

# NLP: A Primer

---

*A language is not just words. It's a culture, a tradition,  
a unification of a community,  
a whole history that creates what a community is.  
It's all embodied in a language.  
—Noam Chomsky*

Imagine a hypothetical person, John Doe. He's the CTO of a fast-growing technology startup. On a busy day, John wakes up and has this conversation with his digital assistant:

*John:* “How is the weather today?”

*Digital assistant:* “It is 37 degrees centigrade outside with no rain today.”

*John:* “What does my schedule look like?”

*Digital assistant:* “You have a strategy meeting at 4 p.m. and an all-hands at 5:30 p.m. Based on today's traffic situation, it is recommended you leave for the office by 8:15 a.m.”

*While he's getting dressed, John probes the assistant on his fashion choices:*

*John:* “What should I wear today?”

*Digital assistant:* “White seems like a good choice.”

You might have used smart assistants such as Amazon Alexa, Google Home, or Apple Siri to do similar things. We talk to these assistants not in a programming language, but in our natural language—the language we all communicate in. This natural language has been the primary medium of communication between humans since time immemorial. But computers can only process data in binary, i.e., 0s and 1s. While we can represent language data in binary, how do we make machines understand the

language? This is where natural language processing (NLP) comes in. It is an area of computer science that deals with methods to analyze, model, and understand human language. Every intelligent application involving human language has some NLP behind it. In this book, we'll explain what NLP is as well as how to use NLP to build and scale intelligent applications. Due to the open-ended nature of NLP problems, there are dozens of alternative approaches one can take to solve a given problem. This book will help you navigate this maze of options and suggests how to choose the best option based on your problem.

This chapter aims to give a quick primer of what NLP is before we start delving deeper into how to implement NLP-based solutions for different application scenarios. We'll start with an overview of numerous applications of NLP in real-world scenarios, then cover the various tasks that form the basis of building different NLP applications. This will be followed by an understanding of language from an NLP perspective and of why NLP is difficult. After that, we'll give an overview of heuristics, machine learning, and deep learning, then introduce a few commonly used algorithms in NLP. This will be followed by a walkthrough of an NLP application. Finally, we'll conclude the chapter with an overview of the rest of the topics in the book. **Figure 1-1** shows a preview of the organization of the chapters in terms of various NLP tasks and applications.

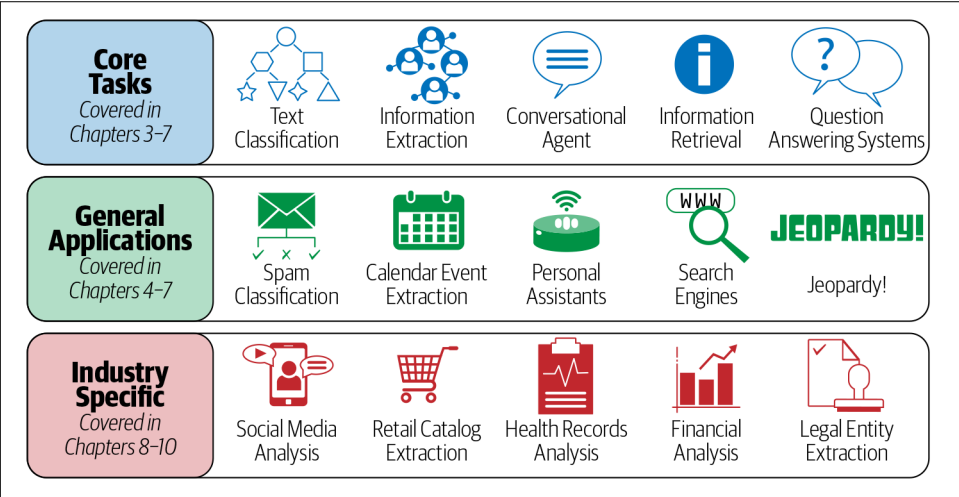


Figure 1-1. NLP tasks and applications

Let's start by taking a look at some popular applications you use in everyday life that have some form of NLP as a major component.

# NLP in the Real World

NLP is an important component in a wide range of software applications that we use in our daily lives. In this section, we'll introduce some key applications and also take a look at some common tasks that you'll see across different NLP applications. This section reinforces the applications we showed you in [Figure 1-1](#), which you'll see in more detail throughout the book.

Core applications:

- Email platforms, such as Gmail, Outlook, etc., use NLP extensively to provide a range of product features, such as spam classification, priority inbox, calendar event extraction, auto-complete, etc. We'll discuss some of these in detail in [Chapters 4 and 5](#).
- Voice-based assistants, such as Apple Siri, Google Assistant, Microsoft Cortana, and Amazon Alexa rely on a range of NLP techniques to interact with the user, understand user commands, and respond accordingly. We'll cover key aspects of such systems in [Chapter 6](#), where we discuss chatbots.
- Modern search engines, such as Google and Bing, which are the cornerstone of today's internet, use NLP heavily for various subtasks, such as query understanding, query expansion, question answering, information retrieval, and ranking and grouping of the results, to name a few. We'll discuss some of these subtasks in [Chapter 7](#).
- Machine translation services, such as Google Translate, Bing Microsoft Translator, and Amazon Translate are increasingly used in today's world to solve a wide range of scenarios and business use cases. These services are direct applications of NLP. We'll touch on machine translation in [Chapter 7](#).

Other applications:

- Organizations across verticals analyze their social media feeds to build a better and deeper understanding of the voice of their customers. We'll cover this in [Chapter 8](#).
- NLP is widely used to solve diverse sets of use cases on e-commerce platforms like Amazon. These vary from extracting relevant information from product descriptions to understanding user reviews. [Chapter 9](#) covers these in detail.
- Advances in NLP are being applied to solve use cases in domains such as health-care, finance, and law. [Chapter 10](#) addresses these.
- Companies such as Arria [1] are working to use NLP techniques to automatically generate reports for various domains, from weather forecasting to financial services.

- NLP forms the backbone of spelling- and grammar-correction tools, such as Grammarly and spell check in Microsoft Word and Google Docs.
- *Jeopardy!* is a popular quiz show on TV. In the show, contestants are presented with clues in the form of answers, and the contestants must phrase their responses in the form of questions. IBM built the Watson AI to compete with the show's top players. Watson won the first prize with a million dollars, more than the world champions. Watson AI was built using NLP techniques and is one of the examples of NLP bots winning a world competition.
- NLP is used in a range of learning and assessment tools and technologies, such as automated scoring in exams like the Graduate Record Examination (GRE), plagiarism detection (e.g., Turnitin), intelligent tutoring systems, and language learning apps like Duolingo.
- NLP is used to build large knowledge bases, such as the Google Knowledge Graph, which are useful in a range of applications like search and question answering.

This list is by no means exhaustive. NLP is increasingly being used across several other applications, and newer applications of NLP are coming up as we speak. Our main focus is to introduce you to the ideas behind building these applications. We do so by discussing different kinds of NLP problems and how to solve them. To get a perspective on what you are about to learn in this book, and to appreciate the nuances that go into building these NLP applications, let's take a look at some key NLP tasks that form the bedrock of many NLP applications and industry use cases.

## NLP Tasks

There is a collection of fundamental tasks that appear frequently across various NLP projects. Owing to their repetitive and fundamental nature, these tasks have been studied extensively. Having a good grip on them will make you ready to build various NLP applications across verticals. (We also saw some of these tasks earlier in [Figure 1-1](#).) Let's briefly introduce them:

### *Language modeling*

This is the task of predicting what the next word in a sentence will be based on the history of previous words. The goal of this task is to learn the probability of a sequence of words appearing in a given language. Language modeling is useful for building solutions for a wide variety of problems, such as speech recognition, optical character recognition, handwriting recognition, machine translation, and spelling correction.

### *Text classification*

This is the task of bucketing the text into a known set of categories based on its content. Text classification is by far the most popular task in NLP and is used in a variety of tools, from email spam identification to sentiment analysis.

### *Information extraction*

As the name indicates, this is the task of extracting relevant information from text, such as calendar events from emails or the names of people mentioned in a social media post.

### *Information retrieval*

This is the task of finding documents relevant to a user query from a large collection. Applications like Google Search are well-known use cases of information retrieval.

### *Conversational agent*

This is the task of building dialogue systems that can converse in human languages. Alexa, Siri, etc., are some common applications of this task.

### *Text summarization*

This task aims to create short summaries of longer documents while retaining the core content and preserving the overall meaning of the text.

### *Question answering*

This is the task of building a system that can automatically answer questions posed in natural language.

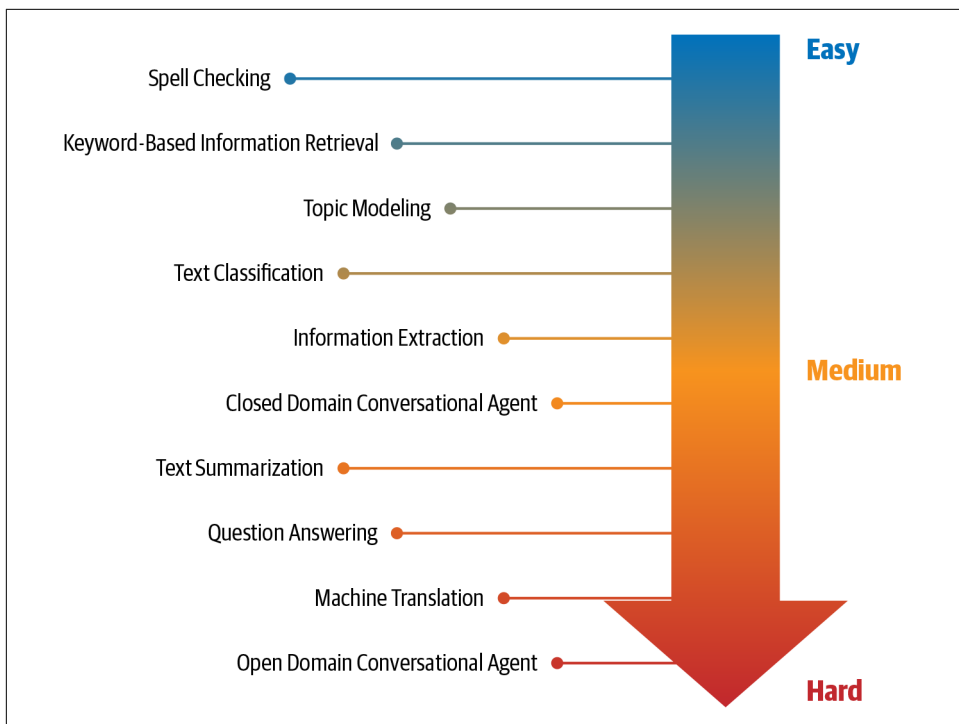
### *Machine translation*

This is the task of converting a piece of text from one language to another. Tools like Google Translate are common applications of this task.

### *Topic modeling*

This is the task of uncovering the topical structure of a large collection of documents. Topic modeling is a common text-mining tool and is used in a wide range of domains, from literature to bioinformatics.

Figure 1-2 shows a depiction of these tasks based on their relative difficulty in terms of developing comprehensive solutions.



*Figure 1-2. NLP tasks organized according to their relative difficulty*

In the rest of the chapters in this book, we'll see these tasks' challenges and learn how to develop solutions that work for certain use cases (even the hard tasks shown in the figure). To get there, it is useful to have an understanding of the nature of human language and the challenges in automating language processing. The next two sections provide a basic overview.

## What Is Language?

Language is a structured system of communication that involves complex combinations of its constituent components, such as characters, words, sentences, etc. Linguistics is the systematic study of language. In order to study NLP, it is important to understand some concepts from linguistics about how language is structured. In this section, we'll introduce them and cover how they relate to some of the NLP tasks we listed earlier.

We can think of human language as composed of four major building blocks: phonemes, morphemes and lexemes, syntax, and context. NLP applications need knowledge of different levels of these building blocks, starting from the basic sounds of language (phonemes) to texts with some meaningful expressions (context).

Figure 1-3 shows these building blocks of language, what they encompass, and a few NLP applications we introduced earlier that require this knowledge. Some of the terms listed here that were not introduced earlier in this chapter (e.g., parsing, word embeddings, etc.) will be introduced later in these first three chapters.

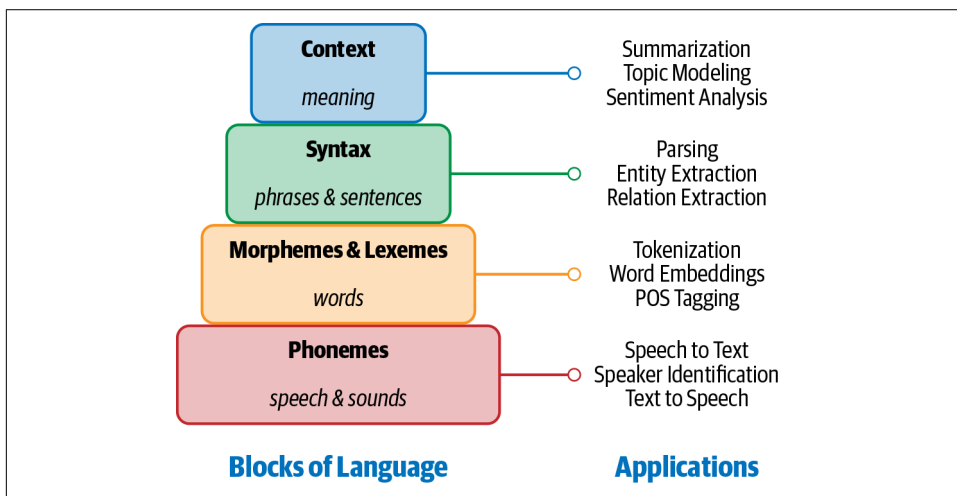


Figure 1-3. Building blocks of language and their applications

## Building Blocks of Language

Let's first introduce what these blocks of language are to give context for the challenges involved in NLP.

### Phonemes

Phonemes are the smallest units of sound in a language. They may not have any meaning by themselves but can induce meanings when uttered in combination with other phonemes. For example, standard English has 44 phonemes, which are either single letters or a combination of letters [2]. Figure 1-4 shows these phonemes along with sample words. Phonemes are particularly important in applications involving speech understanding, such as speech recognition, speech-to-text transcription, and text-to-speech conversion.

