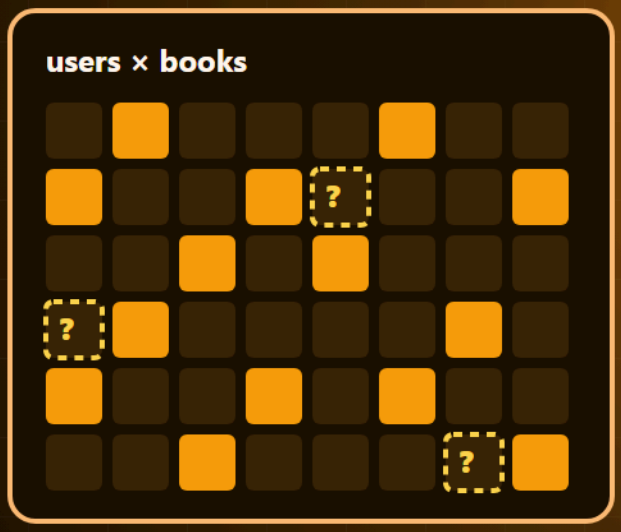



```python
class BPRMF(Recommender):
  def score(self, users):
    U = self.user_emb(users)
    return U @ V.T + b

# ranking loss
loss = -logsigmoid(
  s_pos - s_neg).mean()
loss.backward()
```



A HANDS-ON GUIDE · PYTHON + PYTORCH

Recommender Systems in Practice

Build, evaluate, and serve recommendations with Python and PyTorch

Baselines · Ranking metrics · Content-based · kNN
Matrix factorization · ALS · BPR · Two-tower · Ranking
Cold start · Diversity · FastAPI serving · Monitoring

DATA · EVALUATION · MODELS · RANKING · PRODUCTION

THENEXBOOKS

Recommender Systems in Practice

Build, Evaluate, and Serve Recommendations with Python and PyTorch
From popularity baselines to two-tower retrieval and a production API — built step by step

First Edition

THENEXBOOKS

Recommender Systems in Practice
Build, Evaluate, and Serve Recommendations with Python and PyTorch

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Python, PyTorch, scikit-learn, FastAPI, and the other libraries used in this book change over time; APIs, defaults, and printed output can differ between versions. Metric values in this book were produced on a CPU with the stated library versions and fixed random seeds; your numbers can differ slightly on other machines and versions. Always test recommendation changes with real users before relying on them for business decisions.

Python is a trademark of the Python Software Foundation. PyTorch is a trademark of The Linux Foundation. Other product names mentioned in this book may be trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by any of these organizations. The sentence-embedding model used in Chapter 11 is an openly licensed model; check the license of any model before using it in production.

Pagewell Books, the online bookstore used as the running example in this book, is fictional, as are its staff, customers, authors, book titles, and all sample data. The dataset is generated synthetically from a fixed random seed by code printed in this book; any resemblance to real people, authors, or books is coincidental.

Preface

Open almost any app you use daily and something is recommending something to you: the next video, a product that goes with the one in your cart, a song for a rainy afternoon, a book you might enjoy after the one you just finished. These systems look like magic from the outside. From the inside they are a pipeline of fairly understandable pieces — interaction logs, a way to split and evaluate them, a handful of models that turn past behavior into scores, a ranking step, some business rules, and a service that returns a list in a few milliseconds.

Most tutorials stop after training one model on a public ratings dataset and printing a single error number. Real recommendation work is different. You have to decide what "relevant" means for your business, avoid leaking the future into your test set, beat a popularity baseline that is surprisingly hard to beat, handle new users and new items, keep the lists from collapsing into the same twenty bestsellers, serve results quickly, and notice when the system quietly gets worse. This book covers that whole journey.

We build one recommender system from the ground up, in plain Python, NumPy, scikit-learn, and PyTorch. Every model is implemented from scratch so you can see exactly how it works, every metric is computed by code you write, and every number in the book comes from running that code on a dataset you generate yourself with a fixed seed. When we say a model is better, we show you the table.

