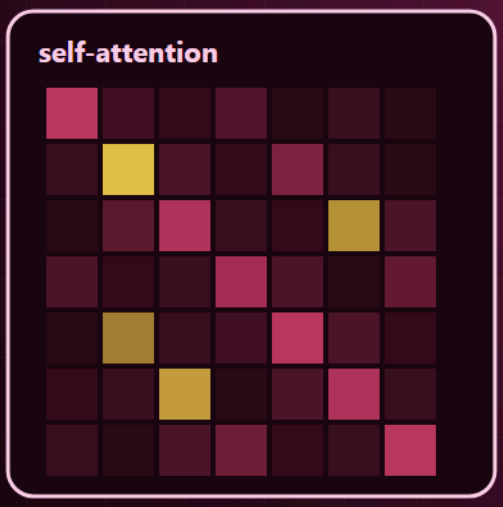
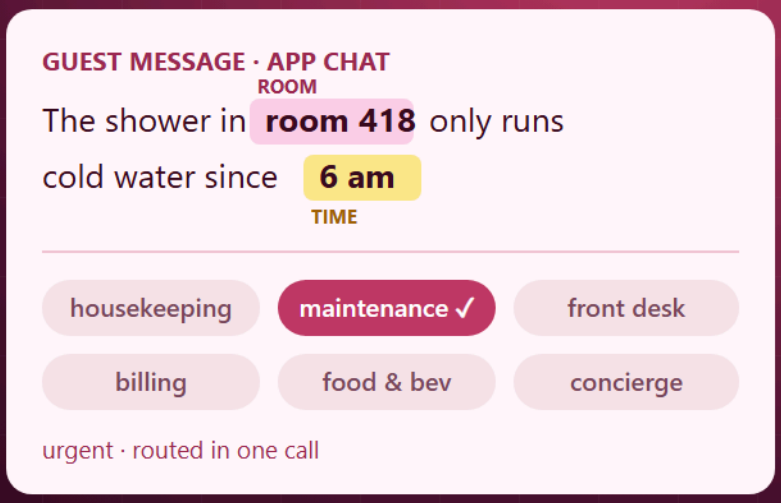







```python
from transformers import pipeline
router = pipeline("text-classification",
                  model="artifacts/copperline-router")
router("No hot water in room 418") # maintenance
```

A HANDS-ON, CPU-FRIENDLY GUIDE

Practical NLP with Transformers

Classify, extract, search, and serve text with Hugging Face

Tokenizers · Pipelines · Fine-tuning with Trainer · NER
Question answering · Embeddings · Semantic search · ONNX · FastAPI

TOKENIZE · FINE-TUNE · EXTRACT · SEARCH · SERVE

THENEXBOOKS

Practical NLP with Transformers

Classify, Extract, Search, and Serve Text with Hugging Face

From your first tokenizer to a tested, quantized guest-message router — built step by step on a CPU

First Edition

THENEXBOOKS

Practical NLP with Transformers
Classify, Extract, Search, and Serve Text with Hugging Face

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. Open-source libraries such as Transformers, Datasets, Tokenizers, sentence-transformers, PyTorch, and ONNX Runtime change quickly; function names, defaults, and printed output can differ between versions. Model outputs and timings shown in this book come from small CPU runs on one test machine; your numbers will differ. Every pretrained model has its own license — check it before any production use.

Hugging Face is a trademark of Hugging Face, Inc. PyTorch is a trademark of The Linux Foundation. Python is a trademark of the Python Software Foundation. ONNX is a trademark of the LF Projects, LLC. Windows is a trademark of the Microsoft group of companies. Other product names are 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.

Copperline Hotels, the hotel group used as the running example in this book, is fictional, as are its hotels, staff, guests, messages, policies, and all sample data. The guest messages are generated by a seeded script printed in the book.

Preface

Every organization that talks to customers is sitting on a pile of text: reviews, emails, chat messages, support tickets, survey comments. Most of it is read once, by whoever happens to see it first, and then forgotten. A broken shower waits in the wrong inbox for half a day. A billing complaint is answered by the restaurant. The same question about parking is typed out by hand two hundred times a month.

Transformer models changed what a small team can do about that. A few years ago, sorting messages by meaning, pulling out room numbers and dates, or finding the policy paragraph that answers a question required a research group. Today a developer who knows basic Python can download a pretrained model, fine-tune it on a few hundred examples on an ordinary laptop, and put it behind an API the same week. The hard part is no longer getting a model to run — it is knowing which model to use, how to tell whether it actually works, and how to ship it responsibly.

This book teaches that practical skill set. We do not just tour the Hugging Face libraries. We build one real system step by step, introduce each idea at the moment it solves a problem we actually have, and measure everything we build against honest baselines, so you learn to make good decisions on your own text problems.