Consonant phonemes, with sample words		Vowel phonemes, with sample words	
1. /b/ - bat	13. /s/ - sun	1. /a/ - ant	13. /oi/ - coin
2. /k/ - cat	14. /t/ - tap	2. /e/ - egg	14. /ar/ - farm
3. /d/ - dog	15. /v/ - van	3. /i/ - in	15. /or/ - for
4. /f/ - fan	16. /w/ - wig	4. /o/ - on	16. /ur/ - hurt
5. /g/ - go	17. /y/ - yes	5. /u/ - up	17. /air/ - fair
6. /h/ - hen	18. /z/ - zip	6. /ai/ - rain	18. /ear/ - dear
7. /j/ - jet	19. /sh/ - shop	7. /ee/ - feet	19. /ure/ <sup>4</sup> - sure
8. /l/ - leg	20. /ch/ - chip	8. /igh/ - night	20. /ə/ - corner (the 'schwa' - an unstressed vowel sound which is close to /u/)
9. /m/ - map	21. /th/ - thin	9. /oa/ - boat	
10. /n/ - net	22. /th/ - then	10. /oo/ - boot	
11. /p/ - pen	23. /ng/ - ring	11. /oo/ - look	
12. /r/ - rat	24. /zh/ <sup>3</sup> - vision	12. /ow/ - cow	

Figure 1-4. Phonemes and examples

## Morphemes and lexemes

A morpheme is the smallest unit of language that has a meaning. It is formed by a combination of phonemes. Not all morphemes are words, but all prefixes and suffixes are morphemes. For example, in the word “multimedia,” “multi-” is not a word but a prefix that changes the meaning when put together with “media.” “Multi-” is a morpheme. **Figure 1-5** illustrates some words and their morphemes. For words like “cats” and “unbreakable,” their morphemes are just constituents of the full word, whereas for words like “tumbling” and “unreliability,” there is some variation when breaking the words down into their morphemes.

unbreakable <i>un + break + able</i>	cats <i>cat + s</i>
tumbling <i>tumble + ing</i>	unreliability <i>un + rely + able + ity</i>

Figure 1-5. Morpheme examples

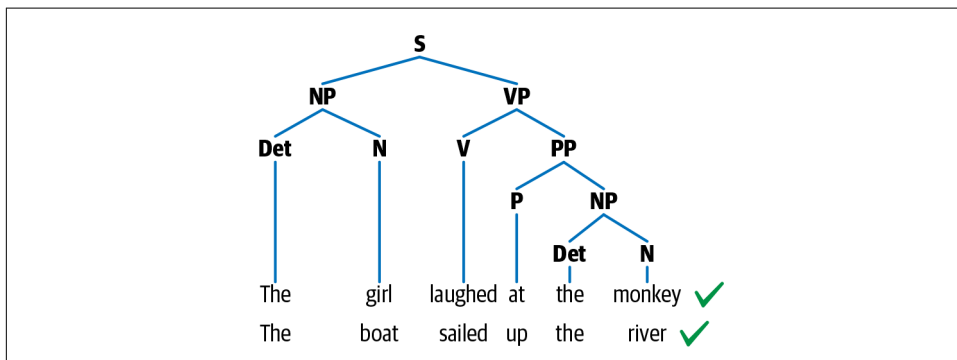
Lexemes are the structural variations of morphemes related to one another by meaning. For example, “run” and “running” belong to the same lexeme form. Morphological analysis, which analyzes the structure of words by studying its morphemes and lexemes, is a foundational block for many NLP tasks, such as tokenization, stemming,



learning word embeddings, and part-of-speech tagging, which we'll introduce in the next chapter.

## Syntax

Syntax is a set of rules to construct grammatically correct sentences out of words and phrases in a language. Syntactic structure in linguistics is represented in many different ways. A common approach to representing sentences is a parse tree. [Figure 1-6](#) shows an example parse tree for two English sentences.



*Figure 1-6. Syntactic structure of two syntactically similar sentences*

This has a hierarchical structure of language, with words at the lowest level, followed by part-of-speech tags, followed by phrases, and ending with a sentence at the highest level. In [Figure 1-6](#), both sentences have a similar structure and hence a similar syntactic parse tree. In this representation, N stands for noun, V for verb, and P for preposition. Noun phrase is denoted by NP and verb phrase by VP. The two noun phrases are “The girl” and “The boat,” while the two verb phrases are “laughed at the monkey” and “sailed up the river.” The syntactic structure is guided by a set of grammar rules for the language (e.g., the sentence comprises an NP and a VP), and this in turn guides some of the fundamental tasks of language processing, such as parsing. Parsing is the NLP task of constructing such trees automatically. Entity extraction and relation extraction are some of the NLP tasks that build on this knowledge of parsing, which we’ll discuss in more detail in [Chapter 5](#). Note that the parse structure described above is specific to English. The syntax of one language can be very different from that of another language, and the language-processing approaches needed for that language will change accordingly.

## Context

Context is how various parts in a language come together to convey a particular meaning. Context includes long-term references, world knowledge, and common sense along with the literal meaning of words and phrases. The meaning of a sentence can change based on the context, as words and phrases can sometimes have multiple meanings. Generally, context is composed from semantics and pragmatics. Semantics is the direct meaning of the words and sentences without external context. Pragmatics adds world knowledge and external context of the conversation to enable us to infer implied meaning. Complex NLP tasks such as sarcasm detection, summarization, and topic modeling are some of tasks that use context heavily.

Linguistics is the study of language and hence is a vast area in itself, and we only introduced some basic ideas to illustrate the role of linguistic knowledge in NLP. Different tasks in NLP require varying degrees of knowledge about these building blocks of language. An interested reader can refer to the books written by Emily Bender [3, 4] on the linguistic fundamentals for NLP for further study. Now that we have some idea of what the building blocks of language are, let's see why language can be hard for computers to understand and what makes NLP challenging.

## Why Is NLP Challenging?

What makes NLP a challenging problem domain? The ambiguity and creativity of human language are just two of the characteristics that make NLP a demanding area to work in. This section explores each characteristic in more detail, starting with ambiguity of language.

### Ambiguity

Ambiguity means uncertainty of meaning. Most human languages are inherently ambiguous. Consider the following sentence: “I made her duck.” This sentence has multiple meanings. The first one is: I cooked a duck for her. The second meaning is: I made her bend down to avoid an object. (There are other possible meanings, too; we'll leave them for the reader to think of.) Here, the ambiguity comes from the use of the word “made.” Which of the two meanings applies depends on the context in which the sentence appears. If the sentence appears in a story about a mother and a child, then the first meaning will probably apply. But if the sentence appears in a book about sports, then the second meaning will likely apply. The example we saw is a direct sentence.

When it comes to figurative language—i.e., idioms—the ambiguity only increases. For example, “He is as good as John Doe.” Try to answer, “How good is he?” The answer depends on how good John Doe is. [Figure 1-7](#) shows some examples illustrating ambiguity in language.

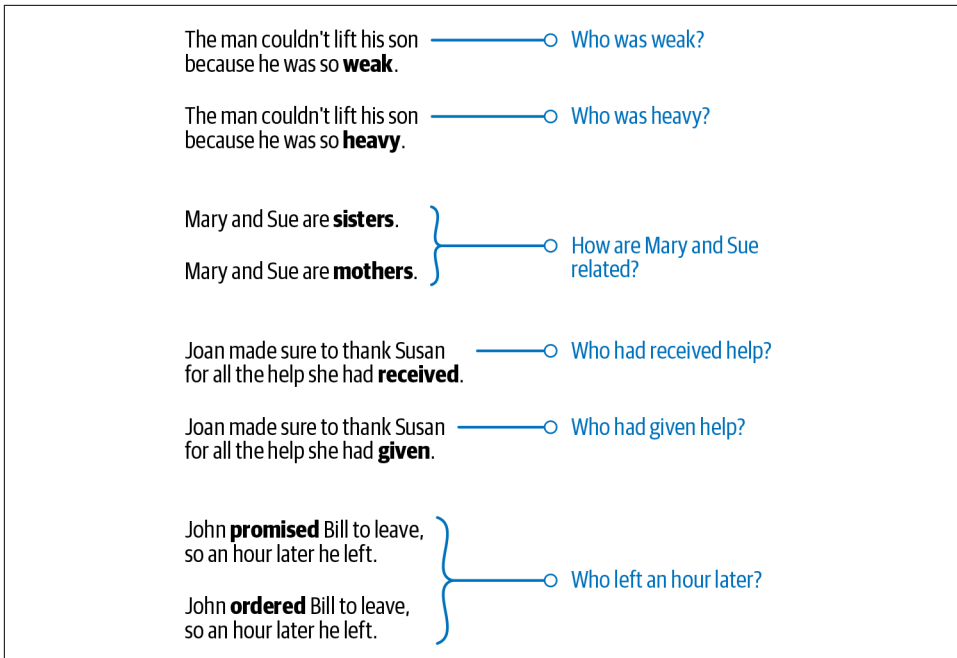


Figure 1-7. Examples of ambiguity in language from the Winograd Schema Challenge

The examples come from the Winograd Schema Challenge [5], named after Professor Terry Winograd of Stanford University. This schema has pairs of sentences that differ by only a few words, but the meaning of the sentences is often flipped because of this minor change. These examples are easily disambiguated by a human but are not solvable using most NLP techniques. Consider the pairs of sentences in the figure and the questions associated with them. With some thought, how the answer changes should be apparent based on a single word variation. As another experiment, consider taking an off-the-shelf NLP system like Google Translate and try various examples to see how such ambiguities affect (or don't affect) the output of the system.

## Common knowledge

A key aspect of any human language is “common knowledge.” It is the set of all facts that most humans are aware of. In any conversation, it is assumed that these facts are known, hence they're not explicitly mentioned, but they do have a bearing on the meaning of the sentence. For example, consider two sentences: “man bit dog” and “dog bit man.” We all know that the first sentence is unlikely to happen, while the second one is very possible. Why do we say so? Because we all “know” that it is very unlikely that a human will bite a dog. Further, dogs are known to bite humans. This knowledge is required for us to say that the first sentence is unlikely to happen while the second one is possible. Note that this common knowledge was not mentioned in

either sentence. Humans use common knowledge all the time to understand and process any language. In the above example, the two sentences are syntactically very similar, but a computer would find it very difficult to differentiate between the two, as it lacks the common knowledge humans have. One of the key challenges in NLP is how to encode all the things that are common knowledge to humans in a computational model.

## Creativity

Language is not just rule driven; there is also a creative aspect to it. Various styles, dialects, genres, and variations are used in any language. Poems are a great example of creativity in language. Making machines understand creativity is a hard problem not just in NLP, but in AI in general.

## Diversity across languages

For most languages in the world, there is no direct mapping between the vocabularies of any two languages. This makes porting an NLP solution from one language to another hard. A solution that works for one language might not work at all for another language. This means that one either builds a solution that is language agnostic or that one needs to build separate solutions for each language. While the first one is conceptually very hard, the other is laborious and time intensive.

All these issues make NLP a challenging—yet rewarding—domain to work in. Before looking into how some of these challenges are tackled in NLP, we should know the common approaches to solving NLP problems. Let's start with an overview of how machine learning and deep learning are connected to NLP before delving deeper into different approaches to NLP.

# Machine Learning, Deep Learning, and NLP: An Overview

Loosely speaking, artificial intelligence (AI) is a branch of computer science that aims to build systems that can perform tasks that require human intelligence. This is sometimes also called “machine intelligence.” The foundations of AI were laid in the 1950s at a workshop organized at Dartmouth College [6]. Initial AI was largely built out of logic-, heuristics-, and rule-based systems. Machine learning (ML) is a branch of AI that deals with the development of algorithms that can learn to perform tasks automatically based on a large number of examples, without requiring handcrafted rules. Deep learning (DL) refers to the branch of machine learning that is based on artificial neural network architectures. ML, DL, and NLP are all subfields within AI, and the relationship between them is depicted in [Figure 1-8](#).

While there is some overlap between NLP, ML, and DL, they are also quite different areas of study, as the figure illustrates. Like other early work in AI, early NLP applications were also based on rules and heuristics. In the past few decades, though, NLP

application development has been heavily influenced by methods from ML. More recently, DL has also been frequently used to build NLP applications. Considering this, let's do a short overview of ML and DL in this section.

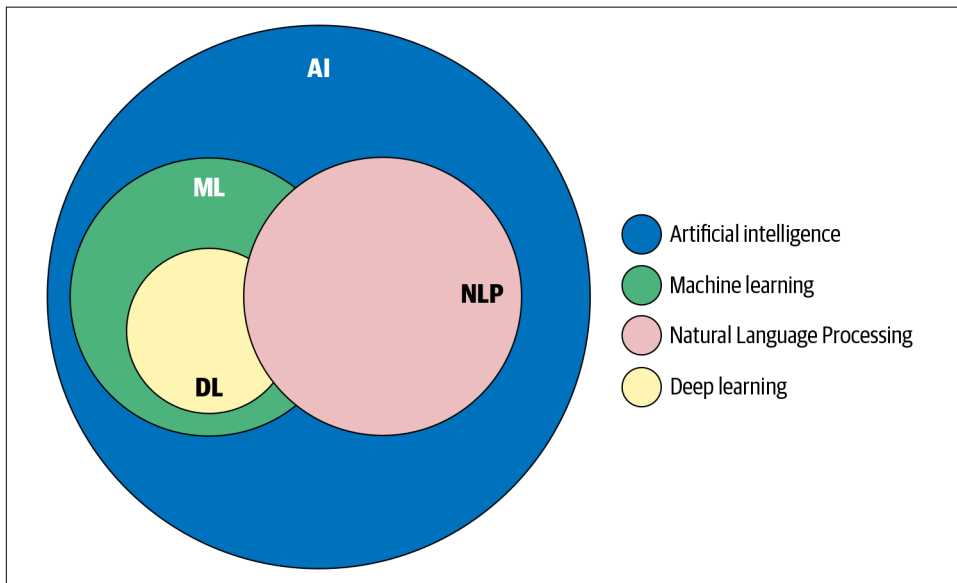


Figure 1-8. How NLP, ML, and DL are related

The goal of ML is to “learn” to perform tasks based on examples (called “training data”) without explicit instruction. This is typically done by creating a numeric representation (called “features”) of the training data and using this representation to learn the patterns in those examples. Machine learning algorithms can be grouped into three primary paradigms: supervised learning, unsupervised learning, and reinforcement learning. In supervised learning, the goal is to learn the mapping function from input to output given a large number of examples in the form of input-output pairs. The input-output pairs are known as *training data*, and the outputs are specifically known as *labels* or *ground truth*. An example of a supervised learning problem related to language is learning to classify email messages as spam or non-spam given thousands of examples in both categories. This is a common scenario in NLP, and we’ll see examples of supervised learning throughout the book, especially in [Chapter 4](#).

Unsupervised learning refers to a set of machine learning methods that aim to find hidden patterns in given input data without any reference output. That is, in contrast to supervised learning, unsupervised learning works with large collections of unlabeled data. In NLP, an example of such a task is to identify latent topics in a large collection of textual data without any knowledge of these topics. This is known as *topic modeling*, and we’ll discuss it in [Chapter 7](#).

Common in real-world NLP projects is a case of semi-supervised learning, where we have a small labeled dataset and a large unlabeled dataset. Semi-supervised techniques involve using both datasets to learn the task at hand. Last but not least, reinforcement learning deals with methods to learn tasks via trial and error and is characterized by the absence of either labeled or unlabeled data in large quantities. The learning is done in a self-contained environment and improves via feedback (reward or punishment) facilitated by the environment. This form of learning is not common in applied NLP (yet). It is more common in applications such as machine-playing games like go or chess, in the design of autonomous vehicles, and in robotics.

Deep learning refers to the branch of machine learning that is based on artificial neural network architectures. The ideas behind neural networks are inspired by neurons in the human brain and how they interact with one another. In the past decade, deep learning-based neural architectures have been used to successfully improve the performance of various intelligent applications, such as image and speech recognition and machine translation. This has resulted in a proliferation of deep learning-based solutions in industry, including in NLP applications.