Who This Book Is For

- **Python developers** who are comfortable with functions, classes, lists, and dictionaries and want to build their first real recommender system.
- **Data scientists and analysts** who know pandas and basic machine learning and need to move from "a model in a notebook" to recommendations people actually see.
- **Backend and product engineers** who are asked to add "recommended for you" or "customers also bought" to an existing product and want to understand the trade-offs.
- **Students** who want a practical companion to the theory of collaborative filtering, matrix factorization, and learning to rank.

You do not need prior experience with recommender systems or deep learning. Basic NumPy and pandas help; each mathematical idea is introduced one formula at a time, followed by plain words and code.

What You Will Learn

- How recommendation problems are framed: explicit ratings vs implicit events, the user–item matrix, sparsity, and the long tail.
- How to build a realistic synthetic dataset with a seeded generator, and how to split interaction data by time without leakage.

- How to evaluate honestly: precision, recall, hit rate, MAP, and NDCG at k, plus coverage, diversity, novelty, and popularity bias, and how offline results relate to A/B tests.
- How to build baselines and classic models: popularity and trending lists, TF-IDF and embedding content models, user-based and item-based k-nearest neighbors, and hybrids.
- How matrix factorization works: SGD in NumPy, PyTorch models with biases, implicit ALS, and Bayesian Personalized Ranking with negative sampling.
- How modern pipelines are built: sequence-aware recommendations, two-tower retrieval with in-batch softmax, a learned ranker, and cold-start strategies.
- How to ship: diversity re-ranking and business rules, serving architecture and nearest-neighbor search, a FastAPI service with tests, monitoring, feedback loops, and retraining.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) introduces recommender systems, interaction data, the Pagewell dataset generator, temporal splits, and baselines. **Part II — Evaluation** (Chapters 6–9) covers ranking metrics, beyond-accuracy metrics, a reusable evaluation harness, and online experiments. **Part III — Content and Neighbors** (Chapters 10–14) builds content-based recommenders with TF-IDF and text embeddings, user-based and item-based collaborative filtering, and hybrids. **Part IV — Matrix Factorization** (Chapters 15–19) implements factor models in NumPy and PyTorch, implicit ALS, and BPR, and shows how to tune and debug them. **Part V — Deep and Sequential Models** (Chapters 20–23) covers sequence models, two-tower retrieval, the ranking stage, and cold start. **Part VI — Production** (Chapters 24–28) covers re-ranking for diversity and business rules, serving architecture, a recommendation API, monitoring and retraining, and a complete case study. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: Pagewell Books

Throughout the book we build the recommender system for **Pagewell Books**, a fictional online bookstore with 2,000 customers, a catalog of 1,200 books across twelve genres, and a full year of browsing, wishlist, purchase, and rating events. Pagewell wants a "Recommended for you" row on its home page, "Readers who bought this also bought" on every book page, sensible suggestions for brand-new customers and brand-new books, and a weekly email that does not send everyone the same bestsellers. The data comes from a seeded generator that you write in Chapter 3, so your numbers match ours, and every model in the book is scored with the same split and the same evaluation code, so the comparisons are fair.

Versions and Conventions

The examples use **Python 3.12** with recent versions of NumPy, pandas, SciPy, scikit-learn, and **PyTorch 2.x**, and were tested on an ordinary **CPU** with PyTorch limited to two threads. Every training run in the book finishes in seconds to a few minutes on a laptop. GPU execution is discussed where it matters, but it was not part of this book's tests. Exact metric values can vary slightly with your CPU, thread count, and library versions even with fixed seeds; when they do, the comparisons and conclusions should still hold.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version differences, or a useful aside.

> **WARNING**
>
> A common mistake or something that can mislead your evaluation, waste compute, or hurt users.

