FREE SAMPLE EDITION

This preview contains the first 1 chapter of the complete book.

BOOK DETAILS

- **Title:** Practical NLP with Python - Text Processing to Modern LLM Workflows
- **Edition:** 1st Edition
- **Publisher:** Alkademy Books
- **Chapters in full book:** 10

CHAPTERS IN THIS PREVIEW

- Chapter 1: The NLP Landscape: From Rules to LLMs

Get the full book at `alkademy.com`

Practical NLP with Python - Text Processing to Modern LLM Workflows

Alkademy Content Team

1st Edition

Alkademy Books

June 2026

PREFACE

A few years ago, I found myself staring at a production system that had quietly stopped working. A text classification pipeline we had built — carefully tuned, reasonably accurate, and happily deployed — had begun to drift. The language in our incoming documents had shifted subtly over time, the vocabulary had evolved, and our bag-of-words model had no way of knowing. We patched it, retrained it, and watched it drift again. Around the same time, a colleague handed me a paper on word embeddings and said, "I think this is what we actually need." He was right, but it took me an embarrassingly long time to connect the dots between the classical NLP I knew and the semantic, representation-based thinking that paper described. That gap — between what most practitioners learn first and what modern NLP actually demands — is the reason I wrote this book.

Natural language processing has undergone a genuine transformation in a short span of time. The field has moved from hand-crafted feature engineering and sparse vector spaces to dense embeddings, transformer architectures, and large language models that can reason across context in ways that still feel remarkable. But here is the problem I keep encountering in teams and codebases: the transition has not been clean. Many practitioners learned NLP through tokenization, TF-IDF, and scikit-learn pipelines, and those skills are still useful — genuinely useful — but they do not map neatly onto the mental models required to work effectively with modern LLM workflows. The result is a kind of conceptual split: people who know the classical foundations but feel uncertain around embeddings and prompting, and people who have jumped straight into

LLM APIs but lack the grounding to debug, evaluate, or scale what they build. This book is my attempt to bridge that split in a practical, code-first way.

This book is written for Python developers and data practitioners who work with text — whether that means building classifiers, search systems, chatbots, document extraction pipelines, or anything else where language is the raw material. I assume you are comfortable writing Python, familiar with libraries like NumPy and pandas, and have at least a passing acquaintance with machine learning concepts. I do not assume any prior NLP experience. If you have some, you will move through the early chapters quickly and find the later material a natural extension of what you already know. If you are newer to the field, the progression is designed to carry you from first principles to production-ready thinking without leaving gaps that will catch you later.

The philosophy behind this book is straightforward: understanding should precede abstraction. It is tempting, especially in a field moving as fast as NLP, to reach immediately for the highest-level tool available. But when that tool breaks — and it will break — you need to know what is happening underneath. So we start with the fundamentals: how text becomes tokens, how tokens become numbers, how those numbers carry meaning. We build classical pipelines not because they are the final destination, but because they teach you to think carefully about representation, about what information you are preserving and what you are throwing away. From there, the move to embeddings and transformer-based models feels like a natural evolution rather than a mysterious leap.

The book is organized into four broad parts. The first part covers the foundations of text processing — cleaning and normalizing raw text, tokenization strategies, and the classical vectorization approaches that shaped the field. You will build working pipelines here, and more importantly, you will develop the vocabulary and intuition that the rest of the book depends on. The second part moves into the representation layer: word embeddings, sentence embeddings, and the semantic similarity workflows that make modern search and retrieval possible. This is where the shift in thinking becomes most visible, and I have tried to make that shift explicit rather than glossing over it. The third part focuses on practical NLP tasks — text classification, named entity recognition, information extraction, and document similarity — approached with both classical and modern methods so you can see the trade-offs clearly. The fourth and final part addresses LLM workflows directly: working with language model APIs, building retrieval-augmented generation pipelines, structuring prompts for reliable outputs, and thinking seriously about evaluation and production concerns. Each part builds on the previous one, but the chapters are written to be

reasonably self-contained if you need to jump to a specific topic.