Who This Book Is For

- **Python developers** who want to add text classification, extraction, or search to their applications and need a clear path from "hello world" pipelines to a tested service.
- **Data analysts and scientists** who know classic machine learning and want to understand what transformers add, when they are worth it, and when a simpler model wins.
- **Backend and platform engineers** who will deploy, monitor, and optimize NLP models and need to understand what happens inside them.
- **Students and self-learners** who want a hands-on, honest introduction to modern NLP that runs on a normal computer without a GPU.

You should be comfortable with basic Python: functions, lists and dictionaries, installing packages, and running scripts. We explain the machine learning and PyTorch you need as we go.

What You Will Learn

- How text becomes numbers: tokenizers, vocabularies, special tokens, padding, truncation, and attention masks.
- What is inside a transformer: embeddings, self-attention, encoder and decoder blocks, and how to choose an architecture.
- How to work with the Hugging Face Hub and the `datasets` library: model cards, licenses, caching, offline use, mapping, and splitting.

- How to fine-tune a small encoder for classification with `Trainer` and with a plain PyTorch loop — and how to evaluate it honestly against baselines.
- How to handle small data, class imbalance, messy labels, and reproducibility.
- How to extract entities, answer questions from documents, summarize, translate, classify without labels, and generate text.
- How to use sentence embeddings for semantic search, re-ranking, duplicate detection, multilingual queries, and clustering.
- How to make models fast and small with ONNX and quantization, serve them with FastAPI, and monitor them responsibly in production.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) introduces transformers for language, sets up the environment and the Copperline dataset, explains tokenization in depth, and shows ready-made pipelines. **Part II — Models and Data** (Chapters 6–10) opens up the model, compares architectures, covers the Hub and the `datasets` library, and builds the baselines every model must beat. **Part III — Fine-Tuning Classifiers** (Chapters 11–15) fine-tunes DistilBERT into the Copperline router, rebuilds the training loop by hand, and covers evaluation, small and imbalanced data, and reproducible experiments. **Part IV — Beyond Classification** (Chapters 16–20) covers named entity recognition, question answering, summarization and translation, zero-shot classification, and text generation. **Part V — Embeddings and Search** (Chapters 21–24) covers sentence embeddings, semantic search and re-ranking, multilingual text, and clustering. **Part VI — Production** (Chapters 25–28) covers efficient inference, serving, monitoring and responsible use, 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: Copperline Hotels

Throughout the book we work with **Copperline Hotels**, a fictional group of three hotels — Copperline Bayfront, Copperline Midtown, and Copperline Canyon. Guests write in through reviews, the mobile app chat, and email, and every message has to reach the right team: housekeeping, maintenance, the front desk, billing, food and beverage, or the concierge. We build the **Copperline Guest Router**: a fine-tuned model that routes each message to a department, an extractor that pulls out room numbers, dates, times, and requested items, a semantic search over the hotels' policies, and a fast, quantized API that ties it all together. All 1,200 guest messages come from a seeded generator printed in Chapter 2, so every number in this book can be reproduced on your own machine.

Versions and Conventions

Examples use **Python 3.12** with recent versions of PyTorch, Transformers (version 5 series), Datasets, Tokenizers, sentence-transformers, scikit-learn, ONNX Runtime, and FastAPI. Everything in the book runs on a **CPU**: we use small models such as DistilBERT and MiniLM, small data subsets, and two CPU threads, so each example finishes in a few minutes on an ordinary computer. Where a GPU would change things, a Note says what to expect. Library APIs move quickly; where a function was renamed or removed in recent versions, we say so. Command examples use Windows paths where it matters, and they work the same way on macOS and Linux with the usual path changes.

> **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 cause wrong results, data leaks, or security problems.

Code appears in `monospace`; long lines are wrapped to fit the page. File captions above code blocks show where the code lives in the companion project, for example `copperline_nlp/classify.py`. Printed output is real output from our test runs unless a caption says it is illustrative. Where a chat model or AI assistant is mentioned, we write *[your AI model]* rather than naming a product.