Code appears in `monospace`; long lines are wrapped to fit the page. File names are shown above each listing, and program output is shown under an **Output** caption. Metric tables use four decimal places, and lists are evaluated at `k = 10` unless a section says otherwise. The project's package is called `pagewell_rec`; Appendix B lists every module and function.

Contents

Preface

PART I — FOUNDATIONS

1 What Recommender Systems Do: Pagewell Books and the Shape of the Problem

2 Interaction Data: Ratings, Implicit Events, and the User–Item Matrix

3 The Pagewell Dataset: A Seeded Synthetic Generator

4 Train/Test Splits Without Leakage: Temporal Splits and Evaluation Sets

5 Baselines That Are Hard to Beat: Random, Popularity, and Trending

PART II — EVALUATION

6 Ranking Metrics: Precision, Recall, Hit Rate, MAP, and NDCG at k

7 Beyond Accuracy: Coverage, Diversity, Novelty, and Popularity Bias

8 An Evaluation Harness: Comparing Models Fairly and Reproducibly

9 Offline vs Online: A/B Tests, Interleaving, and Guardrail Metrics

PART III — CONTENT AND NEIGHBORS

10 Content-Based Recommendations with TF-IDF

11 Text Embeddings for Content-Based Recommendations

12 User-Based Collaborative Filtering with k-Nearest Neighbors

13 Item-Based Collaborative Filtering: Readers Who Bought This Also Bought

14 Hybrid Recommenders: Blending Content, Neighbors, and Popularity

PART IV — MATRIX FACTORIZATION

15 Matrix Factorization from Scratch in NumPy

16 Matrix Factorization in PyTorch: Embeddings, Biases, and Training Loops

17 Implicit Feedback and Alternating Least Squares

18 Bayesian Personalized Ranking and Negative Sampling

19 Tuning and Debugging Factorization Models

PART V — DEEP AND SEQUENTIAL MODELS

20 Sequence and Session Recommendations: From Markov Chains to Next-Item Models

21 Two-Tower Retrieval Models in PyTorch

22 The Ranking Stage: Candidate Features and a Learned Ranker

23 Cold Start: New Users, New Books, and Fallbacks

PART VI — PRODUCTION

24 Diversity, Fairness, and Business Rules: Re-Ranking the Final List

25 Serving Architecture: Batch Precompute, Real-Time APIs, and Nearest-Neighbor Search

26 A Recommendation API with FastAPI

27 Monitoring, Feedback Loops, and Retraining

28 Case Study: The Pagewell Recommender from Data to Production

APPENDICES

A Recommender Systems Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

What recommender systems are for and how they fail, the shape of interaction data, a seeded synthetic dataset for Pagewell Books that you generate yourself, splitting data by time without leaking the future, and the simple baselines every model must beat.

1 · Recommender Systems in Practice

2 · Interaction Data

3 · The Pagewell Dataset

4 · Splits Without Leakage

5 · Baselines That Are Hard to Beat

CHAPTER 1 · PART I — FOUNDATIONS

What Recommender Systems Do: Pagewell Books and the Shape of the Problem

A recommender system decides which handful of items, out of thousands, a person sees next. In this chapter we look at what that job really involves, the main families of techniques, the pipeline that production systems share, the ways they quietly fail, and the fictional bookstore, Pagewell Books, whose recommender we will build from raw data to a running API over the course of the book.

A Bookstore With Too Many Books

Imagine walking into a physical bookshop. Near the door there is a table of new releases, a shelf of staff picks, and maybe a small sign that says "If you liked last year's prize winner, try these." The shop has limited space, so somebody has already made thousands of tiny decisions for you: which books deserve a place on the front table, which ones go spine-out on a high shelf, and which ones are not stocked at all. Those decisions are recommendations, even though nobody calls them that.

An online store removes the shelf-space limit but keeps the attention limit. A web page can list every book in the catalog, yet a visitor will look at the first row, maybe the second, and then leave or search. The screen is the new front table, and it holds perhaps ten books. Someone, or something, has to decide which ten. When that decision is the same for everybody, the store is merchandising. When it changes with the person looking, the store is running a **recommender system**.