By the time you finish this book, you should be able to build end-to-end NLP pipelines from raw text to reliable, deployable results. You will understand how to choose between approaches based on your constraints — data size, latency, interpretability, cost — rather than defaulting to whatever is newest or most impressive. You will be able to work with LLMs as one powerful tool among several, rather than treating them as either magic or mystery.

A word on scope: this book does not attempt to be a comprehensive survey of NLP research, and it does not go deep into training large models from scratch. That is a deliberate choice. The vast majority of practitioners working with text today are consumers and integrators of models, not trainers of them, and the skills that matter most in that context are different from what a research-focused curriculum would emphasize. I have also kept the mathematical exposition light — present where it builds genuine understanding, absent where it would just add noise. There are excellent books for those who want to go deeper into the theory, and I point toward them where it seems useful.

What I hope you take from this book, more than any specific technique, is a way of thinking about text problems — methodical, skeptical, grounded in how language actually works, and ready to evolve as the tools continue to change. The field will keep moving. The foundations, if you build them well, will hold. Let's get started.

To all the lifelong learners, who never stop exploring, questioning, and growing.

To those who dare to teach themselves, and then teach others.

This book is for you.

Contents

CHAPTER 1

THE NLP LANDSCAPE: FROM RULES TO LLMS

1.1 The NLP Landscape: From Rules to LLMs

Imagine asking a computer to read a doctor's handwritten notes, extract every medication mentioned, flag any dangerous drug interactions, and summarize the findings in plain English — all in under a second. A decade ago, building such a system required months of painstaking work by teams of linguists, software engineers, and domain experts. Today, a skilled Python developer can prototype a credible version of that system in an afternoon, leaning on a stack of open-source libraries and pre-trained models that encode knowledge distilled from billions of words of text. How did we get here? And more importantly, what do you actually need to understand to build these systems reliably, maintain them in production, and know when the shiny new approach is the wrong tool for the job? That is precisely what this chapter — and this book — sets out to answer.

1.1.1 What Is Natural Language Processing?

Natural Language Processing, almost universally abbreviated as NLP, is the subfield of artificial intelligence concerned with enabling computers to read, understand, generate, and reason about human language. The word *natural* is doing important work in that definition: it distinguishes the messy, ambiguous, context-dependent language that humans actually use from the formal, unambiguous languages that computers were originally designed to process. Programming languages have grammars that permit exactly one interpretation of any syntactically valid statement. English, Mandarin, Arabic, and every other human language routinely produce sentences that are simultaneously grammatical and deeply ambiguous, where meaning shifts with context, tone, culture, and shared history.

Consider the sentence "I saw the man with the telescope." Did the speaker use a telescope to see the man, or did the man happen to be carrying a telescope? Both readings are grammatically valid. Humans resolve this ambiguity almost instantly using world knowledge, conversational context, and probabilistic reasoning built up over years of language exposure. Teaching a machine to perform the same feat — at scale, across dozens of languages, for millions of users simultaneously — is the central challenge of NLP.

The practical applications of NLP are everywhere. Search engines parse queries and rank documents. Email clients filter spam and suggest replies. Voice assistants transcribe speech and interpret commands. Customer service platforms route tickets and draft responses. Medical systems extract diagnoses from clinical notes. Financial analysts use NLP to parse earnings calls and detect sentiment in news feeds. The field has moved from academic curiosity to critical infrastructure in less than three decades, and the pace of change is accelerating.

Example: Gmail's Smart Reply feature, introduced in 2015, uses NLP to analyze the content of incoming emails and suggest short, contextually appropriate responses such as "Sounds great!" or "I'll look into it." By 2019, Smart Reply was responsible for roughly 10% of all replies sent from Gmail on mobile devices — a clear demonstration that NLP had crossed from research prototype into mass-market utility.