Contents

Preface

PART I — FOUNDATIONS

1 Transformers for Language: What They Do and How This Book Works

2 Setting Up: Python, PyTorch, Hugging Face, and the Copperline Dataset

3 Tokenization: From Text to Token IDs

4 Tokenizer Families: WordPiece, BPE, SentencePiece, and Training Your Own

5 Pipelines: Ready-Made NLP in a Few Lines

PART II — MODELS AND DATA

6 Model Anatomy: Embeddings, Attention, and Transformer Blocks

7 Encoders, Decoders, and Encoder-Decoders

8 The Hugging Face Hub: Model Cards, Licenses, Caching, and Offline Use

9 The datasets Library: Loading, Mapping, Splitting, and Cleaning Text

10 Baselines and Honest Evaluation Splits

PART III — FINE-TUNING CLASSIFIERS

11 Fine-Tuning DistilBERT with Trainer: The Copperline Router

12 Under the Hood: A Training Loop in Plain PyTorch

13 Evaluating Classifiers: Metrics, Confusion Matrices, and Error Analysis

14 Small Data, Imbalance, and Messy Labels

15 Hyperparameters, Reproducibility, and Experiment Tracking

PART IV — BEYOND CLASSIFICATION

16 Named Entity Recognition: Extracting Request Details

17 Extractive Question Answering over Hotel Policies

18 Summarization and Translation with Encoder-Decoder Models

19 Zero-Shot Classification: Routing Without Labels

20 Text Generation and Decoding Strategies

PART V — EMBEDDINGS AND SEARCH

21 Sentence Embeddings with sentence-transformers

22 Semantic Search, Re-Ranking, and Duplicate Detection

23 Multilingual Text: One Model for Many Languages

24 Clustering and Topic Discovery in Guest Feedback

PART VI — PRODUCTION

25 Efficient Inference: ONNX, Quantization, and Batching

26 Serving NLP Models with FastAPI

27 Responsible NLP in Production: Monitoring, Drift, Privacy, and Fairness

28 Case Study: The Complete Copperline Guest Router

APPENDICES

A NLP with Transformers Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

What transformers do for language and when to use them, a reproducible CPU-friendly setup with the seeded Copperline dataset, how tokenizers turn text into numbers, and ready-made pipelines that solve real tasks in a few lines.

1 · Transformers for Language
2 · Setting Up and the Copperline Dataset
3 · Tokenization
4 · Tokenizer Families
5 · Pipelines

CHAPTER 1 · PART I — FOUNDATIONS

Transformers for Language: What They Do and How This Book Works

Every day a hotel group receives hundreds of reviews, chat messages, and emails that someone has to read, sort, and answer. In this chapter we meet that problem, see which parts of it modern transformer models can solve, run our first model in five lines of Python, and map out the road from that first experiment to a complete, tested text-routing service.

A Morning in the Copperline Inbox

Imagine it is 7:45 am at a mid-sized hotel group. Overnight, guests have written in through three channels: public reviews, the hotel app's chat window, and plain email. One guest says the shower in their room only runs cold water. Another wants a table for four on Friday evening. A third is annoyed that a deposit hold from last weekend still shows on their card, and adds, almost as an afterthought, that the bar closed early. A fourth just wants to say the housekeeping team did a lovely job.

Each of these messages belongs to a different team. The cold shower is a job for maintenance, and it is urgent. The table booking goes to food and beverage. The deposit hold belongs to billing, even though the same message also complains about the bar. The thank-you note should reach housekeeping, because praise matters to the people who earned it. Today a person on the front desk forwards every message by hand, and on a busy weekend urgent messages wait in the same queue as compliments.

This is a classic **natural language processing** (NLP) problem: software has to read free-form human text and do something useful with it. The text is short, informal, and full of typos, lowercase phone typing, room numbers, dates, and times. No two guests describe a broken heater the same way. Hand-written rules and word-count models still have their place, and we use them as baselines. But the tool that changed NLP is the **transformer**, a neural network architecture that learns rich representations of text from huge amounts of unlabeled writing and can then be adapted to a specific job with a surprisingly small amount of labeled data.

This book teaches you to use transformers for practical language tasks with the Hugging Face libraries, by building one realistic system from start to finish. You need basic Python (functions, dictionaries, pip, running scripts); machine learning ideas are explained as they appear, and PyTorch only as far as you need it.

Meet Copperline Hotels

Our running example is **Copperline Hotels**, a fictional group of three hotels. Every message in our dataset carries a three-letter hotel code, so it is worth learning them now.

Code	Hotel	Facts that appear in the data
`BAY`	Copperline Bayfront	9 floors; runs the airport shuttle
`MID`	Copperline Midtown	12 floors; parking costs $28 per night
`CAN`	Copperline Canyon	4 floors; free outdoor parking; pets up to 25 kg allowed

Three people at Copperline will appear again and again. **Imogen Treadwell** is the Director of Guest Experience. She owns the routing project: she decides what counts as a good enough result, which messages must be answered within minutes, and when a human should take over from the software. **Yusra Farouk** is the machine learning engineer who builds the system; throughout the book you work alongside her, writing the same code she writes. **Anselm Jansen** runs IT and the platform team, which means he cares about where the models run, how much memory they use, what gets logged, and whether guest data stays private.