That is the whole problem in one sentence: *given a person and a moment, choose a short, ordered list of items from a large catalog that this person is likely to find worth their attention*. Everything else in this book, from sparse matrices to two-tower neural networks to FastAPI endpoints, is machinery for doing that one thing well, measuring how well we did it, and doing it again tomorrow with fresh data.

Notice three words in that sentence. **Short**: we produce a list of k items, usually 5 to 20, not a score for every item. **Ordered**: position matters, because the first slot gets more attention than the tenth. **Likely**: we never know for certain what someone wants; we estimate it from evidence, mostly the traces that people leave when they browse and buy. Keep these three words in mind, because each one changes how we train and evaluate models later.

Where Recommendations Appear

Recommendations show up in more places than most people notice. In a typical online store you will find at least these surfaces:

- **A personalized home row**, often titled "Recommended for you." It is the most visible placement and the one most people mean when they say "the recommender." It should reflect the visitor's own history.
- **Item-to-item lists** on a product page, such as "Customers who bought this also bought." The anchor is the item being viewed, not the person, so this list can work even for anonymous visitors.
- **New-visitor suggestions** for people with no history yet. The system has almost nothing to go on, so it leans on what is popular, what is trending, or what a short onboarding question reveals.
- **Messages**, such as a weekly email with books picked for each subscriber. Email lists are generated in advance, read later, and judged as a whole, so variety matters more than in a single row.
- **Search ranking**. When a query matches hundreds of books, the order of results is itself a recommendation. Personalizing that order uses the same ideas we develop here, combined with text relevance.

Each surface has a different anchor (a person, an item, a query, nothing at all), a different list length, a different moment in the customer's visit, and a different definition of success. A single model rarely serves all of them equally well, which is why real systems are usually several models and rules working together; Chapter 28 chooses what to ship for each Pagewell placement.

The Core Idea: Predict, Rank, Show k

Strip away the details and nearly every recommender follows the same three steps. First, it assigns a **score** to each candidate item for a given user, where a higher score means "more likely to be wanted." Second, it **ranks** the candidates by score. Third, it **shows the top k**, after removing anything that should not appear, such as books the customer already owns.

```
for a user u:
    score(u, i)  for every candidate book i      # how likely is u to want i?
    remove books u already bought                # business rule
    sort by score, highest first
    return the first k books                     # k = 10 for a home row
```

The interesting part is entirely inside `score(u, i)`. A popularity model gives every user the same score for book *i*: the number of people who bought it. A content-based model scores a book by how similar its description is to the books the user already bought. A collaborative model scores a book by what similar users bought, or by how the book's learned "taste vector" lines up with the user's. A learned ranker combines many such signals with a machine learning model. The surrounding loop barely changes.

This framing also clarifies what we are *not* doing. We are not predicting a single correct answer the way a classifier predicts whether an email is spam. A customer who buys one book from our list of ten has been served well even though nine suggestions went unused. We are also not trying to reproduce ratings to two decimal places; as Chapter 15 will show on our data, a model can predict star ratings closely and still produce poor lists. The target is a *useful ordering*, and the metrics in Part II are designed around that.

> **NOTE**
> Throughout the book "item" and "book" mean the same thing, and "user" and "customer" mean the same thing. The techniques apply unchanged to films, songs, recipes, job ads, or news articles; only the data and the business rules differ.

The Main Families of Recommenders

There are many algorithms, but they fall into a few families based on what evidence they use. Knowing the families helps you read papers and library documentation, and it helps you reason about which model to try when a new placement comes along.