Throughout this book, we'll discuss how all these approaches are used for developing various NLP applications. Let's now discuss the different approaches to solve any given NLP problem.

## Approaches to NLP

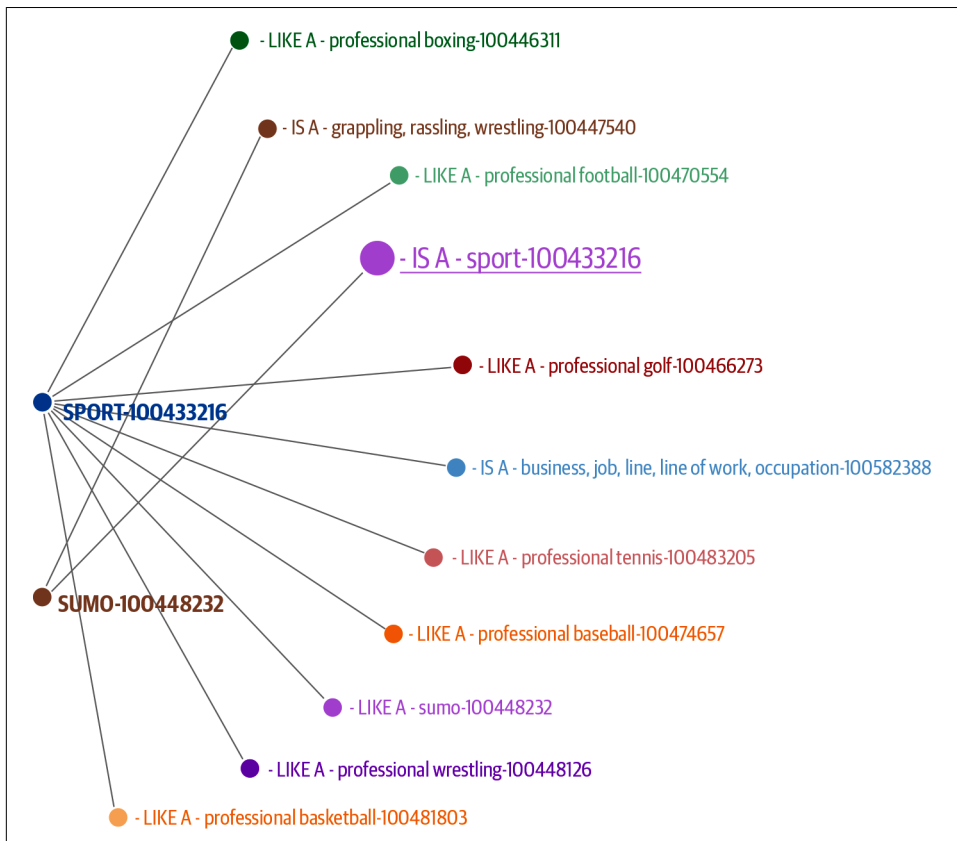
The different approaches used to solve NLP problems commonly fall into three categories: heuristics, machine learning, and deep learning. This section is simply an introduction to each approach—don't worry if you can't quite grasp the concepts yet, as they'll be discussed in detail throughout the rest of the book. Let's jump in by discussing heuristics-based NLP.

### Heuristics-Based NLP

Similar to other early AI systems, early attempts at designing NLP systems were based on building rules for the task at hand. This required that the developers had some expertise in the domain to formulate rules that could be incorporated into a program. Such systems also required resources like dictionaries and thesauruses, typically compiled and digitized over a period of time. An example of designing rules to solve an NLP problem using such resources is lexicon-based sentiment analysis. It uses counts of positive and negative words in the text to deduce the sentiment of the text. We'll cover this briefly in [Chapter 4](#).

Besides dictionaries and thesauruses, more elaborate knowledge bases have been built to aid NLP in general and rule-based NLP in particular. One example is Wordnet [7], which is a database of words and the semantic relationships between them. Some

examples of such relationships are synonyms, hyponyms, and meronyms. Synonyms refer to words with similar meanings. Hyponyms capture is-type-of relationships. For example, baseball, sumo wrestling, and tennis are all hyponyms of sports. Meronyms capture is-part-of relationships. For example, hands and legs are meronyms of the body. All this information becomes useful when building rule-based systems around language. **Figure 1-9** shows an example depiction of such relationships between words using Wordnet.



*Figure 1-9. Wordnet graph for the word “sport” [8]*

More recently, common sense world knowledge has also been incorporated into knowledge bases like Open Mind Common Sense [9], which also aids such rule-based systems. While what we’ve seen so far are largely lexical resources based on word-level information, rule-based systems go beyond words and can incorporate other forms of information, too. Some of them are introduced below.

Regular expressions (regex) are a great tool for text analysis and building rule-based systems. A regex is a set of characters or a pattern that is used to match and find

substrings in text. For example, a regex like `^([a-zA-Z0-9_-\.\.]+)@([a-zA-Z0-9_-\.\.]+\.[a-zA-Z]{2,5})$` is used to find all email IDs in a piece of text. Regexes are a great way to incorporate domain knowledge in your NLP system. For example, given a customer complaint that comes via chat or email, we want to build a system to automatically identify the product the complaint is about. There is a range of product codes that map to certain brand names. We can use regexes to match these easily.

Regexes are a very popular paradigm for building rule-based systems. NLP software like StanfordCoreNLP includes `TokensRegex` [10], which is a framework for defining regular expressions. It is used to identify patterns in text and use matched text to create rules. Regexes are used for deterministic matches—meaning it's either a match or it's not. Probabilistic regexes is a sub-branch that addresses this limitation by including a probability of a match. Interested readers can look at software libraries such as `pregex` [11]. Last accessed June 15, 2020.

Context-free grammar (CFG) is a type of formal grammar that is used to model natural languages. CFG was invented by Professor Noam Chomsky, a renowned linguist and scientist. CFGs can be used to capture more complex and hierarchical information that a regex might not. The Earley parser [12] allows parsing of all kinds of CFGs. To model more complex rules, grammar languages like JAPE (Java Annotation Patterns Engine) can be used [13]. JAPE has features from both regexes as well as CFGs and can be used for rule-based NLP systems like GATE (General Architecture for Text Engineering) [14]. GATE is used for building text extraction for closed and well-defined domains where accuracy and completeness of coverage is more important. As an example, JAPE and GATE were used to extract information on pacemaker implantation procedures from clinical reports [15]. [Figure 1-10](#) shows the GATE interface along with several types of information highlighted in the text as an example of a rule-based system.

Rules and heuristics play a role across the entire life cycle of NLP projects even now. At one end, they're a great way to build first versions of NLP systems. Put simply, rules and heuristics help you quickly build the first version of the model and get a better understanding of the problem at hand. We'll discuss this point in depth in [Chapters 4 and 11](#). Rules and heuristics can also be useful as features for machine learning-based NLP systems. At the other end of the spectrum of the project life cycle, rules and heuristics are used to plug the gaps in the system. Any NLP system built using statistical, machine learning, or deep learning techniques will make mistakes. Some mistakes can be too expensive—for example, a healthcare system that looks into all the medical records of a patient and wrongly decides to not advise a critical test. This mistake could even cost a life. Rules and heuristics are a great way to plug such gaps in production systems. Now let's turn our attention to machine learning techniques used for NLP.



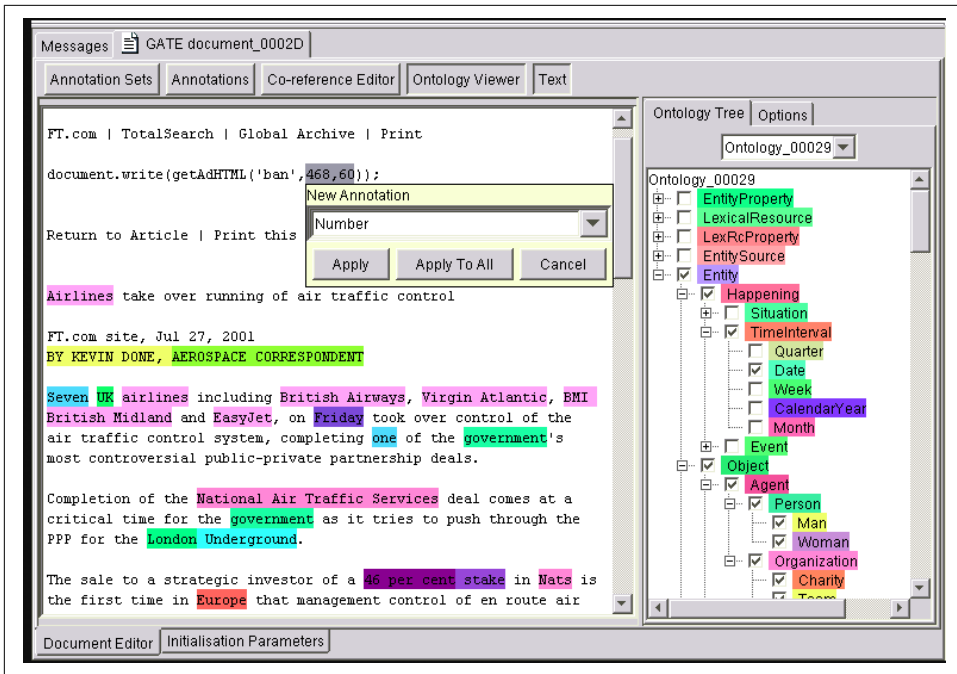


Figure 1-10. GATE tool

## Machine Learning for NLP

Machine learning techniques are applied to textual data just as they're used on other forms of data, such as images, speech, and structured data. Supervised machine learning techniques such as classification and regression methods are heavily used for various NLP tasks. As an example, an NLP classification task would be to classify news articles into a set of news topics like sports or politics. On the other hand, regression techniques, which give a numeric prediction, can be used to estimate the price of a stock based on processing the social media discussion about that stock. Similarly, unsupervised clustering algorithms can be used to club together text documents.

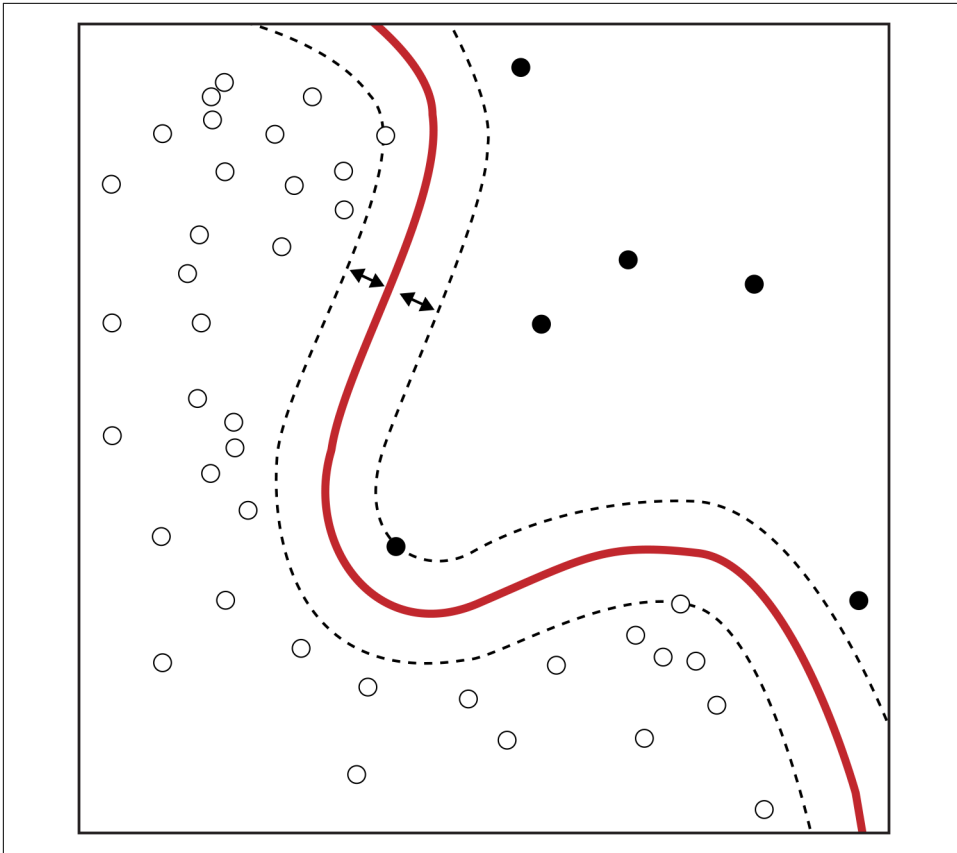
Any machine learning approach for NLP, supervised or unsupervised, can be described as consisting of three common steps: extracting features from text, using the feature representation to learn a model, and evaluating and improving the model. We'll learn more about feature representations for text specifically in [Chapter 3](#) and evaluation in [Chapter 2](#). We'll now briefly outline some of the commonly used supervised ML methods in NLP for the second step (using the feature representation to learn a model). Having a basic idea of these methods will help you understand the concepts discussed in later chapters.

## Naive Bayes

Naive Bayes is a classic algorithm for classification tasks [16] that mainly relies on Bayes' theorem (as is evident from the name). Using Bayes' theorem, it calculates the probability of observing a class label given the set of features for the input data. A characteristic of this algorithm is that it assumes each feature is independent of all other features. For the news classification example mentioned earlier in this chapter, one way to represent the text numerically is by using the count of domain-specific words, such as sport-specific or politics-specific words, present in the text. We assume that these word counts are not correlated to one another. If the assumption holds, we can use Naive Bayes to classify news articles. While this is a strong assumption to make in many cases, Naive Bayes is commonly used as a starting algorithm for text classification. This is primarily because it is simple to understand and very fast to train and run.

## Support vector machine

The support vector machine (SVM) is another popular classification [17] algorithm. The goal in any classification approach is to learn a decision boundary that acts as a separation between different categories of text (e.g., politics versus sports in our news classification example). This decision boundary can be linear or nonlinear (e.g., a circle). An SVM can learn both a linear and nonlinear decision boundary to separate data points belonging to different classes. A linear decision boundary learns to represent the data in a way that the class differences become apparent. For two-dimensional feature representations, an illustrative example is given in [Figure 1-11](#), where the black and white points belong to different classes (e.g., sports and politics news groups). An SVM learns an optimal decision boundary so that the distance between points across classes is at its maximum. The biggest strength of SVMs are their robustness to variation and noise in the data. A major weakness is the time taken to train and the inability to scale when there are large amounts of training data.



*Figure 1-11. A two-dimensional feature representation of an SVM*

### **Hidden Markov Model**

The hidden Markov model (HMM) is a statistical model [18] that assumes there is an underlying, unobservable process with hidden states that generates the data—i.e., we can only observe the data once it is generated. An HMM then tries to model the hidden states from this data. For example, consider the NLP task of part-of-speech (POS) tagging, which deals with assigning part-of-speech tags to sentences. HMMs are used for POS tagging of text data. Here, we assume that the text is generated according to an underlying grammar, which is hidden underneath the text. The hidden states are parts of speech that inherently define the structure of the sentence following the language grammar, but we only observe the words that are governed by these latent states. Along with this, HMMs also make the Markov assumption, which means that each hidden state is dependent on the previous state(s). Human language is sequential in nature, and the current word in a sentence depends on what occurred before it. Hence, HMMs with these two assumptions are a powerful tool for modeling textual

data. In **Figure 1-12**, we can see an example of an HMM that learns parts of speech from a given sentence. Parts of speech like JJ (adjective) and NN (noun) are hidden states, while the sentence “natural language processing ( nlp )...” is directly observed.