The six departments

Imogen's team agreed on six routing targets. The order matters, because later the model will predict a number from 0 to 5, and that number must always mean the same department. We fix the ids once, here, and never change them.

Label id	Department (in code)	Human-readable name	Typical message
0	`housekeeping`	housekeeping	"Could we get extra towels sent up?"
1	`maintenance`	maintenance	"The air conditioning is not working."
2	`front_desk`	front desk	"Can we check in early on Friday?"
3	`billing`	billing	"I was charged twice for the night."
4	`food_beverage`	food and beverage	"Can we book a table for four?"
5	`concierge`	concierge	"Can you book a taxi to the airport?"

Messages arrive through three channels, which appear in the data as `review`, `app_chat`, and `email`. Each message also carries a sentiment label (`negative`, `neutral`, or `positive`) and an urgency flag.

Three business rules

Software that routes guest messages needs rules that a hotel manager can explain to staff. Imogen wrote down three, and our system must follow them exactly.

1. **The routing rule.** A message goes to the department of the **first issue it mentions**. Reviews often mention two things, such as a billing problem followed by a complaint about the bar. The label is still the first issue. The rule keeps labeling consistent.
2. **The confidence threshold.** The router outputs a score for its top department. If that score is below **0.60**, the message is flagged `needs_review = True` and goes to the front-desk supervisor queue, where a person decides. In code this is the constant `CONFIDENCE_THRESHOLD = 0.60`. The threshold is a business decision, not a technical one: Imogen chose it, and in Chapter 13 we measure what it costs and what it buys.
3. **Response times.** Urgent messages (no air conditioning, a cold shower, leaks, a window that will not close, a broken key card or key card reader, a missed airport shuttle) must be acknowledged within **15 minutes**. Everything else must be acknowledged within **2 hours**.

There is also a privacy rule that Anselm insisted on from day one: every prediction is logged with the message id, the model version, the predicted label, and the score, but never the raw guest text in any log that leaves the hotel. We come back to that in Chapter 27.

> **NOTE**
>
> Copperline Hotels, its staff, and its guests are fictional. The 1,200 guest messages we use are generated by a short, seeded Python script that you will read line by line in Chapter 2. Synthetic data has a big advantage for learning: everyone who runs the code gets exactly the same rows and therefore the same numbers. It also has a weakness that we take seriously later. Text built from templates is more regular than real guest writing, so models look better on it than they would on a real inbox. We measure that gap with a separate set of 30 hand-written messages.

What NLP Tasks Look Like

"NLP" covers many different jobs. It helps to sort them by the shape of their output, because the shape decides which kind of model head, which metric, and which chapter you need. Here is how the main task types look when we apply them to Copperline messages.

Classification: one label per message

Text classification assigns a label from a fixed list to a whole piece of text. Routing is classification with six classes: "The shower in room 418 only runs cold water" maps to `maintenance`. Sentiment analysis is classification too, with three classes in our data. Urgency detection is binary classification. Classification is the workhorse of practical NLP, and the core of this book (Part III) is about doing it well: fine-tuning a model, evaluating it honestly, and handling small, imbalanced, messy data.

Extraction: labels on pieces of the text

Sometimes the useful information is a span inside the message. A maintenance request is much more useful if the system also pulls out `room 418`, and a room-service request is more useful with `a club sandwich` and `8:30 pm` attached. This is **named entity recognition** (NER), also called token

classification, because the model assigns a label to each token rather than to the whole text. Our entity types are `HOTEL`, `ROOM`, `DATE`, `TIME`, and `ITEM`, and we train an extractor for them in Chapter 16.

Question answering: find the answer in a document

Guests ask the same questions every week. When is checkout? How much is parking? Why is there still a hold on my card? **Extractive question answering** takes a question and a passage, such as Copperline's 12-entry FAQ, and returns the exact span of the passage that answers it, for example `'11 am'`. The model does not invent words; it points at them. We build it in Chapter 17.

Search: find the most similar text

A guest who writes "Is there somewhere to park my car downtown?" never uses the word "parking", yet the FAQ entry about parking is clearly the right answer. **Semantic search** turns each text into a vector (an embedding) so that texts with similar meaning end up close together, even when they share few words. Part V covers embeddings, search, re-ranking, multilingual text, and clustering.

Generation: produce new text

Summarization, translation, and free text generation produce new words rather than labels or spans. They are useful, and also risky: a generated summary can silently drop a complaint, and a generated reply can promise something the hotel cannot deliver. We look at them carefully in Chapters 18 and 20, and we use them where a human stays in the loop.