Family	Evidence it uses	Typical strength	Typical weakness
Non-personalized	Counts of purchases or views, recent trends	Simple, robust, works for new users	Same list for everyone
Content-based	Item attributes and text	Handles brand-new items, easy to explain	Narrow, repetitive lists
Collaborative filtering	Who bought what	Finds non-obvious connections	Needs history for users and items
Hybrid	Several of the above	Covers each other's gaps	More parts to tune
Learned ranking	Many features per candidate	Combines signals and context	Needs labels, features, and care

Non-personalized recommenders

The simplest recommender shows the best-selling items to everybody. A close cousin, **trending**, counts only recent activity, for example the last 30 days, so that new hits rise quickly and old favorites fade. These models sound too simple to matter, yet they are some of the hardest baselines to beat, because people really do buy popular things more often. Chapter 5 builds both and turns them into the yardstick every later model must clear.

Content-based recommenders

A content-based model looks at what items are *about*. If a reader bought three cozy mysteries, we can recommend more books whose descriptions, genres, and authors resemble those three. The method needs no other customers at all, which means it can place a book that was released this morning and

has zero sales. Its weakness is the flip side of its strength: it only knows "more of the same," and it cannot tell a beloved mystery from a dull one if their descriptions look alike. Chapters 10 and 11 build content models with TF-IDF and with sentence embeddings.

Collaborative filtering

Collaborative filtering ignores what items are about and looks only at behavior: people who bought this also bought that. The idea is old and powerful. If many readers of a particular historical novel also bought a certain travel memoir, that link is useful even if the two books share no words in their descriptions. Neighborhood methods (Chapters 12 and 13) compute similarities between users or between items directly. Matrix factorization methods (Part IV) learn a short vector of hidden "taste" numbers for every user and every book, so that a dot product of the two vectors predicts interest. Collaborative models need history, so they struggle with new users and new books, a problem known as **cold start**.

Hybrids and learned ranking

Real systems mix families. A **hybrid** might blend a collaborative score with a content score and a small popularity term (Chapter 14). A **two-stage** system first retrieves a few hundred candidates cheaply, then applies a heavier **learned ranker** that looks at many features of each candidate: how popular it is, how recently it was released, how well it matches the user's favorite genres, its price, and the retrieval score itself (Chapters 21 and 22). This structure is common in production because it splits the work into a fast, broad step and a slow, precise step.

The Typical Pipeline

Whatever the model, a production recommender is a pipeline with recognizable stages. Here is the shape we will build for Pagewell Books:



1. **Data.** Events (views, wishlist adds, purchases), item metadata (titles, genres, authors, prices, release dates), and user attributes (country, age band, signup date). Most of the effort in real projects goes here: cleaning, de-duplicating, joining, and deciding what each event means.

2. **Candidate generation.** From the full catalog, pick a few hundred books that might interest this user. Popularity, item similarity, and embedding search are typical sources. The goal is *recall*: do not miss the good ones.
3. **Ranking.** Score each candidate more carefully and sort. The goal is *precision at the top*: put the best ones first.
4. **Rules and re-ranking.** Remove owned or unavailable books, apply content policies, add variety so the list is not ten nearly identical titles, and enforce placement-specific needs (Chapter 24).
5. **Serving.** Deliver lists fast enough for a web page, often by precomputing them overnight and storing them, with a fallback for anyone the model does not know (Chapters 25 and 26).
6. **Feedback.** Log what people saw and what they did. That log becomes tomorrow's training data, which closes the loop and creates both opportunities and hazards (Chapter 27).

Small systems collapse stages together, but the loop always exists: what you recommend today shapes what you learn tomorrow.

How Recommenders Fail

Recommenders rarely crash. They fail quietly, by showing lists that look plausible while serving customers and the business poorly. Learning to recognize these failure modes early saves months of misplaced effort.

The popularity echo chamber

A model trained on purchases learns that popular books are bought often, so it recommends them, so they are bought more often, so the next model is even more confident. Over time the same few dozen titles fill every slot, the long tail of the catalog never gets a chance, and regular customers see nothing new. Chapter 7 measures this with catalog coverage and novelty, and Chapter 27 simulates the loop.