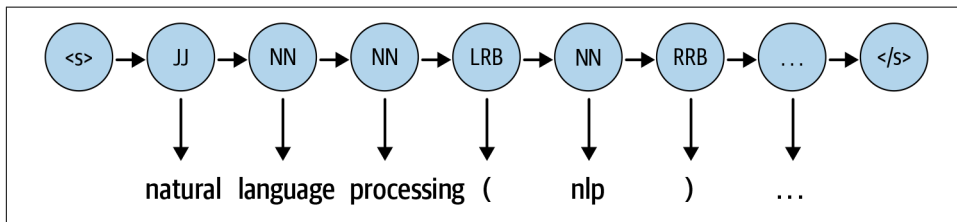


Figure 1-12. A graphical representation of a hidden Markov model

For a detailed discussion on HMMs for NLP, refer to **Chapter 8** in the book *Speech and Language Processing* by Professor Jurafsky [19].

### Conditional random fields

The conditional random field (CRF) is another algorithm that is used for sequential data. Conceptually, a CRF essentially performs a classification task on each element in the sequence [20]. Imagine the same example of POS tagging, where a CRF can tag word by word by classifying them to one of the parts of speech from the pool of all POS tags. Since it takes the sequential input and the context of tags into consideration, it becomes more expressive than the usual classification methods and generally performs better. CRFs outperform HMMs for tasks such as POS tagging, which rely on the sequential nature of language. We discuss CRFs and their variants along with applications in Chapters 5, 6, and 9.

These are some of the popular ML algorithms that are used heavily across NLP tasks. Having some understanding of these ML methods helps to understand various solutions discussed in the book. Apart from that, it is also important to understand when to use which algorithm, which we’ll discuss in the upcoming chapters. To learn more about other steps and further theoretical details of the machine learning process, we recommend the textbook *Pattern Recognition and Machine Learning* by Christopher Bishop [21]. For a more applied machine learning perspective, Aurélien Géron’s book [22] is a great resource to start with. Let’s now take a look at deep learning approaches to NLP.

## Deep Learning for NLP

We briefly touched on a couple of popular machine learning methods that are used heavily in various NLP tasks. In the last few years, we have seen a huge surge in using neural networks to deal with complex, unstructured data. Language is inherently complex and unstructured. Therefore, we need models with better representation and

learning capability to understand and solve language tasks. Here are a few popular deep neural network architectures that have become the status quo in NLP.

## Recurrent neural networks

As we mentioned earlier, language is inherently sequential. A sentence in any language flows from one direction to another (e.g., English reads from left to right). Thus, a model that can progressively read an input text from one end to another can be very useful for language understanding. Recurrent neural networks (RNNs) are specially designed to keep such sequential processing and learning in mind. RNNs have neural units that are capable of remembering what they have processed so far. This memory is temporal, and the information is stored and updated with every time step as the RNN reads the next word in the input. **Figure 1-13** shows an unrolled RNN and how it keeps track of the input at different time steps.

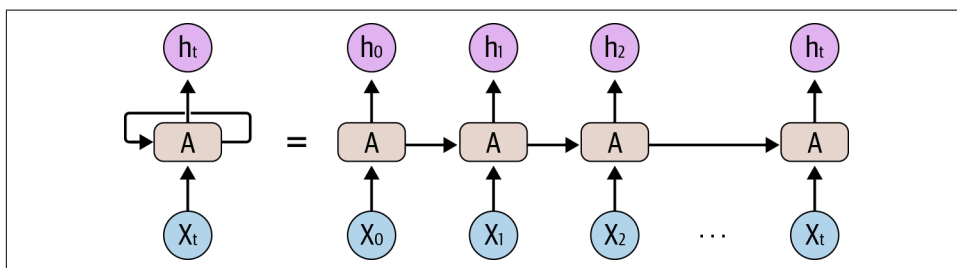


Figure 1-13. An unrolled recurrent neural network [23]

RNNs are powerful and work very well for solving a variety of NLP tasks, such as text classification, named entity recognition, machine translation, etc. One can also use RNNs to generate text where the goal is to read the preceding text and predict the next word or the next character. Refer to “The Unreasonable Effectiveness of Recurrent Neural Networks” [24] for a detailed discussion on the versatility of RNNs and the range of applications within and outside NLP for which they are useful.

## Long short-term memory

Despite their capability and versatility, RNNs suffer from the problem of forgetful memory—they cannot remember longer contexts and therefore do not perform well when the input text is long, which is typically the case with text inputs. Long short-term memory networks (LSTMs), a type of RNN, were invented to mitigate this shortcoming of the RNNs. LSTMs circumvent this problem by letting go of the irrelevant context and only remembering the part of the context that is needed to solve the task at hand. This relieves the load of remembering very long context in one vector representation. LSTMs have replaced RNNs in most applications because of this workaround. Gated recurrent units (GRUs) are another variant of RNNs that are used mostly in language generation. (The article written by Christopher Olah [23] covers

the family of RNN models in great detail.) **Figure 1-14** illustrates the architecture of a single LSTM cell. We'll discuss specific uses of LSTMs in various NLP applications in Chapters 4, 5, 6, and 9.

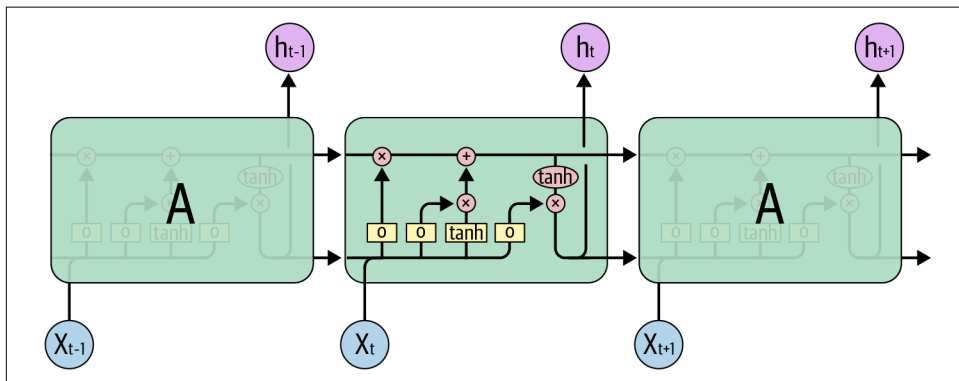


Figure 1-14. Architecture of an LSTM cell [23]

## Convolutional neural networks

Convolutional neural networks (CNNs) are very popular and used heavily in computer vision tasks like image classification, video recognition, etc. CNNs have also seen success in NLP, especially in text-classification tasks. One can replace each word in a sentence with its corresponding word vector, and all vectors are of the same size ( $d$ ) (refer to “Word Embeddings” in **Chapter 3**). Thus, they can be stacked one over another to form a matrix or 2D array of dimension  $n \times d$ , where  $n$  is the number of words in the sentence and  $d$  is the size of the word vectors. This matrix can now be treated similar to an image and can be modeled by a CNN. The main advantage CNNs have is their ability to look at a group of words together using a context window. For example, we are doing sentiment classification, and we get a sentence like, “I like this movie very much!” In order to make sense of this sentence, it is better to look at words and different sets of contiguous words. CNNs can do exactly this by definition of their architecture. We’ll touch on this in more detail in later chapters. **Figure 1-15** shows a CNN in action on a piece of text to extract useful phrases to ultimately arrive at a binary number indicating the sentiment of the sentence from a given piece of text.

As shown in the figure, CNN uses a collection of convolution and pooling layers to achieve this condensed representation of the text, which is then fed as input to a fully connected layer to learn some NLP tasks like text classification. More details on the usage CNNs for NLP can be found in [25] and [26]. We also cover them in **Chapter 4**.

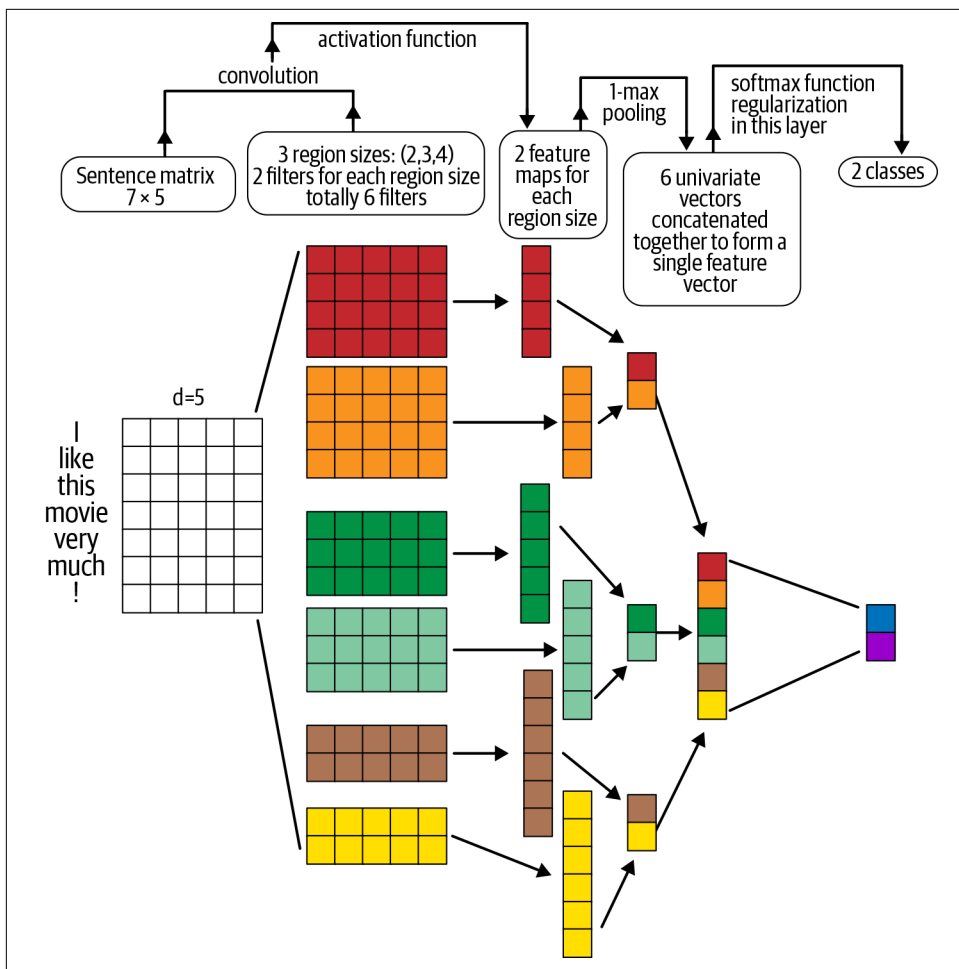


Figure 1-15. CNN model in action [27]

## Transformers

Transformers [28] are the latest entry in the league of deep learning models for NLP. Transformer models have achieved state of the art in almost all major NLP tasks in the past two years. They model the textual context but not in a sequential manner. Given a word in the input, it prefers to look at all the words around it (known as *self-attention*) and represent each word with respect to its context. For example, the word “bank” can have different meanings depending on the context in which it appears. If the context talks about finance, then “bank” probably denotes a financial institution. On the other hand, if the context mentions a river, then it probably indicates a bank of the river. Transformers can model such context and hence have been used heavily

in NLP tasks due to this higher representation capacity as compared to other deep networks.

Recently, large transformers have been used for *transfer learning* with smaller downstream tasks. Transfer learning is a technique in AI where the knowledge gained while solving one problem is applied to a different but related problem. With transformers, the idea is to train a very large transformer model in an unsupervised manner (known as *pre-training*) to predict a part of a sentence given the rest of the content so that it can encode the high-level nuances of the language in it. These models are trained on more than 40 GB of textual data, scraped from the whole internet. An example of a large transformer is BERT (Bidirectional Encoder Representations from Transformers) [29], shown in **Figure 1-16**, which is pre-trained on massive data and open sourced by Google.

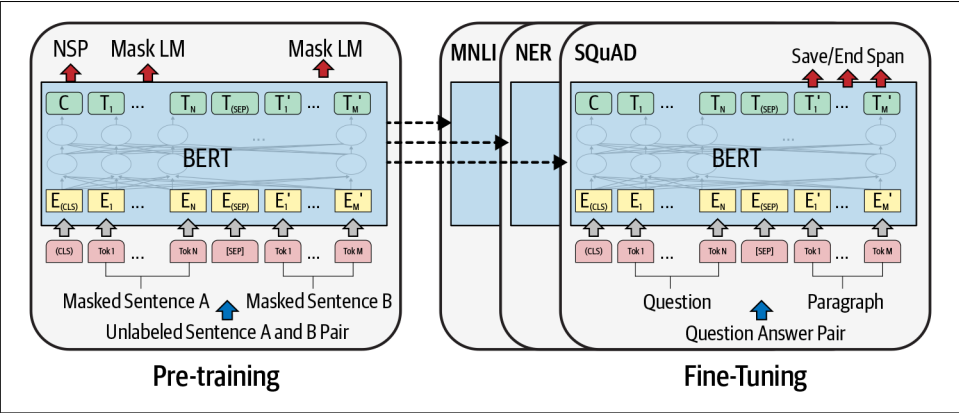


Figure 1-16. BERT architecture: pre-trained model and fine-tuned, task-specific models

The pre-trained model is shown on the left side of **Figure 1-16**. This model is then fine-tuned on downstream NLP tasks, such as text classification, entity extraction, question answering, etc., as shown on the right of **Figure 1-16**. Due to the sheer amount of pre-trained knowledge, BERT works efficiently in transferring the knowledge for downstream tasks and achieves state of the art for many of these tasks. Throughout the book, we have covered various examples of using BERT for various tasks. **Figure 1-17** illustrates the workings of a self-attention mechanism, which is a key component of a transformer. Interested readers can look at [30] for more details on self-attention mechanisms and transformer architecture. We cover BERT and its applications in Chapters 4, 6, and 10.