Zero-shot: no labeled data at all

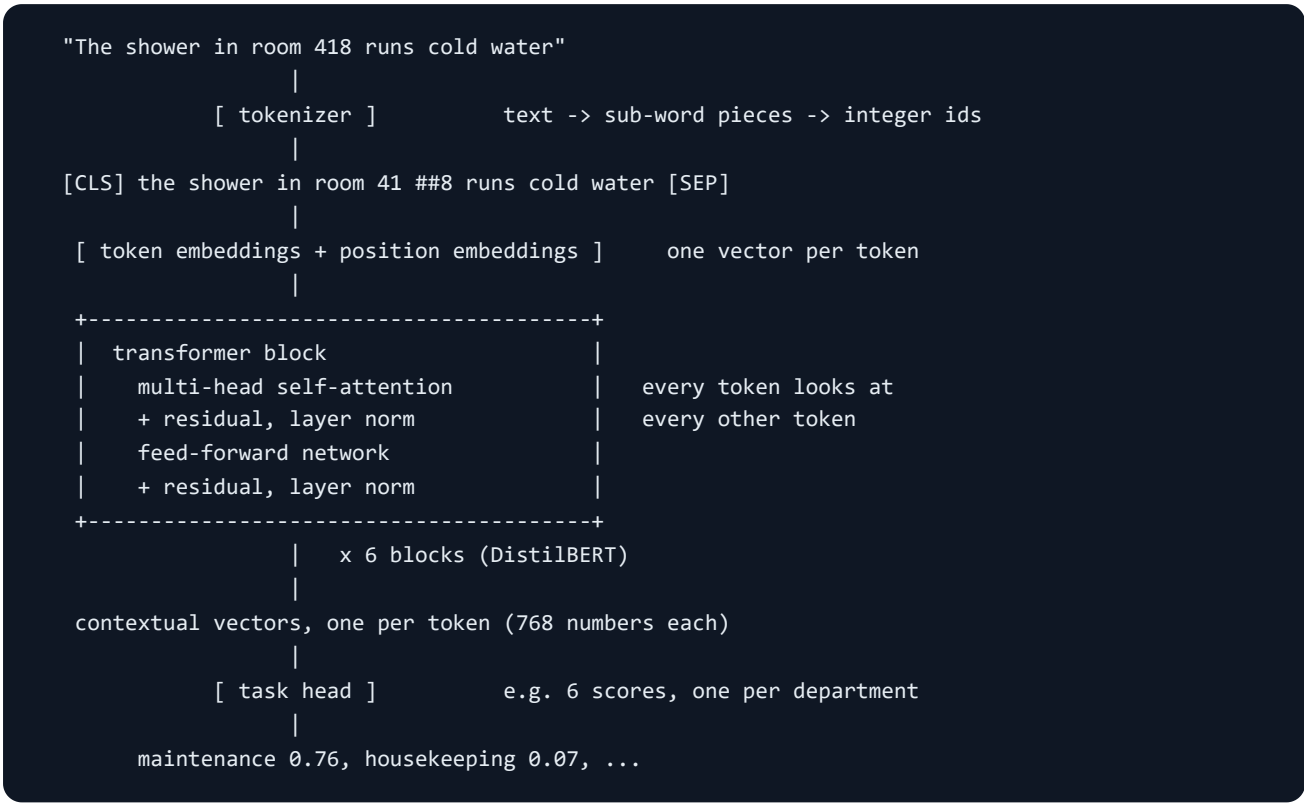
Finally, some models can classify text into labels they were never trained on, by checking how well a sentence like "This message is about billing." fits the message. This **zero-shot** approach is a fast way to get a first router before any labels exist. It is less accurate than a fine-tuned model on our data, and Chapter 19 measures exactly how much less.

Task type	Output shape	Copperline example	Chapter
Classification	One label per text	Route to a department	11–15
Token classification	One label per token	Extract room, date, time, item	16
Extractive QA	A span of a passage	Answer from the FAQ	17
Seq2seq generation	New text	Summarize, translate	18
Zero-shot	Label from a free list	Route without training	19
Text generation	New text, left to right	Draft a reply (carefully)	20
Embeddings and search	A vector per text	FAQ search, duplicates	21–24

What a Transformer Is, in One Picture

We open up the model properly in Chapter 6. For now, one picture and a few sentences are enough to understand everything we do in the first five chapters.

A transformer encoder, from text to a routing decision



Reading from top to bottom: a **tokenizer** splits the text into pieces that come from a fixed vocabulary and maps each piece to an integer id. Notice that the room number `418` becomes two pieces, `41` and `##8`; Chapter 3 explains why and why it matters. Each id is looked up in an **embedding** table and turned into a vector, and a second vector that encodes the token's position is added to it, because the rest of the model has no built-in sense of word order.