1.1.2 The Three Paradigms of NLP

The history of NLP can be organized into three broad paradigms, each representing a fundamentally different answer to the question of how a machine should process language. These paradigms did not simply replace one another; they layered and interacted, and all three remain relevant in production systems today. Understanding why each paradigm emerged, what problems it solved, and where it fell short is not merely historical curiosity — it is the conceptual foundation you need to make good engineering decisions.

The Rule-Based Era

The earliest NLP systems were built on explicit, hand-crafted rules written by human experts. A linguist would study a language, identify its patterns, encode those patterns as formal rules, and then test the system against real text. If the system failed, the linguist would add more rules or refine existing ones. This approach is sometimes called *symbolic NLP* because it represents language as structured symbols manipulated according to logical rules.

Rule-based systems had genuine strengths. They were transparent: you could inspect every rule and understand exactly why the system produced a given output. They were deterministic: the same input always produced the same output. They required no training data, which was a significant advantage in an era when large digital text corpora simply did not exist. And for narrow, well-defined tasks in controlled domains, they could achieve very high precision.

The most influential early example is ELIZA, created by Joseph Weizenbaum at MIT in 1966. ELIZA simulated a psychotherapist by applying pattern-matching rules to user input. If the user typed "I am feeling sad," ELIZA's rules might detect the pattern "I am feeling X" and respond with "Why do you think you are feeling X?" The system had no understanding of sadness, therapy, or human emotion; it was pure pattern substitution. Yet users frequently reported feeling genuinely heard and understood — a phenomenon Weizenbaum found so disturbing that he wrote a book cautioning against anthropomorphizing computers.

Example: Industrial-strength rule-based NLP survived well into the 2000s in the form of systems like IBM's UIMA (Unstructured Information Management Architecture) and commercial products built on it. Healthcare companies used hand-crafted rule sets to extract diagnoses, medications, and dosages from clinical notes. A typical medication extraction rule might specify: "Find a token matching a known drug name, look ahead for a numeric token followed by a unit token

such as mg or ml, and extract the combination as a dosage entity." For a closed vocabulary of known drugs in a standardized note format, such rules could achieve precision above 95%. The brittleness appeared the moment the format changed or a new drug name entered circulation.

The fundamental limitation of rule-based systems is *coverage*: language is too varied, too creative, and too context-dependent for any finite set of rules to handle gracefully. Every new domain, every new document type, every new writing style potentially required new rules. Maintenance costs grew superlinearly. And crucially, rule-based systems had no mechanism for learning from data — they could not improve themselves by observing more examples.

The Statistical Revolution

The statistical paradigm began gaining momentum in the late 1980s and exploded in the 1990s, driven by three converging forces: the digitization of large text corpora, the growth of computational power, and a theoretical shift toward probabilistic models of language. Instead of asking "What rules govern this language?", statistical NLP asked "What patterns appear frequently in large amounts of real text, and how can we use those patterns to make predictions?"

The foundational insight was that language could be modeled as a probability distribution over sequences of words. A *language model* assigns a probability to any sequence of words, capturing the intuition that "The cat sat on the mat" is far more likely than "The mat sat on the cat." Early statistical language models used *n-grams* — sequences of n consecutive words — to estimate these probabilities from corpus counts. A bigram model, for example, estimates the probability of each word given only the immediately preceding word.

Statistical NLP also gave rise to the *bag-of-words* representation, which became the workhorse of text classification for two decades. In a bag-of-words model, a document is represented as a vector of word counts, discarding all information about word order. Pair this representation with a machine learning classifier — Naive Bayes, logistic regression, or a support vector machine — and you have a system that can learn to classify documents as spam or not-spam, positive or negative sentiment, relevant or irrelevant, simply by observing labeled examples. The system learns which words are predictive of which categories without anyone writing a rule about it.