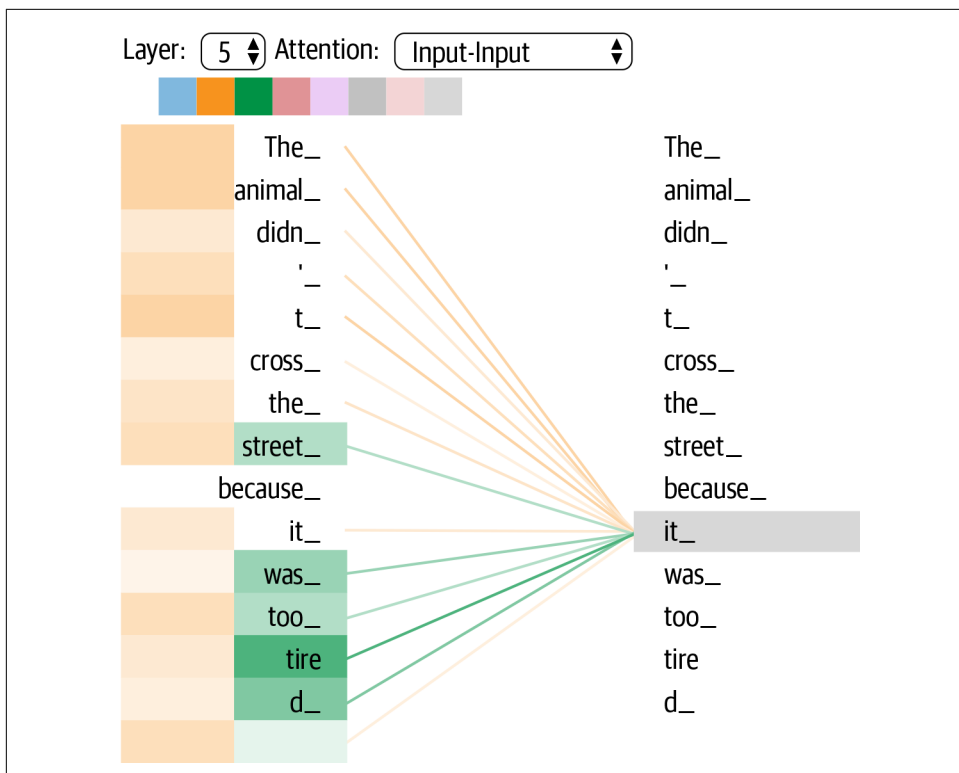


Figure 1-17. Self-attention mechanism in a transformer [30]

## Autoencoders

An autoencoder is a different kind of network that is used mainly for learning compressed vector representation of the input. For example, if we want to represent a text by a vector, what is a good way to do it? We can learn a mapping function from input text to the vector. To make this mapping function useful, we “reconstruct” the input back from the vector representation. This is a form of unsupervised learning since you don’t need human-annotated labels for it. After the training, we collect the vector representation, which serves as an encoding of the input text as a dense vector. Autoencoders are typically used to create feature representations needed for any downstream tasks. [Figure 1-18](#) depicts the architecture of an autoencoder.

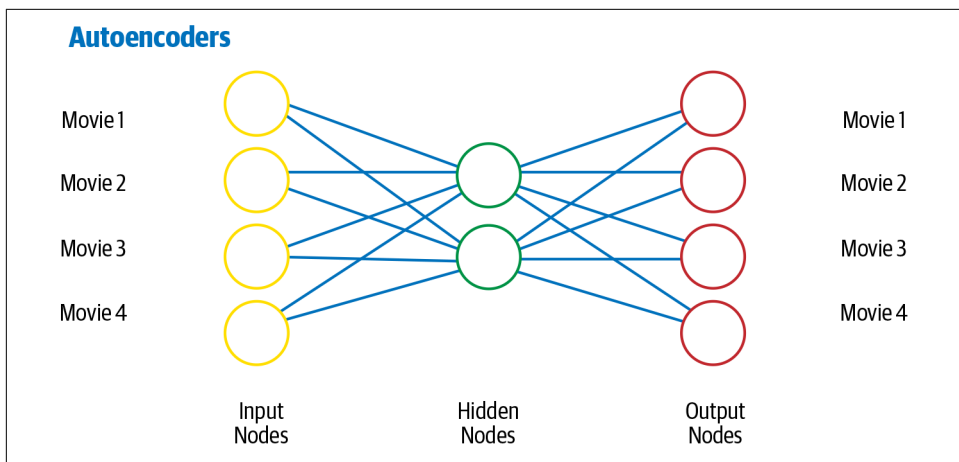


Figure 1-18. Architecture of an autoencoder

In this scheme, the hidden layer gives a compressed representation of input data, capturing the essence, and the output layer (decoder) reconstructs the input representation from the compressed representation. While the architecture of the autoencoder shown in [Figure 1-18](#) cannot handle specific properties of sequential data like text, variations of autoencoders, such as LSTM autoencoders, address these well. More information about autoencoders can be found in [31].

We briefly introduced some of the popular DL architectures for NLP here. For a more detailed study of deep learning architectures in general, refer to [31], and specifically for NLP, refer to [25]. We hope this introduction gives you enough background to understand the use of DL in the rest of this book.

Going by all the recent achievements of DL models, one might think that DL should be the go-to way to build NLP systems. However, that is far from the truth for most industry use cases. Let's look at why this is the case.

## Why Deep Learning Is Not Yet the Silver Bullet for NLP

Over the last few years, DL has made amazing advances in NLP. For example, in text classification, LSTM- and CNN-based models have surpassed the performance of standard machine learning techniques such as Naive Bayes and SVM for many classification tasks. Similarly, LSTMs have performed better in sequence-labeling tasks like entity extraction as compared to CRF models. Recently, powerful transformer models have become state of the art in most of these NLP tasks, ranging from classification to sequence labeling. A huge trend right now is to leverage large (in terms of number of parameters) transformer models, train them on huge datasets for generic NLP tasks like language models, then adapt them to smaller downstream tasks. This approach

(known as *transfer learning*) has also been successful in other domains, such as computer vision and speech.

Despite such tremendous success, DL is still not the silver bullet for all NLP tasks when it comes to industrial applications. Some of the key reasons for this are as follows:

#### *Overfitting on small datasets*

DL models tend to have more parameters than traditional ML models, which means they possess more expressivity. This also comes with a curse. Occam's razor [32] suggests that a simpler solution is always preferable given that all other conditions are equal. Many times, in the development phase, sufficient training data is not available to train a complex network. In such cases, a simpler model should be preferred over a DL model. DL models overfit on small datasets and subsequently lead to poor generalization capability, which in turn leads to poor performance in production.

#### *Few-shot learning and synthetic data generation*

In disciplines like computer vision, DL has made significant strides in few-shot learning (i.e., learning from very few training examples) [33] and in models that can generate superior-quality images [34]. Both of these advances have made training DL-based vision models on small amounts of data feasible. Therefore, DL has achieved much wider adoption for solving problems in industrial settings. We have not yet seen similar DL techniques be successfully developed for NLP.

#### *Domain adaptation*

If we utilize a large DL model that is trained on datasets originating from some common domains (e.g., news articles) and apply the trained model to a newer domain that is different from the common domains (e.g., social media posts), it may yield poor performance. This loss in generalization performance indicates that DL models are not always useful. For example, models trained on internet texts and product reviews will not work well when applied to domains such as law, social media, or healthcare, where both the syntactic and semantic structure of the language is specific to the domain. We need specialized models to encode the domain knowledge, which could be as simple as domain-specific, rule-based models.

#### *Interpretable models*

Apart from efficient domain adaptation, controllability and interpretability is hard for DL models because, most of the time, they work like a black box. Businesses often demand more interpretable results that can be explained to the customer or end user. In those cases, traditional techniques might be more useful. For example, a Naive Bayes model for sentiment classification may explain the effect of strong positive and negative words on the final prediction of sentiment.

As of today, obtaining such insights from an LSTM-based classification model is difficult. This is in contrast to computer vision, where DL models are not black boxes. There are plenty of techniques [35] in computer vision that are used to gain insight into why a model is making a particular prediction. Such approaches for NLP are not as common.

#### *Common sense and world knowledge*

Even though we have achieved good performance on benchmark NLP tasks using ML and DL models, language remains a bigger enigma to scientists. Beyond syntax and semantics, language encompasses knowledge of the world around us. Language for communication relies on logical reasoning and common sense regarding events from the world. For example, “I like pizza” implies “I feel happy when I eat pizza.” A more complex reasoning example would be, “If John walks out of the bedroom and goes to the garden, then John is not in the bedroom anymore, and his current location is the garden.” This might seem trivial to us humans, but it requires multistep reasoning for a machine to identify events and understand their consequences. Since this world knowledge and common sense are inherent in language, understanding them is crucial for any DL model to perform well on various language tasks. Current DL models may perform well on standard benchmarks but are still not capable of common sense understanding and logical reasoning. There are some efforts to collect common sense events and logical rules (such as if-then reasoning), but they are not well integrated yet with ML or DL models.

#### *Cost*

Building DL-based solutions for NLP tasks can be pretty expensive. The cost, in terms of both money and time, stems from multiple sources. DL models are known to be data guzzlers. Collecting a large dataset and getting it labeled can be very expensive. Owing to the size of DL models, training them to achieve desired performance can not only increase your development cycles but also result in a heavy bill for the specialized hardware (GPUs). Further, deploying and maintaining DL models can be expensive in terms of both hardware requirements and effort. Last but not least, because they’re bulky, these models may cause latency issues during inference time and may not be useful in cases where low latency is a must. To this list, one can also add technical debt arising from building and maintaining a heavy model. Loosely speaking, technical debt is the cost of rework that arises from prioritizing speedy delivery over good design and implementation choices.

#### *On-device deployment*

For many use cases, the NLP solution needs to be deployed on an embedded device rather than in the cloud—for example, a machine-translation system that helps tourists speak the translated text even without the internet. In such cases, owing to limitations of the device, the solution must work with limited memory

and power. Most DL solutions do not fit such constraints. There are some efforts in this direction [36, 37, 38] where one can deploy DL models on edge devices, but we're still quite far from generic solutions.

In most industry projects, one or more of the points mentioned above plays out. This leads to longer project cycles and higher costs (hardware, manpower), and yet the performance is either comparable or sometimes even lower than ML models. This results in a poor return on investment and often causes the NLP project to fail.

Based on this discussion, it may be apparent that DL is not always the go-to solution for all industrial NLP applications. So, this book starts with fundamental aspects of various NLP tasks and how we can solve them using techniques ranging from rule-based systems to DL models. We emphasize the data requirements and model-building pipeline, not just the technical details of individual models. Given the rapid advances in this area, we anticipate that newer DL models will come in the future to advance the state of the art but that the fundamentals of NLP tasks will not change substantially. This is why we'll discuss the basics of NLP and build on them to develop models of increasing complexity wherever possible, rather than directly jumping to the cutting edge.

Echoing Professor Zachary Lipton from Carnegie Mellon University and Professor Jacob Steinhardt from UC Berkeley [39], we also want to provide a word of caution about consuming a lot of scientific articles, research papers, and blogs on ML and NLP without context and proper training. Following a large volume of cutting-edge work may cause confusion and not-so-precise understanding. Many recent DL models are not interpretable enough to indicate the sources of empirical gains. Lipton and Steinhardt also recognize the possible conflation of technical terms and misuse of language in ML-related scientific articles, which often fail to provide any clear path to solving the problem at hand. Therefore, in this book, we carefully describe various technical concepts in the application of ML in NLP tasks via examples, code, and tips throughout the chapters.

So far, we've covered some foundational concepts related to language, NLP, ML, and DL. Before we wrap up **Chapter 1**, let's look at a case study to help get a better understanding of the various components of an NLP application.

## An NLP Walkthrough: Conversational Agents

Voice-based conversational agents like Amazon Alexa and Apple Siri are some of the most ubiquitous applications of NLP, and they're the ones most people are already familiar with. **Figure 1-19** shows the typical interaction model of a conversational agent.

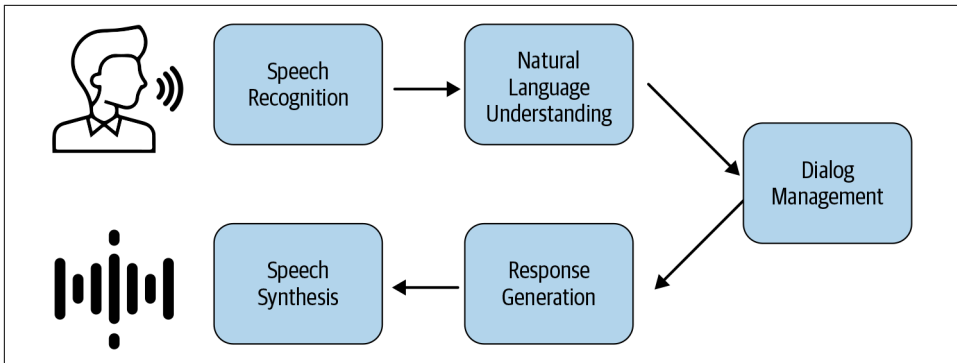


Figure 1-19. Flow of conversation agents

Here, we'll walk through all the major NLP components used in this flow:

1. *Speech recognition and synthesis*: These are the main components of any voice-based conversational agent. Speech recognition involves converting speech signals to their phonemes, which are then transcribed as words. Speech synthesis achieves the reverse process by transforming textual results into spoken language to the user. Both of these techniques have advanced considerably in the last decade, and we recommend using cloud APIs for most standard cases.
2. *Natural language understanding*: This is the next component in the conversational agent pipeline, where the user response received (transcribed as text) is analyzed using a natural language understanding system. This can be broken into many small NLP subtasks, such as:
  - *Sentiment analysis*: Here, we analyze the sentiment of the user response. This will be covered in [Chapter 4](#).
  - *Named entity recognition*: Here, we identify all the important entities the user mentioned in their response. This will be covered in [Chapter 5](#).
  - *Coreference resolution*: Here, we find out the references of the extracted entities from the conversation history. For example, a user may say “*Avengers Endgame* was awesome” and later refer back to the movie, saying “The movie’s special effects were great.” In this case, we would want to link that “movie” is referring to *Avengers Endgame*. This is covered briefly in [Chapter 5](#).
3. *Dialog management*: Once we’ve extracted the useful information from the user’s response, we may want to understand the user’s intent—i.e., if they’re asking a factual question like “What is the weather today?” or giving a command like “Play Mozart songs.” We can use a text-classification system to classify the user response as one of the pre-defined intents. This helps the conversational agent know what’s being asked. Intent classification will be covered in [Chapters 4 and 6](#). During this process, the system may ask a few clarifying questions to elicit fur-

ther information from the user. Once we've figured out the user's intent, we want to figure out which suitable action the conversational agent should take to fulfill the user's request. This is done based on the information and intent extracted from the user's response. Examples of suitable actions could be generating an answer from the internet, playing music, dimming lights, or asking a clarifying question. We'll cover this in [Chapter 6](#).

4. *Response generation*: Finally, the conversational agent generates a suitable action to perform based on a semantic interpretation of the user's intent and additional inputs from the dialogue with the user. As mentioned earlier, the agent can retrieve information from the knowledge base and generate responses using a pre-defined template. For example, it might respond by saying, "Now playing Symphony No. 25" or "The lights have been dimmed." In certain scenarios, it can also generate a completely new response.

We hope this brief case study provided an overview of how different NLP components we'll be discussing throughout this book will come together to build one application: a conversational agent. We'll see more details about these components as we progress through the book, and we'll discuss conversational agents specifically in [Chapter 6](#).