Then come the **transformer blocks**. The key ingredient is **self-attention**: for every token, the model computes how much it should "look at" every other token in the message and mixes their information accordingly. That is how the vector for "cold" ends up carrying information about "shower" and "water", and how the vector for "room" learns that it is followed by a number. After attention, a small feed-forward network processes each position separately. Residual connections and layer normalization keep the training stable. A model stacks several of these blocks; the DistilBERT model we fine-tune has six.

The output is a list of **contextual vectors**, one per token. The same word gets a different vector in a different sentence, which is exactly what simpler word-count methods cannot do. On top sits a small **task head**, often a single linear layer, that turns those vectors into what we need: six department scores, eleven entity tags per token, or a start and end position for an answer.

The architecture was introduced in the 2017 paper "Attention Is All You Need". BERT (2018) showed how well an encoder-only transformer works for understanding tasks after pretraining, GPT-2 (2019) did the same for text generation with a decoder-only model, T5 (2019) framed many tasks as text-to-text with an encoder-decoder, and DistilBERT (2019) compressed BERT into a smaller, faster model. Those four families are the ones we use in this book.

Pretrained, Then Fine-Tuned

If transformers had to learn English from our 1,200 hotel messages, they would be useless. The idea that makes them practical is **transfer learning** in two stages.

In the first stage, **pretraining**, a research lab trains the model on a very large collection of ordinary text with a task that needs no human labels. BERT-style encoders learn by **masked language modeling**: hide some words and predict them from the surrounding context. GPT-style decoders learn by predicting the next token from the tokens before it. Doing this over and over teaches the model a great deal about grammar, word meaning, and how ideas relate. Pretraining is expensive, and we never do it in this book. We download the result, a checkpoint, from the Hugging Face Hub.

In the second stage, **fine-tuning**, we take that pretrained model, attach a fresh task head, and keep training on a small labeled dataset for our specific job. Because the model already understands a lot of language, it needs far fewer labeled examples than a model trained from scratch. In Chapter 11 we fine-tune `distilbert/distilbert-base-uncased`, a 6-layer distilled version of the 12-layer BERT-base model, on just 192 labeled messages for three short epochs on a laptop CPU. On our held-out test set of 180 messages, the reference run (seed 42) reaches an accuracy of 0.9056 and a macro-F1 of 0.8839. That is a useful router trained in a couple of minutes, although Chapter 15 shows that such a short run varies a lot from seed to seed.

Encoders, Decoders, and Encoder-Decoders in Two Sentences

An **encoder** (BERT, DistilBERT, MiniLM) reads the whole input at once, so every token can see the tokens on both sides of it, which makes encoders the natural choice for understanding tasks: classification, extraction, and embeddings. A **decoder** (GPT-2, distilgpt2, and the chat models you know) uses a causal mask so each token sees only what came before it, which makes it good at generating text left to right, while an **encoder-decoder** (T5, BART) encodes an input and then decodes a new output while attending to it, which suits summarization and translation.

Most of this book uses small encoders, because routing, extraction, and search are understanding tasks and encoders are cheap to run. Chapter 7 compares the three families side by side, and it explains where large chat models fit. We refer to those as "[your AI model]" throughout, because the specific product does not matter for the ideas, and because this book is about models you can run, inspect, and fine-tune yourself.

When Not to Use a Transformer

A practical book has to say this early: a transformer is not always the right tool. Before you fine-tune anything, ask whether a simpler method already solves the problem.

- **Rules and lookups.** If every message from the app's "Request towels" button arrives with a fixed prefix, a single `if` statement routes it perfectly, costs nothing to run, and never surprises anyone.
- **Regular expressions.** Room numbers, booking references, email addresses, and card numbers follow patterns. A regex is often more reliable than a model for these, and you can read it. We use regexes for redacting personal data in Chapter 27.
- **Classic machine learning.** A TF-IDF representation (word and word-pair counts, weighted by how rare they are) fed into logistic regression trains in well under a second and is a very strong baseline for short texts.

That last point is not hypothetical. In Chapter 10 we train a TF-IDF plus logistic regression baseline on the same 192 messages as the fine-tuned router. On the templated test set it scores an accuracy of 0.9611, **higher** than the router's 0.9056. If we stopped there, we would ship the simple model. But on the 30 hand-written challenge messages that use wording the templates never use, the order flips: the router scores 0.6333 and TF-IDF 0.5333. The keyword model memorized template vocabulary; the transformer most likely benefited from the general language knowledge it brought from pretraining. Neither number alone tells the full story, and the real lesson is about how you test, not which model wins.