Case Study: Spam filtering is the canonical success story of statistical NLP. In the early 2000s, email spam had grown to represent the majority of all email traffic, and rule-based filters were losing the arms race against spammers who constantly modified their messages to evade detection.

Paul Graham's 2002 essay "A Plan for Spam" described a Bayesian classifier that learned from each user's personal corpus of spam and legitimate mail. Because the classifier updated its probabilities as it saw new examples, it adapted to new spamming tactics automatically. Within a few years, statistical spam filters had become standard in every major email platform, and the spam problem — while never fully solved — became manageable.

The statistical era also produced methods that remain in use today: *TF-IDF* (Term Frequency-Inverse Document Frequency) for measuring word importance in documents, *Hidden Markov Models* for sequence labeling tasks like part-of-speech tagging, and *Conditional Random Fields* for named entity recognition. These methods are not obsolete; they appear in production systems where interpretability, speed, or limited data make them preferable to neural approaches.

Key Insight

Statistical NLP did not eliminate the need for human expertise — it transformed its nature. Instead of writing rules about language, practitioners needed expertise in feature engineering: deciding which properties of text to measure and feed to a classifier. A skilled NLP engineer in 2005 spent much of their time crafting features like character n-grams, capitalization patterns, and syntactic indicators. This art of feature engineering was largely automated away by the neural revolution, but the underlying intuition — that the right representation of data is often more important than the choice of model — remains as true as ever.

The Neural Turn and the Rise of Large Language Models

The neural paradigm began with word embeddings and shallow neural networks in the early 2010s, deepened with recurrent architectures and convolutional networks in the mid-2010s, and culminated — so far — with the Transformer architecture introduced in 2017 and the explosion of large language models (LLMs) that followed. Each step in this progression represented not just an incremental improvement but a qualitative shift in what NLP systems could do.

The pivotal early contribution was *word2vec*, published by Tomas Mikolov and colleagues at Google in 2013. Word2vec trained a shallow neural network to predict words from their contexts (or contexts from their words), and the learned representations — called *word embeddings* — captured semantic relationships in a striking way. The famous demonstration was that the vector arithmetic `king - man + woman` produced a vector close to `queen`. For the first time, a model

had learned something that looked like semantic meaning directly from raw text, without any hand-crafted knowledge.

Recurrent Neural Networks (RNNs), and particularly Long Short-Term Memory networks (LSTMs), brought sequential modeling to NLP. Unlike bag-of-words models, LSTMs processed text word by word, maintaining a hidden state that could in principle capture long-range dependencies. They achieved state-of-the-art results on machine translation, sentiment analysis, and named entity recognition. But they were slow to train, difficult to parallelize, and struggled with very long sequences.

The Transformer architecture, introduced in the paper "Attention Is All You Need" by Vaswani et al. (2017), solved the parallelization problem by replacing recurrence with a mechanism called *self-attention*. Self-attention allows every position in a sequence to directly attend to every other position, computing relevance weights that determine how much each word should influence the representation of every other word. This made Transformers highly parallelizable on modern GPU hardware and capable of capturing long-range dependencies more effectively than LSTMs.

Example: BERT (Bidirectional Encoder Representations from Transformers), released by Google in 2018, demonstrated the power of pre-training a large Transformer on massive text corpora and then fine-tuning it on specific downstream tasks. Before BERT, achieving state-of-the-art results on a task like question answering required training a specialized architecture from scratch on task-specific data. After BERT, the recipe became: download a pre-trained model, add a small task-specific output layer, and fine-tune on a modest dataset. BERT improved the state of the art on eleven NLP benchmarks simultaneously — an unprecedented result that signaled the arrival of a new paradigm.

GPT-3, released by OpenAI in 2020, pushed the paradigm further still. With 175 billion parameters trained on hundreds of billions of words of text, GPT-3 demonstrated *few-shot learning*: the ability to perform new tasks from just a handful of examples provided in the input prompt, with no gradient updates at all. This was not fine-tuning; it was a model that had internalized so much knowledge about language and the world that it could generalize to new tasks through *in-context learning* alone. The practical implication was profound: tasks that previously required collecting labeled data, training a model, and deploying infrastructure could now be accomplished by writing a well-crafted prompt.

