicit feedback: far more data, each point saying less. A 5-star rating is unambiguous — the user loved it. A view is a shrug. Squeezing meaning out of shrugs is a harder, subtler problem than rating prediction — the subject of Chapter 7.

A canonical example is the RetailRocket e-commerce dataset (Kaggle: `retailrocket/ecommerce-dataset`): roughly 2.7 million events from about 1.4 million visitors across about 417,000 items — views, add-to-carts, transactions, nobody rating anything. Pure behavior, and a standard implicit-feedback benchmark. Note the hierarchy of intent: a purchase says more than an add-to-cart, which says more than a view.

Most of this book uses explicit ratings — the cleanest way to learn the core ideas — but in production, implicit feedback is the data you'll mostly have.

Rating prediction vs ranking

Given the feedback, what should the model predict? Two natural answers define the two classic tasks of recommendation.

Rating prediction asks: what rating would user u give item i? You predict **r̂_ui** (read "r-hat-u-i"), an estimate of the true rating **r_ui** they'd give if they consumed the item. This is the Netflix Prize framing — literally "how many stars?" — and chapters 5 and 6 grew out of that lineage.

Ranking asks a different question: of all the items, which K should I show this user? You produce **R_u(K)** — the top-K ranked list for user u. No stars, no numbers; just an ordering, with the best guesses first.

Here's what matters: the product ships a ranked list. No homepage shows "predicted rating: 4.31 stars" under each film; it shows *The Godfather*, then *Goodfellas*, then *Heat*, in a chosen order. Rating prediction was a convenient mathematical proxy, but the user sees a ranking — which is why the field moved from prediction to ranking. Predicting r̂_ui is a fine intermediate step; producing a good R_u(K) is the goal.

Note too that the two tasks can disagree. A model might predict r̂_ui well on average yet produce a bad top-K list, because ranking cares about the *order* of the top items, not absolute numbers. Consistently half a star too optimistic? Fine ratings, broken ordering — and the ordering is what the user experiences.

This distinction drives everything downstream: Chapter 3 shows how to measure a ranking properly — precision@K, recall@K computed against **rel(u)**, the held-out relevant items — and Chapter 9 shows how to optimize for the ranking directly instead of hoping good ratings become good rankings.

The user–item matrix

The central object of classic recommendation is the **user–item matrix**: a table with one row per user and one column per item, entry (u, i) = **r_ui** — the rating user u gave item i (or a mark of an observed interaction, for implicit feedback).

```
          Item 1   Item 2   Item 3   Item 4   ...
User 1       5        ?        3        ?       ...
User 2       ?        4        ?        2       ...
User 3       1        ?        ?        ?       ...
   ...
```

Most entries are missing — no user has rated most items. The **sparsity** of the matrix is the fraction missing:

sparsity = 1 − (observed ratings) / (number of users × number of items)

A dense matrix has sparsity near 0; a recommendation matrix is the opposite. Concretely, with MovieLens-100K (100,000 ratings, 943 users, 1,682 movies): 943 × 1,682 = 1,586,126 possible cells, of which 100,000 are filled:

- filled = 100,000 / 1,586,126 ≈ 6.3%
- sparsity ≈ 93.7%

So even a tiny, carefully curated dataset is over 93% empty — and real-world matrices are far sparser, typically well under 1% filled.

That sparsity is the defining difficulty: inferring what a user would like from scattered data points in a mostly blank matrix. Almost every technique in this book is, at heart, a strategy for filling in those blanks — borrowing patterns from other users' rows, other items' columns, genre information, sequences of behavior — because the matrix itself never gives you enough.

The cold-start problem

New users and items break that assumption. This is the **cold-start problem**, in two forms:

- **New users.** A user signs up and has rated nothing. Their row of the matrix is entirely blank. What do you recommend?
- **New items.** A new movie enters the catalog with no ratings yet. Its column is entirely blank. How is it ever discovered?

Standard mitigations, briefly:

1. **Popularity fallback.** Recommend what's popular with everyone — top-rated movies, bestsellers — until you know better. Crude, but it always works.
2. **Asking for preferences.** Onboarding quizzes: "pick three genres you like," "rate these ten movies." It costs the user effort, so keep it short.
3. **Content features.** Use information about the items themselves — genres, descriptions, tags. If a new user loves sci-fi, recommend sci-fi movies immediately, no history required: **content-based filtering** (Chapter 4), which combined with collaborative methods gives **hybrid models** (Chapter 8).

Cold start never fully goes away — there's always a new user or item — but these mitigations keep a system useful while it gathers data.

A tour of MovieLens, our running dataset

Almost every example in this book runs on MovieLens, the long-running dataset from the **GroupLens** lab at the University of Minnesota — the MNIST of recommendation: small enough for memory, real enough to teach real lessons, standardized enough to compare across decades.