> **TIP**
>
> Write down your baseline before you write any model code. A majority-class guess (always predict the most common department) scores an accuracy of 0.2000 on our test set. Any model that cannot beat a keyword baseline on realistic data is not ready, no matter how impressive its architecture sounds.

The CPU-Only Promise

Many transformer tutorials assume you have a powerful GPU. This book does not. Every example was run on an ordinary CPU, with PyTorch limited to two threads so that results are repeatable and your machine stays responsive. That is possible because we deliberately choose small models and small training runs.

- Our main encoder, DistilBERT, has about 66 million parameters. Loading it takes a few seconds and classifying a batch of messages takes a fraction of a second per message.
- Fine-tuning the router uses 192 training rows for three epochs, which is 36 optimizer steps. On our test machine training took about 96 seconds when the machine was otherwise quiet. The NER extractor took about two minutes.

- Embedding models such as `sentence-transformers/all-MiniLM-L6-v2` have about 23 million parameters and run comfortably on a CPU.

Timings come from one shared test machine and vary from run to run; treat them as rough guides, not benchmarks. Where a production run would differ (all 840 training rows, more epochs, a GPU), we say so. GPU settings are labeled untested here.

Your First Transformer in Five Lines

Enough talk. Chapter 2 walks through installing everything; if you already have Python 3.12 and recent versions of `torch` and `transformers`, you can run this now. It uses a DistilBERT model that someone else already fine-tuned for sentiment on movie-review sentences (the SST-2 dataset), and asks it about three Copperline messages.

first_pipeline.py

```
import copperline_nlp  # switches Hugging Face to offline mode
from transformers import pipeline

clf = pipeline("sentiment-analysis",
               model="distilbert/distilbert-base-uncased-finetuned-sst-2-english")
print(clf("The shower in room 418 only runs cold water. Please fix this."))
print(clf("Can we book a table for four at 7:30 pm on Friday?"))
print(clf(["Check-in was quick and the receptionist was lovely."]))
```

Output

```
Loading weights: 100%|██████████| 104/104 [00:00<00:00, 621.36it/s]
[{'label': 'NEGATIVE', 'score': 0.9977599382400513}]
[{'label': 'NEGATIVE', 'score': 0.9921042919158936}]
[{'label': 'POSITIVE', 'score': 0.9998513460159302}]
```

The first line imports our project package, `copperline_nlp`, before anything from Hugging Face. Importing it sets two environment variables that keep the libraries offline, so they use only models already in your local cache and never try to download. If you are running this before setting up the project, you can drop that line; the pipeline will then download the model on first use (about 270 MB) and cache it. The progress bar on the first output line comes from loading the weights; you will see it, or something like it, often.

The `pipeline()` function hides three steps: it tokenizes the text, runs the model, and turns the raw scores into a readable label and probability. It accepts a single string or a list of strings and always returns a list of dictionaries. Chapter 5 is all about pipelines.

Now look at the results. The cold shower is `NEGATIVE` with a score of 0.9978, which is right. The check-in compliment is `POSITIVE` at 0.9999, also right. But the table booking, a perfectly polite and neutral request, comes back `NEGATIVE` with a score of 0.9921. The model is not broken. It was trained with

only two labels, positive and negative, so it has no way to say "neutral", and it is forced to pick one. Worse, it picks one with high confidence.

This tiny example contains two lessons that run through the whole book. First, a model's labels are a contract: an off-the-shelf model answers the question it was trained on, not the question you have. Copperline needs three sentiment classes and six departments, so we will fine-tune our own models with our own labels. Second, a high score is not the same as a correct answer. The score of 0.9921 measures how strongly the model prefers one of its two options, not how likely it is to be right. We return to this gap between confidence and correctness when we choose thresholds in Chapter 13.

> **WARNING**
> Never use an off-the-shelf model's confidence score as a safety signal without checking it on your own data. The SST-2 sentiment model is 99% "sure" that a neutral table booking is negative. If Copperline escalated every message with a negative score above 0.95 to a manager, every table booking would be escalated.

A Tour of the Book

The book has six parts. Each builds on the one before, and each chapter adds one piece to the Copperline Guest Router or teaches a skill you need to build it well.

Part	Chapters	What you build or learn
I — Foundations	1–5	Setup, the dataset, tokenizers, ready-made pipelines
II — Models and Data	6–10	Model internals, the Hub, `datasets`, baselines, honest splits
III — Fine-Tuning Classifiers	11–15	The router: Trainer, a plain PyTorch loop, evaluation, small data, tracking
IV — Beyond Classification	16–20	Entity extraction, question answering, summarization, zero-shot, generation
V — Embeddings and Search	21–24	Sentence embeddings, FAQ search, re-ranking, multilingual text, clustering
VI — Production	25–28	ONNX and quantization, a FastAPI service, monitoring and privacy, the case study

Part III is the heart of the book: we fine-tune DistilBERT into the router twice, first with the high-level `Trainer` and then with a plain PyTorch loop so nothing stays magic, and then evaluate it the way a careful engineer would. Chapter 28 assembles the complete system and walks three messages through it end to end.