## Wrapping Up

From the broader contours of what a language is to a concrete case study of a real-world NLP application, we've covered a range of NLP topics in this chapter. We also discussed how NLP is applied in the real world, some of its challenges and different tasks, and the role of ML and DL in NLP. This chapter was meant to give you a baseline of knowledge that we'll build on throughout the book. The next two chapters (Chapters [2](#) and [3](#)) will introduce you to some of the foundational steps necessary for building NLP applications. Chapters [4–7](#) focus on core NLP tasks along with industrial use cases that can be solved with them. In Chapters [8–10](#), we discuss how NLP is used across different industry verticals such as e-commerce, healthcare, finance, etc. [Chapter 11](#) brings everything together and discusses what it takes to build end-to-end NLP applications in terms of design, development, testing, and deployment. With this broad overview in place, let's start delving deeper into the world of NLP.

## References

- [1] [Arria.com](#). "NLG for Your Industry". Last accessed June 15, 2020.
- [2] UCL. [Phonetic symbols for English](#). Last accessed June 15, 2020.

---

# NLP Pipeline

*The whole is more than the sum of its parts. It is more correct to say that the whole is something else than the sum of its parts, because summing up is a meaningless procedure, whereas the whole-part relationship is meaningful.*  
—Kurt Koffka

In the previous chapter, we saw examples of some common NLP applications that we might encounter in everyday life. If we were asked to build such an application, think about how we would approach doing so at our organization. We would normally walk through the requirements and break the problem down into several sub-problems, then try to develop a step-by-step procedure to solve them. Since language processing is involved, we would also list all the forms of text processing needed at each step. This step-by-step processing of text is known as a *pipeline*. It is the series of steps involved in building any NLP model. These steps are common in every NLP project, so it makes sense to study them in this chapter. Understanding some common procedures in any NLP pipeline will enable us to get started on any NLP problem encountered in the workplace. Laying out and developing a text-processing pipeline is seen as a starting point for any NLP application development process. In this chapter, we will learn about the various steps involved and how they play important roles in solving the NLP problem and we'll see a few guidelines about when and how to use which step. In later chapters, we'll discuss specific pipelines for various NLP tasks (e.g., Chapters 4–7).

Figure 2-1 shows the main components of a generic pipeline for modern-day, data-driven NLP system development. The key stages in the pipeline are as follows:

1. Data acquisition
2. Text cleaning
3. Pre-processing



4. Feature engineering
5. Modeling
6. Evaluation
7. Deployment
8. Monitoring and model updating

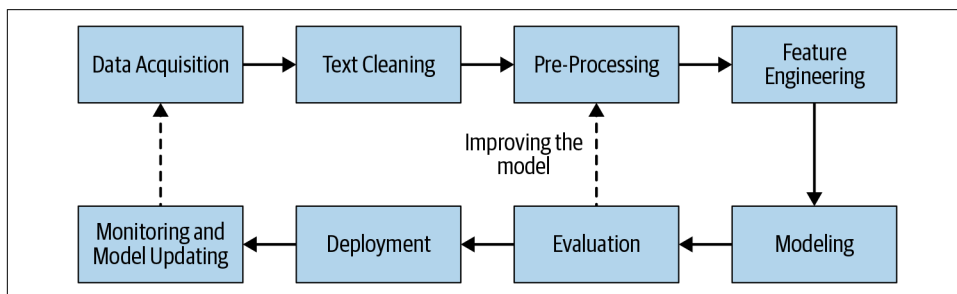


Figure 2-1. Generic NLP pipeline

The first step in the process of developing any NLP system is to collect data relevant to the given task. Even if we're building a rule-based system, we still need some data to design and test our rules. The data we get is seldom clean, and this is where text cleaning comes into play. After cleaning, text data often has a lot of variations and needs to be converted into a canonical form. This is done in the pre-processing step. This is followed by feature engineering, where we carve out indicators that are most suitable for the task at hand. These indicators are converted into a format that is understandable by modeling algorithms. Then comes the modeling and evaluation phase, where we build one or more models and compare and contrast them using a relevant evaluation metric(s). Once the best model among the ones evaluated is chosen, we move toward deploying this model in production. Finally, we regularly monitor the performance of the model and, if need be, update it to keep up its performance.

Note that, in the real world, the process may not always be linear as it's shown in the pipeline in [Figure 2-1](#); it often involves going back and forth between individual steps (e.g., between feature extraction and modeling, modeling and evaluation, and so on). Also, there are loops in between, most commonly going from evaluation to pre-processing, feature engineering, modeling, and back to evaluation. There is also an overall loop that goes from monitoring to data acquisition, but this loop happens at the project level.

Note that exact step-by-step procedures may depend on the specific task at hand. For example, a text-classification system may require a different feature extraction step compared to a text-summarization system. We will focus on application-specific

pipeline stages in subsequent chapters in the book. Also, depending on the phase of the project, different steps can take different amounts of time. In the initial phases, most of the time is used in modeling and evaluation, whereas once the system matures, feature engineering can take far more time.

For the rest of this chapter, we'll look at the individual stages of the pipeline in detail along with examples. We'll describe some of the most common procedures at each stage and discuss some use cases to illustrate them. Let's start with the first step: data acquisition.

## Data Acquisition

Data is the heart of any ML system. In most industrial projects, it is often the data that becomes the bottleneck. In this section, we'll discuss various strategies for gathering relevant data for an NLP project.

Let's say we're asked to develop an NLP system to identify whether an incoming customer query (for example, using a chat interface) is a sales inquiry or a customer care inquiry. Depending on the type of query, it should be automatically routed to the right team. How can one go about building such a system? Well, the answer depends on the type and amount of data we have to work with.

In an ideal setting, we'll have the required datasets with thousands—maybe even millions—of data points. In such cases, we don't have to worry about data acquisition. For example, in the scenario we just described, we have historic queries from previous years, which sales and support teams responded to. Further, the teams tagged these queries as sales, support, or other. So, not only do we have the data, but we also have the labels. However, in many AI projects, one is not so lucky. Let's look at what we can do in a less-than-ideal scenario.

If we have little or no data, we can start by looking at patterns in the data that indicate if the incoming message is a sales or support query. We can then use regular expressions and other heuristics to match these patterns to separate sales queries from support queries. We evaluate this solution by collecting a set of queries from both categories and calculating what percentage of the messages were correctly identified by our system. Say we get OK-ish numbers. We would like to improve the system performance.

Now we can start thinking about using NLP techniques. For this, we need labeled data, a collection of queries where each one is labeled with sales or support. How can we get such data?

### *Use a public dataset*

We could see if there are any public datasets available that we can leverage. Take a look at the compilation by Nicolas Iderhoff [1] or search Google’s specialized search engine for datasets [2]. If you find a suitable dataset that’s similar to the task at hand, great! Build a model and evaluate. If not, then what?

### *Scrape data*

We could find a source of relevant data on the internet—for example, a consumer or discussion forum where people have posted queries (sales or support). Scrape the data from there and get it labeled by human annotators.

For many industrial settings, gathering data from external sources does not suffice because the data doesn’t contain nuances like product names or product-specific user behavior and thus might be very different from the data seen in production environments. This is when we’ll have to start looking for data inside the organization.

### *Product intervention*

In most industrial settings, AI models seldom exist by themselves. They’re developed mostly to serve users via a feature or product. In all such cases, the AI team should work with the product team to collect more and richer data by developing better instrumentation in the product. In the tech world, this is called *product intervention*.

Product intervention is often the best way to collect data for building intelligent applications in industrial settings. Tech giants like Google, Facebook, Microsoft, Netflix, etc., have known this for a long time and have tried to collect as much data as possible from as many users as possible.

### *Data augmentation*

While instrumenting products is a great way to collect data, it takes time. Even if you instrument the product today, it can take anywhere between three to six months to collect a decent-sized, comprehensive dataset. So, can we do something in the meantime?

NLP has a bunch of techniques through which we can take a small dataset and use some tricks to create more data. These tricks are also called *data augmentation*, and they try to exploit language properties to create text that is syntactically similar to source text data. They may appear as hacks, but they work very well in practice. Let’s look at some of them:

### *Synonym replacement*

Randomly choose “k” words in a sentence that are not stop words. Replace these words with their synonyms. For synonyms, we can use Synsets in Wordnet [3, 4].

### Back translation

Say we have a sentence, S1, in English. We use a machine-translation library like Google Translate to translate it into some other language—say, German. Let the corresponding sentence in German be S2. Now, we'll use the machine-translation library again to translate back to English. Let the output sentence be S3.

We'll find that S1 and S3 are very similar in meaning but are slight variations of each other. Now we can add S3 to our dataset. This trick works beautifully for text classification. **Figure 2-2** [5] shows an example of back translation in action.

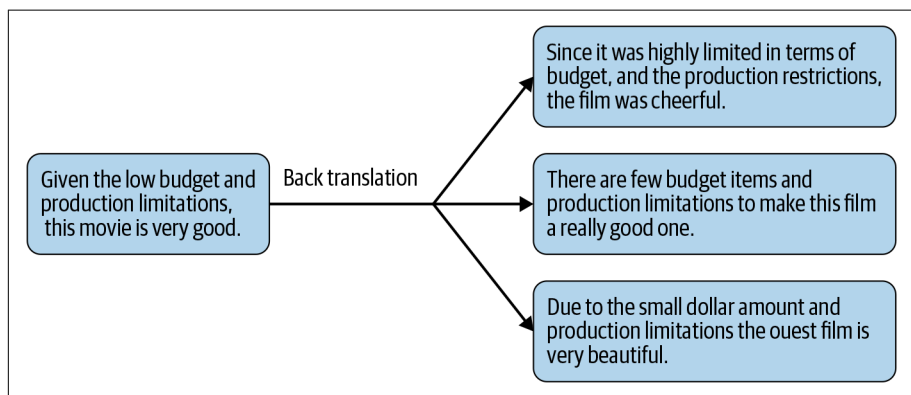


Figure 2-2. Back translation

### TF-IDF-based word replacement

Back translation can lose certain words that are crucial to the sentence. In [5], the authors use **TF-IDF**, a concept we'll introduce in **Chapter 3**, to handle this.

### Bigram flipping

Divide the sentence into bigrams. Take one bigram at random and flip it. For example: "I am going to the supermarket." Here, we take the bigram "going to" and replace it with the flipped one: "to going."

### Replacing entities

Replace entities like person name, location, organization, etc., with other entities in the same category. That is, replace person name with another person name, city with another city, etc. For example, in "I live in California," replace "California" with "London."

### Adding noise to data

In many NLP applications, the incoming data contains spelling mistakes. This is primarily due to characteristics of the platform where the data is being generated (for example, Twitter). In such cases, we can add a bit of noise to data to train robust models. For example, randomly choose a word in a sentence and replace it with another word that's closer in spelling to the first word. Another source of

noise is the “fat finger” problem [6] on mobile keyboards. Simulate a QWERTY keyboard error by replacing a few characters with their neighboring characters on the QWERTY keyboard.

#### *Advanced techniques*

There are other advanced techniques and systems that can augment text data. Some of the notable ones are:

##### *Snorkel [7, 8, 52]*

This is a system for building training data automatically, without manual labeling. Using Snorkel, a large training dataset can be “created”—without manual labeling—using heuristics and creating synthetic data by transforming existing data and creating new data samples. This approach was shown to work well at Google in the recent past [9].

##### *Easy Data Augmentation (EDA) [10, 11] and NLPAug [12]*

These two libraries are used to create synthetic samples for NLP. They provide implementation of various data augmentation techniques, including some techniques that we discussed previously.

##### *Active learning [13]*

This is a specialized paradigm of ML where the learning algorithm can interactively query a data point and get its label. It is used in scenarios where there is an abundance of unlabeled data but manually labeling is expensive. In such cases, the question becomes: for which data points should we ask for labels to maximize learning while keeping the labeling cost low?

In order for most of the techniques we discussed in this section to work well, one key requirement is a clean dataset to start with, even if it’s not very big. In our experience, data augmentation techniques can work really well. Further, in day-to-day ML practice, datasets come from heterogeneous sources. A combination of public datasets, labeled datasets, and augmented datasets are used for building early-stage production models, as we often may not have large datasets for our custom scenarios to start with. Once we have the data we want for a given task, we proceed to the next step: text cleaning.

## Text Extraction and Cleanup

Text extraction and cleanup refers to the process of extracting raw text from the input data by removing all the other non-textual information, such as markup, metadata, etc., and converting the text to the required encoding format. Typically, this depends on the format of available data in the organization (e.g., static data from PDF, HTML or text, some form of continuous data stream, etc.), as shown in [Figure 2-3](#).

Text extraction is a standard data-wrangling step, and we don't usually employ any NLP-specific techniques during this process. However, in our experience, it is an important step that has implications for all other aspects of the NLP pipeline. Further, it can also be the most time-consuming part of a project. While the design of text-extraction tools is beyond the scope of this book, we'll look at a few examples to illustrate different issues involved in this step in this section. We'll also touch on some of the important aspects of text extraction from various sources as well as cleanup to make them consumable in downstream pipelines.



Figure 2-3. (a) PDF invoice, [14] (b) HTML texts, and (c) text embedded in an image [15]

## HTML Parsing and Cleanup

Say we're working on a project where we're building a forum search engine for programming questions. We've identified Stack Overflow as a source and decided to extract question and best-answer pairs from the website. How can we go through the text-extraction step in this case? If we observe the HTML markup of a typical Stack Overflow question page, we notice that questions and answers have special tags associated with them. We can utilize this information while extracting text from the HTML page. While it may seem like writing our own HTML parser is the way to go, for most cases we encounter, it's more feasible to utilize existing libraries such as BeautifulSoup [16] and Scrapy [17], which provide a range of utilities to parse web pages. The following code snippet shows how to use BeautifulSoup to address the problem described here, extracting a question and its best-answer pair from a Stack Overflow web page:

```
from bs4 import BeautifulSoup
from urllib.request import urlopen
myurl = "https://stackoverflow.com/questions/415511/ \
how-to-get-the-current-time-in-python"
html = urlopen(myurl).read()
soupified = BeautifulSoup(html, "html.parser")
question = soupified.find("div", {"class": "question"})
questiontext = question.find("div", {"class": "post-text"})
print("Question: \n", questiontext.get_text().strip())
answer = soupified.find("div", {"class": "answer"})
answertext = answer.find("div", {"class": "post-text"})
print("Best answer: \n", answertext.get_text().strip())
```

Here, we're relying on our knowledge of the structure of an HTML document to extract what we want from it. This code shows the output as follows:

```
Question:
What is the module/method used to get the current time?
Best answer:
Use:
>>> import datetime
>>> datetime.datetime.now()
datetime.datetime(2009, 1, 6, 15, 8, 24, 78915)

>>> print(datetime.datetime.now())
2009-01-06 15:08:24.789150

And just the time:
>>> datetime.datetime.now().time()
datetime.time(15, 8, 24, 78915)

>>> print(datetime.datetime.now().time())
15:08:24.789150
```

See the documentation for more information.

To save typing, you can import the datetime object from the datetime module:

```
>>> from datetime import datetime
```

Then remove the leading datetime. from all of the above.

In this example, we had a specific need: extracting a question and its answer. In some scenarios—for example, extracting postal addresses from web pages—we would get all the text (instead of only parts of it) from the web page first, before doing anything else. Typically, all HTML libraries have some function that can strip off all HTML tags and return only the content between the tags. But this often results in noisy output, and you may end up seeing a lot of JavaScript in the extracted content as well. In such cases, we should look to extract content between only those tags that typically contain text in web pages.