Stale lists

Lists computed once and never refreshed keep recommending a book the customer bought yesterday, or a seasonal title in the wrong season, or nothing from this month's releases. Freshness is a property of the whole pipeline, not of the model alone: how often you retrain, how often you recompute lists, and whether you filter by the latest purchases at serving time.

Cold start

A new customer has no history; a new book has no buyers. Collaborative models are blind in both cases. Without a plan, new customers get an empty or random row, and new books never get recommended, so they never get bought, so they never acquire the history the model needs. Chapter 23 treats both cases.

Leakage-inflated offline wins

The most dangerous failure happens before anything ships. If your evaluation accidentally lets the model see the future, such as purchases from the test period counted in a popularity feature, or a random split that mixes next month's purchases into training, offline numbers look wonderful and online results disappoint. Chapter 4 is devoted to splitting data honestly, and we will be strict about it for the rest of the book.

Optimizing the wrong target

A model can be excellent at the thing you trained it for and poor at the thing you care about. Predicting star ratings is not the same as ranking books a person will buy; maximizing clicks is not the same as building long-term trust. Part II is about choosing metrics that reflect the placement's real purpose and reporting several of them side by side.

> **WARNING**
> A recommender that "beats the baseline by 30%" in an offline experiment has proven nothing until you know how the data was split, which users were evaluated, which items were excluded, and whether the baseline was tuned with the same care. Treat every surprising offline win as a bug until you have ruled out leakage.

Meet Pagewell Books

To make every idea concrete we use one running example. **Pagewell Books** is a fictional online bookstore that sells English-language books to customers in the US, the UK, Canada, Australia, Ireland, and New Zealand. Prices are in US dollars. Its catalog has 1,200 books by 300 authors across 12 genres, and it has 2,000 customers whose views, wishlist adds, purchases, and ratings during 2025 we will study in detail. The data is synthetic, generated by a short Python program with a fixed random seed, so every reader gets exactly the same tables and the same numbers. Chapter 2 shows you how to generate it, and Chapter 3 explains how the generator works and why we chose a synthetic dataset over a public one.

The four placements

Pagewell wants recommendations in four places. Each one will guide our choices later.

Placement	Anchor	List size	What good looks like
"Recommended for you" home row	The signed-in customer	10 books	Relevant to this reader, fresh, not already owned
"Readers who bought this also bought"	The book on the page	5 books	Clearly related to the anchor book

New-customer suggestions	Little or nothing	10 books	Safe, appealing, a good first impression
Weekly email	The subscriber	10 books	Relevant and varied across genres

The people

Three people at Pagewell will appear from time to time, mostly when a decision needs an owner.

- **Ines Marlow** is head of data and leads the recommender project. She cares about honest evaluation and about being able to explain why a model was chosen.
- **Lena Achterberg** is the merchandising and product manager. She owns the business rules (never recommend a book the customer already bought, keep the email varied) and the A/B tests that decide what ships.
- **Bastian Voss** is the backend engineer. He owns the recommendation API and the nightly job that refreshes the lists, and he cares about latency, fallbacks, and not being paged at night.

In a small company one person might wear all three hats, but the three viewpoints are always present.

A few familiar faces in the data

We will keep returning to the same examples so you can watch each model's behavior on cases you already know. **User 106** is a customer in the UK, aged 18 to 24, who signed up on 2025-06-08 and reads mostly mysteries and literary fiction. **User 7** is a Canadian customer who signed up in November 2025, so the models trained on data before November know nothing about them; they are our cold-start example. Book **1124, "The Quiet Orchard"** by Ysolde Jessamy, is the most-bought mystery in our training data and serves as the anchor book whenever we need an "also bought" example.

> **TIP**
>
> Pick two or three "familiar faces" in your own projects as well: a heavy user, a new user, and a well-known item. Printing their lists after every model change catches problems that aggregate metrics hide, such as a list full of books the user already owns.