What the finished router does

To make the goal concrete, here is what the final service in Chapter 26 returns for one guest message. You send it text; it returns a department, a confidence score, whether a person must review it, and the entities it found.

POST /route, response for "The shower in room 418 only runs cold water."

```
{"department": "maintenance", "confidence": 0.8576, "needs_review": false,
 "entities": [{"type": "ROOM", "text": "room 418", "start": 14, "end": 22}]}
```

The same service sends "Can you book a taxi to the airport at 6 am on Friday?" to the review queue, because its top score, 0.3547 for concierge, is below the 0.60 threshold. That is not a failure. It is the system doing exactly what Imogen asked: when the model is unsure, a human decides.

The `copperline_nlp` Package

All the reusable code in this book lives in one small Python package called `copperline_nlp`. Each chapter that creates a module shows its code, and short runnable scripts with numbered names exercise the package and print the results we quote. Here is the layout you will have by the end of the book.

Project layout

```
copperline/
  copperline_nlp/
    __init__.py       sets offline mode for Hugging Face on import
    config.py         seed, CPU threads, paths, model ids, setup()      (Ch 2)
    data.py           dataset generator, splits, FAQ, challenge set     (Ch 2, 9)
    baselines.py      TF-IDF, majority class, zero-shot baselines       (Ch 10, 19)
    classify.py       the router: train, load, predict                  (Ch 11)
    manual_loop.py    the same training in plain PyTorch                (Ch 12)
    tracking.py       experiment tracking helpers                       (Ch 15)
    ner.py            the entity extractor                              (Ch 16)
    search.py         FAQ search and re-ranking                         (Ch 22)
    onnx_export.py    ONNX export, int8 quantization, timing            (Ch 25)
    api.py            FastAPI service: GET /health, POST /route         (Ch 26)
    monitoring.py     drift and confidence monitoring                   (Ch 27)
    redact.py         removing personal data before logging             (Ch 27)
  01_make_data.py ... 11_model_facts.py    runnable chapter scripts
  data/               copperline_messages.jsonl, copperline_faq.jsonl
  artifacts/          trained models and ONNX files
```

Every script starts the same way: it imports from `copperline_nlp` first, then calls `setup()`, which fixes the random seed to 42, limits PyTorch to two CPU threads, keeps Hugging Face offline, and makes sure printing non-English characters works on any console. Because of that, you should get the same numbers we print. When a result depends on timing or on your hardware, we say so.

How to read the code and output in this book

- A label above a code block such as `copperline_nlp/config.py` names the file the code belongs to. Blocks labeled **Output** show what the code printed when we ran it.
- Long outputs are shortened. A line containing `...` marks a cut; nothing else is changed.
- Output we could not run on our test machine (anything that needs a GPU, a cloud account, or a tool that is not installed) is labeled **illustrative** or **untested here**, and we tell you what to expect instead of pretending.
- The code targets recent versions of the Hugging Face stack (Transformers v5); where older tutorials use a different name, we point it out.

> **NOTE**
>
> The models we use are open models published on the Hugging Face Hub, and we name them by their repository id, such as `distilbert/distilbert-base-uncased`. Each model has its own license in its model card. The ones in this book use permissive licenses (Apache-2.0 or MIT), but you should always read the license of any model yourself before using it in a product. Chapter 8 shows how.

A Note on Honesty

You will see a sentiment model confidently mislabel a polite request, a pretrained entity tagger call Imogen Treadwell an organization, a semantic search engine rank the right FAQ answer third, a summarizer drop half of a review, and a simple keyword model beat our fine-tuned transformer on the test set. Each of those moments teaches something you need for real systems.

Imogen's standard for the project is a good standard for the book: the router does not have to be perfect, but it must be measured, its limits must be known, and it must hand difficult messages to a person rather than guess.

Try It Yourself

1. Run the five-line sentiment example on five messages of your own: two clear complaints, two clear compliments, and one neutral question. How many neutral questions do you need to try before you find one the model calls `POSITIVE`?
2. Take five messages from your own email or chat (or invent them) and label each with one of the six Copperline departments using the first-issue rule. Which ones were hard to label, and why? Write one sentence of labeling guidance that would have made each hard case easier.
3. For each task type in the table in this chapter, think of one task at your own job or in a product you use. Which of them could be solved with a rule or a regular expression instead of a model?