The subsequent years brought GPT-4, Claude, Gemini, LLaMA, Mistral, and a proliferating

ecosystem of open-source and commercial LLMs, each with different capability profiles, context window sizes, and cost structures. NLP had transformed from a specialized engineering discipline into something closer to a general-purpose reasoning infrastructure.

1.1.3 Why Python Dominates NLP

If you have spent any time in data science or machine learning, the dominance of Python will not surprise you. But it is worth understanding why Python became the lingua franca of NLP specifically, because that history explains the shape of the ecosystem you will be working in throughout this book.

Python's rise in NLP was not inevitable. In the 1990s and early 2000s, Perl was the dominant language for text processing, valued for its powerful regular expression support and its "there's more than one way to do it" philosophy. Java was the language of choice for production NLP systems, and frameworks like the Stanford NLP toolkit and Apache OpenNLP were Java-native. C and C++ were used for performance-critical components. Python existed but was considered a scripting language, not a serious platform for research or production NLP.

The turning point came from the machine learning community. NumPy (2006) provided efficient array operations. SciPy built scientific computing on top of it. Matplotlib enabled visualization. And then scikit-learn (2011) gave practitioners a clean, consistent API for machine learning that made it genuinely pleasant to experiment with different algorithms. When the deep learning revolution arrived, Theano, then TensorFlow, then PyTorch all chose Python as their primary interface. Researchers who wanted to use these frameworks had to use Python, and the network effects compounded rapidly.

For NLP specifically, the NLTK library (Natural Language Toolkit), originally developed by Steven Bird and Edward Loper at the University of Pennsylvania, made Python the natural choice for NLP education and research as early as 2001. NLTK provided tokenizers, stemmers, parsers, corpora, and a rich set of teaching materials — all in Python. When spaCy arrived in 2015 with industrial-strength performance and a clean API, it cemented Python's position for production NLP. The HuggingFace Transformers library, launched in 2018, then made state-of-the-art neural NLP accessible to any Python developer who could write a few lines of code.

Today, the Python NLP ecosystem is unmatched in breadth and depth. The table below summarizes the core libraries you will use throughout this book:

Library	Primary Use	Paradigm
NLTK	Text preprocessing, education, corpora	Statistical / Classical
spaCy	Production NLP pipelines, NER, parsing	Statistical + Neural
scikit-learn	Text classification, feature extraction	Statistical / ML
HuggingFace Transformers	Pre-trained Transformer models	Neural / LLM
Gensim	Topic modeling, word embeddings	Statistical / Neural
LangChain	LLM application orchestration	LLM
OpenAI Python SDK	GPT API access	LLM

Table 1.1: Core Python NLP libraries and their primary roles

Installing this environment is straightforward with pip or conda. The following snippet installs the core libraries and verifies that each is working with a minimal "hello world" check:

```
pip install nltk spacy scikit-learn transformers torch
python -m spacy download en_core_web_sm
```

```
# Verify NLTK
import nltk
nltk.download('punkt', quiet=True)
from nltk.tokenize import word_tokenize
print(word_tokenize("Hello, NLP world!"))
# ['Hello', ',', 'NLP', 'world', '!']

# Verify spaCy
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("Apple is looking at buying U.K. startup for $1 billion")
for ent in doc.ents:
    print(ent.text, ent.label_)
# Apple  ORG
# U.K.   GPE
# $1 billion  MONEY

```

```python
# Verify HuggingFace Transformers
from transformers import pipeline
classifier = pipeline("sentiment-analysis")
result = classifier("I love learning NLP with Python!")
print(result)
# [{'label': 'POSITIVE', 'score': 0.9998}]

# Verify scikit-learn
from sklearn.feature_extraction.text import TfidfVectorizer
corpus = ["NLP is fascinating", "Python makes NLP easy"]
vec = TfidfVectorizer()
X = vec.fit_transform(corpus)
print(X.shape)
# (2, 6)
```