What "Good" Means at Pagewell

Before writing any code, it is worth writing down what success looks like, because it shapes the metrics we pick in Part II. For the home row, Ines proposes a simple offline target: of the books a customer buys in the weeks after we compute the list, how many did we put in the top 10, and how high? That idea leads directly to precision, recall, hit rate, and NDCG at 10. For the "also bought" row, relevance to the anchor book matters most, so we will inspect lists by eye as well as measure them. For the email,

Lena insists that ten books from one genre is a failure even if each book is individually relevant, which leads to diversity metrics. For new customers, Bastian's requirement is blunt: the row must never be empty, so every path through the system needs a fallback.

There is also a business view that no offline metric captures fully: revenue, customer return rate, and long-term satisfaction. Offline metrics are a proxy for those outcomes, useful for comparing models quickly and cheaply, but the final verdict comes from controlled experiments with real customers. Chapter 9 explains how Pagewell would run those experiments, and why an offline winner can lose online.

Finally, there are constraints. Recommendations are built from personal behavior, so privacy rules apply. In many regions, for example under the GDPR in the European Union, there are rules about how personal data may be collected, stored, and used for profiling. This book is not legal advice: in a real project, consult your organization's privacy policy and legal team about what data you may use and how long you may keep it. Our synthetic data contains no real people, which is one of the reasons we chose it.

The Plan of the Book

The book follows the pipeline from data to production, and it builds one small Python package, `pagewell_rec`, a module at a time. Every chapter's code runs on an ordinary laptop CPU in seconds to a couple of minutes.

Part	Chapters	What you build
I — Foundations	1–5	The data, honest splits, and strong baselines
II — Evaluation	6–9	Ranking metrics, beyond-accuracy metrics, a harness, online tests
III — Content and Neighbors	10–14	TF-IDF, embeddings, user and item kNN, hybrids
IV — Matrix Factorization	15–19	FunkSVD, PyTorch MF, ALS, BPR, tuning
V — Deep and Sequential Models	20–23	Next-item models, two-tower retrieval, a learned ranker, cold start
VI — Production	24–28	Re-ranking, serving, a FastAPI service, monitoring, a case study

The order matters. We build evaluation before fancy models on purpose: without a trustworthy way to compare models, every later chapter would be guesswork. And we keep every result on the same split, with the same rules about which users are evaluated and which books are excluded, so that numbers in Chapter 18 can be compared fairly with numbers in Chapter 5.

One more promise about honesty. The numbers you will see come from real runs on the Pagewell data. Some of the lessons they teach are humbling: a simple popularity list is hard to beat, a classic item-to-item neighborhood model is a very strong baseline on our data, and the model with the best rating

predictions is not the one with the best lists. We report those results as they are. Because the data is synthetic, the exact numbers will not transfer to your store, but the reasons behind them usually do.

Setting Up Your Environment

All code in the book targets Python 3.12 and PyTorch 2.x; the examples were tested with PyTorch 2.14 on CPU and with recent versions of NumPy, pandas, SciPy, scikit-learn, sentence-transformers, and FastAPI. You do not need a GPU. If you have one, later chapters show where a `.to(device)` call would move work onto it, but GPU runs were not part of this book's tests.

Create the project folder and a virtual environment

A virtual environment keeps this project's packages separate from everything else on your machine. Create a folder called `pagewell-rec` and an environment inside it. On Windows, the `py` launcher picks the Python version; on macOS and Linux, call `python3.12` directly.

Windows (PowerShell)

```
mkdir pagewell-rec
cd pagewell-rec
py -3.12 -m venv .venv
.venv\Scripts\Activate.ps1
python -m pip install --upgrade pip
```

macOS / Linux

```
mkdir pagewell-rec
cd pagewell-rec
python3.12 -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
```

Once the environment is active, `python` refers to the environment's interpreter on every platform, and commands in the book that start with `py -3.12` can be typed as `python` instead.