## Unicode Normalization

As we develop code for cleaning up HTML tags, we may also encounter various Unicode characters, including symbols, emojis, and other graphic characters. A handful of Unicode characters are shown in [Figure 2-4](#).

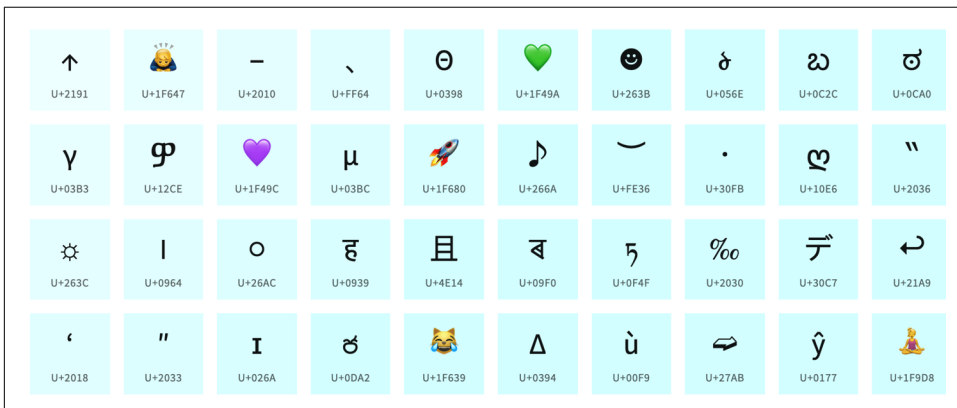


Figure 2-4. Unicode characters [18]

To parse such non-textual symbols and special characters, we use Unicode normalization. This means that the text we see should be converted into some form of binary representation to store in a computer. This process is known as *text encoding*. Ignoring encoding issues can result in processing errors further in the pipeline.

There are several encoding schemes, and the default encoding can be different for different operating systems. Sometimes (more commonly than you think), especially when dealing with text in multiple languages, social media data, etc., we may have to convert between these encoding schemes during the text-extraction process. Refer to [19] for an introduction to how language is represented on computers and what difference an encoding scheme makes. Here is an example of Unicode handling:



```
text = 'I love 🍕! Shall we book a 🚗 to gizza?'
Text = text.encode("utf-8")
print(Text)
```

which outputs:

```
b'I love Pizza \xf0\x9f\x8d\x95! Shall we book a cab \xf0\x9f\x9a\x95
to get pizza?'
```

This processed text is machine readable and can be used in downstream pipelines. We address issues regarding handling Unicode characters with this same example in more detail in [Chapter 8](#).

## Spelling Correction

In the world of fast typing and fat-finger typing [6], incoming text data often has spelling errors. This can be prevalent in search engines, text-based chatbots deployed on mobile devices, social media, and many other sources. While we remove HTML tags and handle Unicode characters, this remains a unique problem that may hurt the linguistic understanding of the data, and shorthand text messages in social micro-blogs often hinder language processing and context understanding. Two such examples follow:

**Shorthand typing:** Hllo world! I am back!

**Fat finger problem [20]:** I promise that I will not bresk the silence again!

While shorthand typing is prevalent in chat interfaces, fat-finger problems are common in search engines and are mostly unintentional. Despite our understanding of the problem, we don't have a robust method to fix this, but we still can make good attempts to mitigate the issue. Microsoft released a REST API [21] that can be used in Python for potential spell checking:

```
import requests
import json

api_key = "<ENTER-KEY-HERE>"
example_text = "Hollo, wrld" # the text to be spell-checked

data = {'text': example_text}
params = {
    'mkt': 'en-us',
    'mode': 'proof'
}
headers = {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Ocp-Apim-Subscription-Key': api_key,
}

response = requests.post(endpoint, headers=headers, params=params, data=data)
json_response = response.json()
print(json.dumps(json_response, indent=4))
```

Output (partially shown here):

```
"suggestions": [  
  {  
    "suggestion": "Hello",  
    "score": 0.9115257530801  
  },  
  {  
    "suggestion": "Hollow",  
    "score": 0.858039839213461  
  },  
  {  
    "suggestion": "Hallo",  
    "score": 0.597385084464481  
  }  
]
```

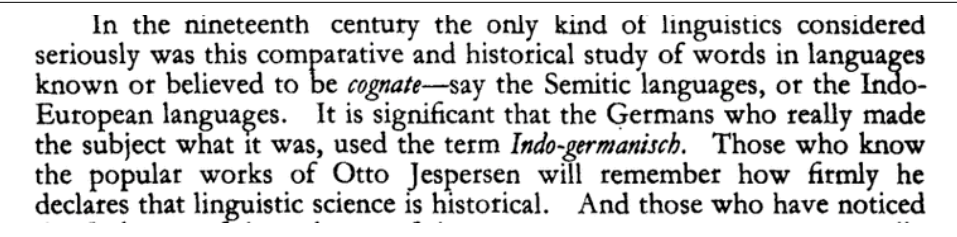
You can see the full tutorial in [21].

Going beyond APIs, we can build our own spell checker using a huge dictionary of words from a specific language. A naive solution would be to look for all words that can be composed with minimal alteration (addition, deletion, substitution) to its constituent letters. For example, if “Hello” is a valid word that is already present in the dictionary, then the addition of “o” (minimal) to “Hllo” would make the correction.

## System-Specific Error Correction

HTML or raw text scraped from the web are just a couple of sources for textual data. Consider another scenario where our dataset is in the form of a collection of PDF documents. The pipeline in this case starts with extraction of plain text from PDF documents. However, different PDF documents are encoded differently, and sometimes, we may not be able to extract the full text, or the structure of the text may get messed up. If we need full text or our text has to be grammatical or in full sentences (e.g., when we want to extract relations between various people in the news based on newspaper text), this can impact our application. While there are several libraries, such as PyPDF [22], PDFMiner [23], etc., to extract text from PDF documents, they are far from perfect, and it’s not uncommon to encounter PDF documents that can’t be processed by such libraries. We leave their exploration as an exercise for the reader. [24] discusses some of the issues involved in PDF-to-text extraction in detail.

Another common source of textual data is scanned documents. Text extraction from scanned documents is typically done through optical character recognition (OCR), using libraries such as Tesseract [25, 26]. Consider the example image—a snippet from a 1950 article in a journal [27]—shown in [Figure 2-5](#).



In the nineteenth century the only kind of linguistics considered seriously was this comparative and historical study of words in languages known or believed to be *cognate*—say the Semitic languages, or the Indo-European languages. It is significant that the Germans who really made the subject what it was, used the term *Indo-germanisch*. Those who know the popular works of Otto Jespersen will remember how firmly he declares that linguistic science is historical. And those who have noticed

Figure 2-5. An example of scanned text

The code snippet below shows how the Python library `pytesseract` can be used to extract text from this image:

```
from PIL import Image
from pytesseract import image_to_string
filename = "somefile.png"
text = image_to_string(Image.open(filename))
print(text)
```

This code will print the output as follows, where “\n” indicates a newline character:

```
'in the nineteenth century the only Kind of linguistics considered\nseriously
was this comparative and historical study of words in languages\nknown or
believed to Fe cognate—say the Semitic languages, or the Indo-\nEuropean
languages. It is significant that the Germans who really made\nthe subject what
it was, used the term Indo-germanisch. Those who know\nthe popular works of
Otto Jespersen will remember how fitmly he\ndeclares that linguistic
science is historical. And those who have noticed'
```

We notice that there are two errors in the output of the OCR system in this case. Depending on the quality of the original scan, OCR output can potentially have larger amounts of errors. How do we clean up this text before feeding it into the next stage of the pipeline? One approach is to run the text through a spell checker such as `pyenchant` [28], which will identify misspellings and suggest some alternatives. More recent approaches use neural network architectures to train word/character-based language models, which are in turn used for correcting OCR text output based on the context [29].

Recall that we saw an example of a voice-based assistant in [Chapter 1](#). In such cases, the source of text extraction is the output of an automatic speech recognition (ASR) system. Like OCR, it's common to see some errors in ASR, owing to various factors, such as dialectical variations, slang, non-native English, new or domain-specific vocabulary, etc. The above-mentioned approach of spell checkers or neural language models can be followed here as well to clean up the extracted text.

What we've seen so far are just some examples of potential issues that may come up during the text-extraction and cleaning process. Though NLP plays a very small role in this process, we hope these examples illustrate how text extraction and cleanup

could pose challenges in a typical NLP pipeline. We'll also touch on these aspects in upcoming chapters for different NLP applications, where relevant. Let's move on to the next step in our pipeline: pre-processing.

## Pre-Processing

Let's start with a simple question: we already did some cleanup in the previous step; why do we still have to pre-process text? Consider a scenario where we're processing text from Wikipedia pages about individuals to extract biographical information about them. Our data acquisition starts with crawling such pages. However, our crawled data is all in HTML, with a lot of boilerplate from Wikipedia (e.g., all the links in the left panel), possibly the presence of links to multiple languages (in their script), etc. All such information is irrelevant for extracting features from text (in most cases). Our text-extraction step removed all this and gave us the plain text of the article we need. However, all NLP software typically works at the sentence level and expects a separation of words at the minimum. So, we need some way to split a text into words and sentences before proceeding further in a processing pipeline. Sometimes, we need to remove special characters and digits, and sometimes, we don't care whether a word is in upper or lowercase and want everything in lowercase. Many more decisions like this are made while processing text. Such decisions are addressed during the pre-processing step of the NLP pipeline. Here are some common pre-processing steps used in NLP software:

### *Preliminaries*

Sentence segmentation and word tokenization.

### *Frequent steps*

Stop word removal, stemming and lemmatization, removing digits/punctuation, lowercasing, etc.

### *Other steps*

Normalization, language detection, code mixing, transliteration, etc.

### *Advanced processing*

POS tagging, parsing, coreference resolution, etc.

While not all steps will be followed in all the NLP pipelines we encounter, the first two are more or less seen everywhere. Let's take a look at what each of these steps mean.

## Preliminaries

As mentioned earlier, NLP software typically analyzes text by breaking it up into words (tokens) and sentences. Hence, any NLP pipeline has to start with a reliable system to split the text into sentences (sentence segmentation) and further split a sentence into words (word tokenization). On the surface, these seem like simple tasks, and you may wonder why they need special treatment. We will see why in the coming two subsections.

### Sentence segmentation

As a simple rule, we can do sentence segmentation by breaking up text into sentences at the appearance of full stops and question marks. However, there may be abbreviations, forms of addresses (Dr., Mr., etc.), or ellipses (...) that may break the simple rule.

Thankfully, we don't have to worry about how to solve these issues, as most NLP libraries come with some form of sentence and word splitting implemented. A commonly used library is Natural Language Tool Kit (NLTK) [30]. The code example below shows how to use a sentence and word splitter from NLTK and uses the first paragraph of this chapter as input:

```
from nltk.tokenize import sent_tokenize, word_tokenize

mytext = "In the previous chapter, we saw examples of some common NLP applications that we might encounter in everyday life. If we were asked to build such an application, think about how we would approach doing so at our organization. We would normally walk through the requirements and break the problem down into several sub-problems, then try to develop a step-by-step procedure to solve them. Since language processing is involved, we would also list all the forms of text processing needed at each step. This step-by-step processing of text is known as pipeline. It is the series of steps involved in building any NLP model. These steps are common in every NLP project, so it makes sense to study them in this chapter. Understanding some common procedures in any NLP pipeline will enable us to get started on any NLP problem encountered in the workplace. Laying out and developing a text-processing pipeline is seen as a starting point for any NLP application development process. In this chapter, we will learn about the various steps involved and how they play important roles in solving the NLP problem and we'll see a few guidelines about when and how to use which step. In later chapters, we'll discuss specific pipelines for various NLP tasks (e.g., Chapters 4-7)."
```

```
my_sentences = sent_tokenize(mytext)
```

## Word tokenization

Similar to sentence tokenization, to tokenize a sentence into words, we can start with a simple rule to split text into words based on the presence of punctuation marks. The NLTK library allows us to do that. If we take the previous example:

```
for sentence in my_sentences:
    print(sentence)
    print(word_tokenize(sentence))
```

For the first sentence, the output is printed as follows:

```
In the previous chapter, 'we', 'saw', 'a', 'quick',
overview', 'of', 'what', 'is', 'NLP', 'what', 'are', 'some', 'of', 'the',
common', 'applications', 'and', 'challenges', 'in', 'NLP', 'and', 'an',
introduction', 'to', 'different', 'tasks', 'in', 'NLP', '.']
```

While readily available solutions work for most of our needs and most NLP libraries will have a tokenizer and sentence splitter bundled with them, it's important to remember that they're far from perfect. For example, consider this sentence: "Mr. Jack O'Neil works at Melitas Marg, located at 245 Yonge Avenue, Austin, 70272." If we run this through the NLTK tokenizer, O, ', and Neil are identified as three separate tokens. Similarly, if we run the sentence: "There are \$10,000 and €1000 which are there just for testing a tokenizer" through this tokenizer, while \$ and 10,000 are identified as separate tokens, €1000 is identified as a single token. In another scenario, if we want to tokenize tweets, this tokenizer will separate a hashtag into two tokens: a "#" sign and the string that follows it. In such cases, we may need to use a custom tokenizer built for our purpose. To complete our example, we'll perform word tokenization after we perform sentence tokenization.

A point to note in this context is that NLTK also has a tweet tokenizer; we'll see how it's useful in Chapters 4 and 8. To summarize, although word- and sentence-tokenization approaches appear to be elementary and easy to implement, they may not always meet our specific tokenization needs, as we saw in the above examples. Note that we refer to NLTK's example, but these observations hold true for any other library as well. We leave that exploration as an exercise for the reader.

As tokenization may differ from one domain to the other, tokenization is also heavily dependent on language. Each language can have various linguistic rules and exceptions. **Figure 2-6** shows an example where "N.Y.!" has a total of three punctuations. But in English, N.Y. stands for New York, hence "N.Y." should be treated as a single word and not be tokenized further. Such language-specific exceptions can be specified in the tokenizer provided by spaCy [31]. It's also possible in spaCy to develop custom rules to handle such exceptions for languages that have high inflections (prefixes or suffixes) and complex morphology.

Another important fact to keep in mind is that any sentence segmenter and tokenizer will be sensitive to the input they receive. Let's say we're writing software to extract some information, such as company, position, and salary, from job offer letters. They follow a certain format, with a To and a From address, a signed note at the end, and so on. How will we decide what a sentence is in such a case? Should the entire address be considered a single "sentence"? Or should each line be split separately? Answers to such questions depend on what you want to extract and how sensitive the rest of the pipeline is about such decisions. For identifying specific patterns (e.g., dates or money expressions), well-formed regular expressions are the first step. In many practical scenarios, we may end up using a custom tokenizer or sentence segmenter that suits our text structure instead of or on top of an existing one available in a standard NLP library [32].