MovieLens comes in several editions:

Edition	Era	Notes
32M	collected 10/2023, released 5/2024	GroupLens's current recommendation for new research
25M	—	—
20M	—	still the most common in published papers
10M	—	—
1M	—	—
100K	1997–98, released 4/1998	the classic teaching edition; what this book uses
latest-small / latest-full	frozen since 9/2018	fine for tutorials; not for reporting research results

All MovieLens editions share the same shape: explicit ratings on a 1–5 star scale. The classic editions guarantee every user rated at least 20 movies — a deliberate minimum that keeps no user row *completely* empty, keeping the data friendly to collaborative methods while still sparse overall. One edition difference matters: only 100K ships user demographics in

`u.user` (age, gender, occupation, zip code); later editions carry none. We'll use 100K throughout — it downloads in seconds and runs anywhere, and for the fundamentals, 100,000 ratings teach the same lessons as 32 million.

Downloading and loading MovieLens-100K

The 100K edition lives at `https://files.grouplens.org/datasets/movielens/ml-100k.zip` (about 5 MB). Note the older layout: newer editions use `ratings.csv` and `movies.csv`, but ml-100k uses tab- and pipe-separated files with no headers — which matters for the `read_csv` calls:

```
import urllib.request
import zipfile
import pandas as pd

# Download (~5 MB) and unzip
url = 'https://files.grouplens.org/datasets/movielens/ml-100k.zip'
urllib.request.urlretrieve(url, '/tmp/ml-100k.zip')
with zipfile.ZipFile('/tmp/ml-100k.zip') as z:
    z.extractall('/tmp')

# ml-100k/u.data — tab-separated, NO header:
# user_id, item_id, rating (1-5), timestamp (Unix)
ratings = pd.read_csv('/tmp/ml-100k/u.data', sep='\t',
                      names=['user_id', 'item_id', 'rating',
'timestamp'])

# ml-100k/u.item — pipe-separated, NO header:
# movie_id | title | release_date | video_release_date | imdb_url | 19
genre flags
# Titles contain accented characters, so read with latin-1.
item_cols = ['movie_id', 'title', 'release_date', 'video_release_date',
'imdb_url']
genre_names = ['unknown', 'Action', 'Adventure', 'Animation',
"Children's",
               'Comedy', 'Crime', 'Documentary', 'Drama', 'Fantasy',
               'Film-Noir', 'Horror', 'Musical', 'Mystery', 'Romance',
               'Sci-Fi', 'Thriller', 'War', 'Western']
items = pd.read_csv('/tmp/ml-100k/u.item', sep='|', encoding='latin-1',
                    names=item_cols + genre_names, usecols=[0, 1] +
list(range(5, 24)))
print(ratings.head())
print(items.head())
```

On the genre columns: `u.item` carries 19 binary flags per movie (a movie can have several); the genre names and order come from `u.genre`, a small `name|index` file — the

list above is exactly that order. The flags live in columns 5–23 of `u.item`, hence `usecols=[0, 1] + list(range(5, 24))`.

There's also `u.user` (pipe-separated `user_id|age|gender|occupation|zip_code`) with the demographics 100K shipped, plus ready-made 80/20 train/test splits `u1.base`/`u1.test` through `u5.base`/`u5.test` — five folds, same format as `u.data`, handy if you want a fixed, reproducible split of your own.

First look at the data

First rule of a new dataset: look at it.

```
print(ratings.shape)        # how many ratings, how many columns
print(ratings.describe())   # mean, spread, min/max of ratings and ids
print(ratings['rating'].value_counts().sort_index())
```

You should see 100,000 rows and 4 columns; ratings 1–5 with mean around 3.5 and standard deviation just over 1; a top-heavy distribution — 4s most common, 5s close behind, 1s and 2s rare. Typical: people mostly bother to rate things they like, so explicit ratings skew upward. Remember that whenever you average them.

Now the sparsity computation:

```
n_users = ratings['user_id'].nunique()   # 943
n_items = ratings['item_id'].nunique()   # 1682
possible = n_users * n_items
filled = len(ratings)
sparsity = 1 - filled / possible
print(f'users: {n_users}, items: {n_items}, possible cells:
{possible}')
print(f'filled: {filled / possible:.1%}  ->  sparsity: {sparsity:.1%}')
```

That prints roughly 6.3% filled — 93.7% sparse. By recommendation standards this is *dense*; production matrices are typically well under 1% filled.