1.1.4 Choosing the Right Approach

One of the most common mistakes practitioners make when entering the field is reaching for the most powerful available tool regardless of whether the task warrants it. Deploying a 70-billion-parameter LLM to classify customer emails into five categories when a logistic regression trained on 500 labeled examples would achieve 94% accuracy is not just wasteful — it introduces latency, cost, and operational complexity that can sink a project. Conversely, trying to build a nuanced document summarization system with hand-crafted rules will produce a brittle, unmaintainable nightmare. The art of NLP engineering lies in matching the approach to the constraints of the problem.

Four dimensions dominate this decision: *data availability*, *latency requirements*, *interpretability needs*, and *maintenance burden*. Rule-based systems require no training data and are highly interpretable, but they are expensive to build and maintain. Statistical models need labeled data (typically hundreds to thousands of examples) and offer reasonable interpretability through feature importance scores. Neural models, especially fine-tuned Transformers, need thousands to tens of thousands of labeled examples for fine-tuning but can leverage pre-training to reduce this

requirement substantially. LLMs accessed via prompting can work with zero labeled examples but introduce API costs, latency, and the challenge of prompt engineering.

Scenario: A legal technology company needs to extract clause types from commercial contracts — for example, identifying indemnification clauses, limitation-of-liability clauses, and governing-law clauses. They have a team of expert lawyers who understand the domain deeply but only 200 labeled contract examples. The contracts are highly structured and follow recognizable patterns. In this scenario, a hybrid approach makes sense: use rule-based patterns (regular expressions and keyword lists curated by the lawyers) as a first pass, then use a fine-tuned BERT model to handle the ambiguous cases the rules miss. The rules handle 80% of cases with near-perfect precision and are fully explainable to clients; the neural component handles edge cases without requiring the lawyers to write increasingly baroque rules.

The following table provides a practical decision framework:

Factor	Rule-Based	Statistical/ML	Neural/LLM
Labeled data needed	None	Hundreds–thousands	Thousands (fine-tune) / Zero (prompt)
Latency	Very low	Low	Medium–High
Interpretability	Very high	Medium	Low
Maintenance cost	High	Medium	Low–Medium
Handles novel inputs	Poorly	Moderately	Well
Domain adaptation	Manual rules	Retrain	Fine-tune or re-prompt
Compute cost	Negligible	Low	High

Table 1.2: Decision framework for choosing an NLP approach

It is also worth noting that these paradigms are not mutually exclusive. Modern production NLP systems are frequently pipelines that combine multiple approaches. A customer service platform might use a fast rule-based pre-filter to route obviously simple queries, a statistical classifier to categorize the remaining queries into topic buckets, and an LLM to generate the actual response for complex queries. Understanding all three paradigms is not academic breadth for its own sake — it is the practical toolkit you need to build systems that work in the real world.

Common Mistake

Do not treat LLMs as a universal replacement for all prior NLP techniques. LLMs are powerful but expensive, slow relative to classical methods, non-deterministic, and difficult to audit. For high-throughput, low-latency tasks — such as classifying millions of short texts per day — a fine-tuned small model or even a TF-IDF plus logistic regression pipeline will often outperform an LLM on cost-per-query by two to three orders of magnitude while delivering comparable accuracy. Always benchmark against simpler baselines before committing to a complex solution.

1.1.5 A Roadmap for This Book

This book is organized to build your NLP knowledge in layers, mirroring the historical progression of the field while keeping practical application at the center of every chapter. The early chapters cover the foundations: text preprocessing, tokenization, linguistic structure, and the classical statistical methods that still power many production systems. These chapters will give you the intuition and the hands-on Python skills to work with raw text at scale.