Install the libraries

Install the packages the book uses in one go. PyTorch publishes separate CPU and GPU builds; the plain `pip install torch` command picks a default build for your platform, and the PyTorch website has a selector if you want a specific CPU-only or CUDA build.

```
python -m pip install numpy pandas scipy scikit-learn threadpoolctl matplotlib
python -m pip install torch
python -m pip install sentence-transformers
python -m pip install fastapi uvicorn httpx pytest
```

Downloads can be large, PyTorch in particular, so expect the second line to take a few minutes. You only need sentence-transformers from Chapter 11 on and FastAPI from Chapter 26 on, so you can postpone those lines if you prefer.

Check the installation

The quickest check is a one-liner that imports PyTorch and prints its version:

```
py -3.12 -c "import torch; print(torch.__version__)"
```

Output

```
2.14.1+cpu
```

The `+cpu` suffix tells you this is a CPU-only build; a CUDA build shows a different suffix. Your version number will depend on when you install. For a fuller check, save this small script as `check_env.py` in the project folder and run it:

check_env.py

```
import sys

import numpy, pandas, scipy, sklearn, torch

sys.stdout.reconfigure(encoding="utf-8")
print("python      ", sys.version.split()[0])
for mod in (numpy, pandas, scipy, sklearn, torch):
    print(f"{mod.__name__:12s}", mod.__version__)
print("cuda available:", torch.cuda.is_available())
```

Output

```
python       3.12.0
numpy        2.5.3
pandas       3.0.6
scipy        1.18.1
sklearn      1.9.1
torch        2.14.1+cpu
cuda available: False
```

You should see something similar, with whatever versions your installation picked. If an import fails, the environment is probably not active: run the activation command again and check that `python -c "import sys; print(sys.prefix)"` points inside `.venv`.

> **NOTE**
>
> On some systems the console uses a non-UTF-8 code page, which can garble book titles or symbols in printed output. Every script in this book starts with `sys.stdout.reconfigure(encoding="utf-8")`, and you can also set the environment variable `PYTHONIOENCODING=utf-8` before running scripts.

The project layout

By the end of the book your project folder will look like this. You do not need to create these files now; each chapter adds the files it introduces, and no chapter imports a module before it has been written.

```
pagewell-rec/
  generate_data.py  explore_data.py          # Chapters 2-3
  run_baselines.py  run_content.py  run_knn.py  run_mf.py  run_bpr.py
  run_sequence.py  run_two_tower.py  run_ranking.py  run_cold_start.py
  run_rerank.py  build_artifacts.py  app.py  test_app.py  run_all.sh
  data/              users.csv  books.csv  events.csv  ratings.csv
  artifacts/         precomputed lists and embeddings (Chapters 11, 25)
  pagewell_rec/
    __init__.py  config.py  data.py          # Chapter 3
    split.py                                 # Chapter 4
    base.py  baselines.py                    # Chapter 5
    metrics.py                               # Chapters 6-8
    content.py                               # Chapters 10-11
    knn.py  hybrid.py                        # Chapters 12-14
    mf_numpy.py  mf_torch.py  bpr.py         # Chapters 15-18
    sequence.py  two_tower.py  ranking.py    # Chapters 20-22
    rerank.py  serving.py                    # Chapters 24-25
```

The package name uses an underscore, `pagewell_rec`, because Python module names cannot contain hyphens; the folder uses a hyphen, `pagewell-rec`, by convention. Scripts at the top level import from the package, so always run them from inside the project folder.

Keeping the CPU calm

PyTorch and scikit-learn use every CPU core by default, which can slow things down on a shared machine. From Chapter 3 on, `config.py` has a helper, `limit_threads(2)`, that caps the thread pools at two; the timings in the book were measured that way.

A First Taste: Counting Popular Books

We will not build any models in this chapter, but it is worth seeing how little code the simplest recommender needs. Once you have generated the Pagewell data in Chapter 2, the following snippet counts how many distinct customers bought each book before November 2025 and prints the five best sellers. Those five books are, in effect, a non-personalized "Recommended for you" row that every customer would see.