To make the matrix concrete: a `pivot_table` builds exactly the matrix from the diagram — rows users, columns items, entries r_ui, everything else missing.

```
# A small slice of the user-item matrix: users 1..8, first 8 items
R = ratings.pivot_table(index='user_id', columns='item_id',
                        values='rating', aggfunc='mean')
print(R.iloc[:8, :8])
```

The printed frame is mostly `NaN` — sparsity staring back at you. For readable columns, join against the titles from `u.item`:

```
title = dict(zip(items['movie_id'], items['title']))
small = R.iloc[:8, :8].rename(columns=title)
print(small)
```

Now the columns read *Toy Story (1995)*, *GoldenEye (1995)* instead of anonymous ids. One more useful one-liner:

```
# How many movies carry each genre flag?
print(items[genre_names].sum().sort_values(ascending=False).head())
```

Drama and Comedy dominate, as you'd expect. These genre flags are content features — exactly what Chapter 4 uses against the new-item cold start: an unknown movie still has a genre, and its genre says something about who might like it.

Where this leaves us

The problem, on one page: recommenders learn from **explicit** feedback (ratings, thumbs) or **implicit** feedback (clicks, views, purchases — abundant but ambiguous). Their job is **rating prediction** (estimate r̂_ui) or **ranking** (produce R_u(K)) — and since the product ships a ranked list, ranking is where the field is headed. The data lives in a **user–item matrix** that's overwhelmingly **sparse** (93.7% empty in MovieLens-100K; far emptier in the wild). **Cold start** (new users, new items) means we can't always rely on the matrix — hence popularity fallbacks, onboarding questions, and content features. And we have MovieLens-100K, loaded and inspected, to test every idea in the rest of the book.

The natural next question: how do we know whether our recommendations are any good? That's Chapter 3 — Evaluation.

Exercises

1. **Download and explore.** Download ml-100k with the code in this chapter and print the ten most-rated movies, joining the titles from `u.item`. What are the top three?
2. **Sparsity sense-check.** The code computes sparsity from `nunique()` counts of users and items *present in the data*. Construct an argument for why this is the right choice (and what would go wrong if you instead assumed user ids run from 1 to max without gaps).
3. **Cold-start design.** A new user joins your music recommender with zero history. Propose a concrete three-question onboarding flow (what you'd ask, and how you'd turn the answers into an initial ranked list R_u(10)). One paragraph.

Key takeaways

- Explicit feedback (ratings, thumbs) is deliberate and trustworthy but scarce; implicit feedback (clicks, views, purchases) is abundant but ambiguous — a click is not a compliment.
- Rating prediction estimates r̂_ui; ranking produces R_u(K). The product ships a ranked list, so ranking is what users actually see.
- The user–item matrix has rows u ∈ U, columns i ∈ I, and entries r_ui. Its sparsity is 1 − observed/(users × items): ~93.7% for MovieLens-100K, far higher in production.
- Cold start (new users, new items) is mitigated with popularity fallbacks, preference onboarding, and content features.
- MovieLens editions: 100K (1997–98, released 4/1998), 1M, 10M, 20M, 25M, 32M (collected 10/2023, released 5/2024 — GroupLens's current recommendation for new research; 20M remains the most common in published papers), and latest-small/latest-full (frozen since 9/2018, tutorials only). All are 5-star explicit ratings; the classic editions guarantee ≥20 ratings per user, and only the 100K edition ships user demographics.
- ml-100k's verified layout: `u.data` is tab-separated with no header (user_id, item_id, rating 1–5, timestamp); `u.item` is pipe-separated (movie_id, title, …, 19 genre flags, read with `encoding='latin-1'`); `u.genre` gives the genre name order; ready-made 80/20 splits live in `u1.base` / `u1.test` … `u5.base` / `u5.test`.

Key terms

- **Explicit feedback:** deliberately given ratings or opinions (stars, thumbs up/down).
- **Implicit feedback:** observed behavior (clicks, views, purchases, watch time) used as a signal of interest.
- **Rating prediction:** estimating r̂_ui, the rating user u would give item i.
- **Ranking:** producing R_u(K), the top-K ranked list for user u.
- **User–item matrix:** rows are users u ∈ U, columns are items i ∈ I, entries are r_ui.
- **Sparsity:** the fraction of the user–item matrix that is missing: 1 − observed/(|U| × |I|).
- **Cold start:** the problem of recommending for new users (no history) or new items (no interactions).
- **rel(u):** the held-out relevant items for user u, used when evaluating a ranking.
- **MovieLens:** GroupLens's long-running explicit-rating movie dataset; the book's running example.

Get the Full Book

You've just read the opening of **Recommender Systems: From Ratings to Retrieval** — the complete beginner-friendly guide with 13 chapters, runnable Python for every concept, real MovieLens walkthroughs, and coverage from nearest neighbors and matrix factorization through two-tower retrieval, sequential models, graphs, LLMs, and production systems.

Get the full 142-page book at aifrontierpost.com — available as PDF, ePub, and Kindle editions.