The middle chapters move into the neural era: word embeddings, sequence models, and the Transformer architecture. Rather than treating these as black boxes, we will open them up and examine how they work, why certain design choices were made, and what their failure modes look like. Understanding the internals of a Transformer is not just intellectually satisfying — it is practically necessary when you need to debug a model that is producing unexpected outputs or when you need to choose between architectural variants.

The later chapters focus on modern LLM workflows: prompt engineering, retrieval-augmented generation, fine-tuning, and the emerging patterns for building LLM-powered applications with frameworks like LangChain. By the time you reach those chapters, you will have the foundation to evaluate LLM capabilities critically rather than accepting vendor claims at face value.

Throughout, Python is the vehicle. Every concept is grounded in working code that you can run, modify, and extend. The goal is not to survey the field from a distance but to give you the practical skills and the conceptual depth to build real NLP systems — and to keep building them as the field continues to evolve at its remarkable pace.

Key Takeaways

- NLP has progressed through three major paradigms — rule-based, statistical, and neural/LLM-based systems — each representing a fundamentally different approach to processing language. All three paradigms remain relevant in production systems today.
- Understanding classical rule-based and statistical approaches builds the intuition necessary to diagnose why modern neural methods succeed and where they fail. The history of NLP is not obsolete knowledge; it is the conceptual foundation for good engineering decisions.
- Python has become the dominant language for NLP due to its rich ecosystem of libraries — including NLTK, spaCy, scikit-learn, and HuggingFace Transformers — and the network effects created by the broader machine learning and data science communities converging on the same platform.
- Choosing the right NLP approach requires balancing data availability, latency requirements, interpretability needs, and maintenance burden. LLMs are not universally superior; for many tasks, simpler and faster approaches deliver better value. Production systems frequently combine multiple paradigms in a single pipeline.
- The field of NLP is evolving rapidly, but the fundamentals of text representation, probabilistic modeling, and sequence processing remain stable. Mastering these foundations makes it far easier to evaluate and adopt new techniques as they emerge.

Exercises

1. **Paradigm Survey.** Identify three real-world NLP applications from your own experience — for example, a voice assistant, a spam filter, and a machine translation service. For each application, research or reason carefully about which paradigm (rule-based, statistical, or neural/LLM-based) most likely describes its current implementation. Write a brief paragraph for each application explaining your reasoning, citing any specific architectural clues such as the year the product was launched, known vendor disclosures, or performance characteristics that suggest a particular approach.
2. **Environment Setup.** Install the core NLP Python environment on your local machine or in a cloud notebook environment such as Google Colab. Your environment should include

spaCy with the `en_core_web_sm` model, NLTK with the punkt tokenizer data, HuggingFace Transformers with PyTorch, and scikit-learn. Verify each library by running the hello-world snippets provided in Section 1.4 of this chapter and confirming that the output matches the expected results. Document any installation issues you encounter and how you resolved them — this troubleshooting experience is itself valuable preparation for production work.

3. **Sentiment Detector Comparison.** Write a one-page technical comparison of two approaches to sentiment analysis: (a) a rule-based system that uses a hand-crafted lexicon of positive and negative words (such as the VADER lexicon available in NLTK) to score text, and (b) a modern LLM-based approach using the HuggingFace `pipeline("sentiment-analysis")` interface. Test both approaches on at least ten sentences that you construct to probe edge cases — for example, sentences with negation ("not bad at all"), sarcasm ("oh great, another Monday"), and domain-specific language. In your comparison, address accuracy on your test cases, approximate inference latency, computational cost, and what it would take to maintain or update each system if the domain shifted. Conclude with a recommendation for which approach you would choose for a specific use case of your choice, and justify that recommendation.

You've reached the end of the preview.

The complete edition contains all chapters, including:

- Full chapter content with detailed examples and exercises
- Glossary of key terms
- Appendices and reference material
- A comprehensive index

Get the full book at `alkademy.com`