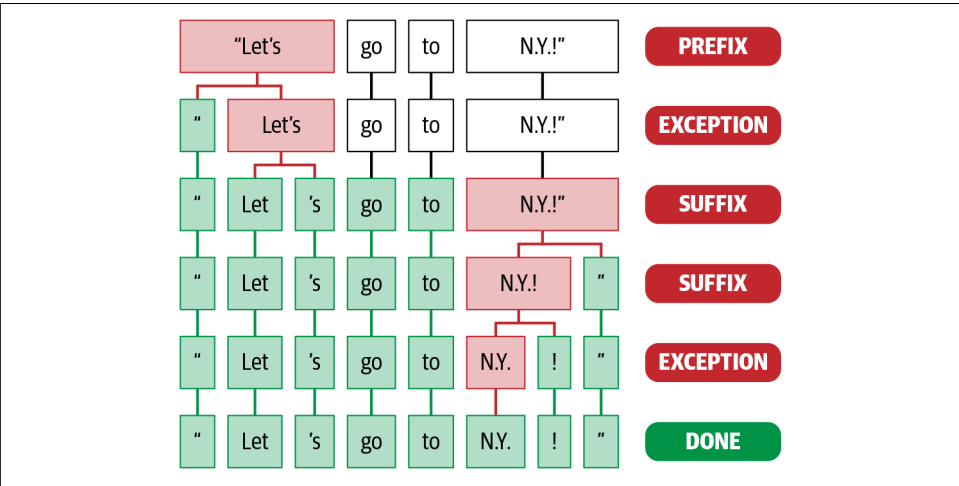


Figure 2-6. Language-specific (English here) exceptions during tokenization [31]

## Frequent Steps

Let's look at some other frequently performed pre-processing operations in an NLP pipeline. Say we're designing software that identifies the category of a news article as one of politics, sports, business, and other. Assume we have a good sentence segmenter and word tokenizer in place. At that point, we would have to start thinking about what kind of information is useful for developing a categorization tool. Some of the frequently used words in English, such as a, an, the, of, in, etc., are not particularly useful for this task, as they don't carry any content on their own to separate between the four categories. Such words are called *stop words* and are typically (though not always) removed from further analysis in such problem scenarios. There is no standard list of stop words for English, though. There are some popular lists (NLTK has one, for example), although what a stop word is can vary depending on

what we're working on. For example, the word “news” is perhaps a stop word for this problem scenario, but it may not be a stop word for the offer letter data in the example mentioned in the previous step.

Similarly, in some cases, upper or lowercase may not make a difference for the problem. So, all text is lowercased (or uppercased, although lowercasing is more common). Removing punctuation and/or numbers is also a common step for many NLP problems, such as text classification (Chapter 4), information retrieval (Chapter 7), and social media analytics (Chapter 8). We'll see examples of how and if these steps are useful in upcoming chapters.

The code example below shows how to remove stop words, digits, and punctuation and lowercase a given collection of texts:

```
from nltk.corpus import stopwords
from string import punctuation
def preprocess_corpus(texts):
    mystopwords = set(stopwords.words("english"))
    def remove_stops_digits(tokens):
        return [token.lower() for token in tokens if token not in mystopwords
                not token.isdigit() and token not in punctuation]
    return [remove_stops_digits(word_tokenize(text)) for text in texts]
```

It's important to note that these four processes are neither mandatory nor sequential in nature. The above function is just an illustration of how to add those processing steps into our project. The pre-processing we saw here, while specific to textual data, has nothing particularly linguistic about it—we're not looking at any aspect of language other than frequency (stop words are very frequent words), and we're removing non-alphabetic data (punctuation, digits). Two commonly used pre-processing steps that take the word-level properties into account are stemming and lemmatization.

## Stemming and lemmatization

Stemming refers to the process of removing suffixes and reducing a word to some base form such that all different variants of that word can be represented by the same form (e.g., “car” and “cars” are both reduced to “car”). This is accomplished by applying a fixed set of rules (e.g., if the word ends in “-es,” remove “-es”). More such examples are shown in Figure 2-7. Although such rules may not always end up in a linguistically correct base form, stemming is commonly used in search engines to match user queries to relevant documents and in text classification to reduce the feature space to train machine learning models.

The following code snippet shows how to use a popular stemming algorithm called Porter Stemmer [33] using NLTK:



```
from nltk.stem.porter import PorterStemmer
stemmer = PorterStemmer()
word1, word2 = "cars", "revolution"
print(stemmer.stem(word1), stemmer.stem(word2))
```

This gives “car” as the stemmed version for “cars,” but “revolut” as the stemmed form of “revolution,” even though the latter is not linguistically correct. While this may not affect the performance of a search engine, derivation of correct linguistic form becomes useful in some other scenarios. This is accomplished by another process, closer to stemming, called lemmatization.

Lemmatization is the process of mapping all the different forms of a word to its base word, or *lemma*. While this seems close to the definition of stemming, they are, in fact, different. For example, the adjective “better,” when stemmed, remains the same. However, upon lemmatization, this should become “good,” as shown in [Figure 2-7](#). Lemmatization requires more linguistic knowledge, and modeling and developing efficient lemmatizers remains an open problem in NLP research even now.

Stemming	Lemmatization
adjustable -> adjust	was -> (to) be
formality -> formaliti	better -> good
formaliti -> formal	meeting -> meeting
airliner -> airlin	

Figure 2-7. Difference between stemming and lemmatization [34]

The following code snippet shows the usage of a lemmatizer based on WordNet from NLTK:

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordnetLemmatizer()
print(lemmatizer.lemmatize("better", pos="a")) #a is for adjective
```

And this code snippet shows a lemmatizer using spaCy:

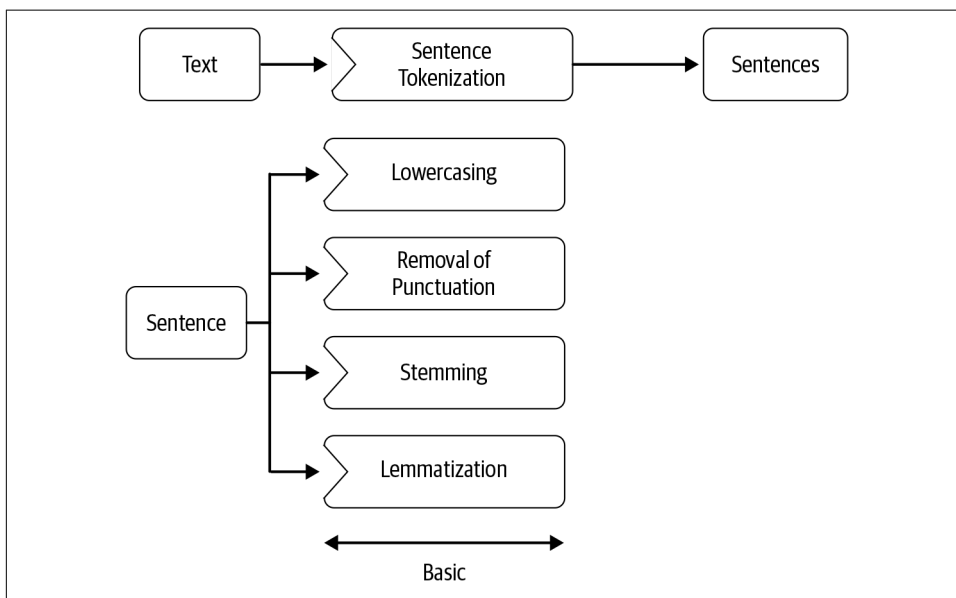
```
import spacy
sp = spacy.load('en_core_web_sm')
token = sp(u'better')
for word in token:
    print(word.text, word.lemma_)
```

NLTK prints the output as “good,” whereas spaCy prints “well”—both are correct. Since lemmatization involves some amount of linguistic analysis of the word and its context, it is expected that it will take longer to run than stemming, and it’s also typically used only if absolutely necessary. We’ll see how stemming and lemmatization are useful in the next chapters. The choice of lemmatizer is optional; we can choose NLTK or spaCy given what framework we’re using for other pre-processing steps in order to use a single framework in the complete pipeline.

Remember that not all of these steps are always necessary, and not all of them are performed in the order in which they're discussed here. For example, if we were to remove digits and punctuation, what is removed first may not matter much. However, we typically lowercase the text before stemming. We also don't remove tokens or lowercase the text before doing lemmatization because we have to know the part of speech of the word to get its lemma, and that requires all tokens in the sentence to be intact. A good practice to follow is to prepare a sequential list of pre-processing tasks to be done after having a clear understanding of how to process our data.

**Figure 2-8** lists the different pre-processing steps we've seen in this subsection so far, as a quick summary.

Note that these are the more common pre-processing steps, but they're by no means exhaustive. Depending on the nature of the data, some additional pre-processing steps may be important. Let's take a look at a few of those steps.



*Figure 2-8. Common pre-processing steps on a blob of text*

## Other Pre-Processing Steps

So far, we've seen a few common pre-processing steps in an NLP pipeline. While we haven't explicitly stated the nature of the texts, we have assumed that we're dealing with regular English text. What's different if that's not the case? Let's introduce a few more pre-processing steps to deal with such scenarios, using a few examples.

## Text normalization

Consider a scenario where we're working with a collection of social media posts to detect news events. Social media text is very different from the language we'd see in, say, newspapers. A word can be spelled in different ways, including in shortened forms, a phone number can be written in different formats (e.g., with and without hyphens), names are sometimes in lowercase, and so on. When we're working on developing NLP tools to work with such data, it's useful to reach a canonical representation of text that captures all these variations into one representation. This is known as *text normalization*. Some common steps for text normalization are to convert all text to lowercase or uppercase, convert digits to text (e.g., 9 to nine), expand abbreviations, and so on. A simple way to incorporate text normalization can be found in Spacy's source code [35], which is a dictionary showing different spellings of a preset collection of words mapped to a single spelling. We'll see more examples of text normalization in [Chapter 8](#).

## Language detection

A lot of web content is in non-English languages. For example, say we're asked to collect all reviews about our product on the web. As we navigate different e-commerce websites and start crawling pages related to our product, we notice several non-English reviews showing up. Since a majority of the pipeline is built with language-specific tools, what will happen to our NLP pipeline, which is expecting English text? In such cases, language detection is performed as the first step in an NLP pipeline. We can use libraries like Polyglot [36] for language detection. Once this step is done, the next steps could follow a language-specific pipeline.

## Code mixing and transliteration

The discussion above was about a scenario where the content is in non-English languages. However, there's another scenario where a single piece of content is in more than one language. Many people across the world speak more than one language in their day-to-day lives. Thus, it's not uncommon to see them using multiple languages in their social media posts, and a single post may contain many languages. As an example of code mixing, we can look at a Singlish (Singapore slang + English) phrase from LDC [37] in [Figure 2-9](#).



Figure 2-9. Code mixing in a single Singlish phrase

A single popular phrase has words from Tamil, English, Malay, and three Chinese language variants. Code mixing refers to this phenomenon of switching between languages. When people use multiple languages in their write-ups, they often type words from these languages in Roman script, with English spelling. So, the words of another language are written along with English text. This is known as *transliteration*. Both of these phenomena are common in multilingual communities and need to be handled during the pre-processing of text. We'll discuss more about these in [Chapter 8](#), where we'll see examples of these phenomena in social media text.

This concludes our discussion of common pre-processing steps. While this list is by no means exhaustive, we hope it gives you some idea of the different forms of pre-processing that may be required, depending on the nature of the dataset. Now, let's take a look at a few more pre-processing steps in the NLP pipeline—ones that need advanced language processing beyond what we've seen so far.

## Advanced Processing

Imagine we're asked to develop a system to identify person and organization names in our company's collection of one million documents. The common pre-processing steps we discussed earlier may not be relevant in this context. Identifying names requires us to be able to do POS tagging, as identifying proper nouns can be useful in identifying person and organization names. How do we do POS tagging during the pre-processing stage of the project? We're not going into the details of how POS taggers are developed (see Chapter 8 in [38] for details) in this book. Pre-trained and readily usable POS taggers are implemented in NLP libraries such as NLTK, spaCy [39], and Parsey McParseface Tagger [40], and we generally don't have to develop our own POS-tagging solutions. The following code snippet illustrates how to use many of the pre-built pre-processing functions we've discussed so far using the NLP library spaCy:

```
import spacy
nlp = spacy.load('en_core_web_sm')
doc = nlp(u'Charles Spencer Chaplin was born on 16 April 1889 toHannah Chaplin
(born Hannah Harriet
Pedlingham Hill) and Charles Chaplin Sr')
for token in doc:
    print(token.text, token.lemma_, token.pos_,
          token.shape_, token.is_alpha, token.is_stop)
```

In this simple snippet, we can see tokenization, lemmatization, POS tagging, and several other steps in action! Note that if needed we can add additional processing steps with the same code snippet; we'll leave that as an exercise for the reader. A point to note is that there may be differences in the output among different NLP libraries for the same pre-processing step. This is due in part to implementation differences and algorithmic variations among different libraries. Which library (or libraries) you'll

eventually want to use in your project is a subjective decision based on the amount of language processing you want.

Let's now consider a slightly different problem: along with identifying person and organization names in our company's collection of one million documents, we're also asked to identify if a given person and organization are related to each other in some way (e.g., Satya Nadella is related to Microsoft through the relation CEO). This is known as the problem of *relation extraction*, which we'll discuss in greater detail in [Chapter 5](#). But for now, think about what kind of pre-processing we need for this case. We need POS tagging, which we already know how to add to our pipeline. We need a way of identifying person and organization names, which is a separate information extraction task known as *named entity recognition (NER)*, which we'll discuss in [Chapter 5](#). Apart from these two, we need a way to identify patterns indicating "relation" between two entities in a sentence. This requires us to have some form of syntactic representation of the sentence, such as parsing, which we saw in [Chapter 1](#). Further, we also want a way to identify and link multiple mentions of an entity (e.g., Satya Nadella, Mr. Nadella, he, etc.). We accomplish this with the pre-processing step known as *coreference resolution*. We saw an example of this in "[An NLP Walkthrough: Conversational Agents](#)" on page 31. [Figure 2-10](#) shows the output from Stanford CoreNLP [41], which illustrates a parser output and coreference resolution output for an example sentence, along with other pre-processing steps we discussed previously.

What we've seen so far in this section are some of the most common pre-processing steps in a pipeline. They're all available as pre-trained, usable models in different NLP libraries. Apart from these, additional, customized pre-processing may be necessary, depending on the application. For example, consider a case where we're asked to mine the social media sentiment on our product. We start by collecting data from, say, Twitter, and quickly realize there are tweets that are not in English. In such cases, we may also need a language-detection step before doing anything else.

Additionally, what steps we need also depends on a specific application. If we're creating a system to identify whether the reviewer is expressing a positive or negative sentiment about a movie from a review they wrote, we might not worry much about parsing or coreference resolution, but we would want to consider stop word removal, lowercasing, and removing digits. However, if we're interested instead in extracting calendar events from emails, we'll probably be better off not removing stop words or doing stemming, but rather including, say, parsing. In the case where we want to extract relationships between different entities in the text and events mentioned in it, we would need coreference resolution, as we discussed previously. We'll see examples of cases requiring such steps in [Chapter 5](#).

## Input

Chaplin wrote, directed, and composed the music for most of his films.

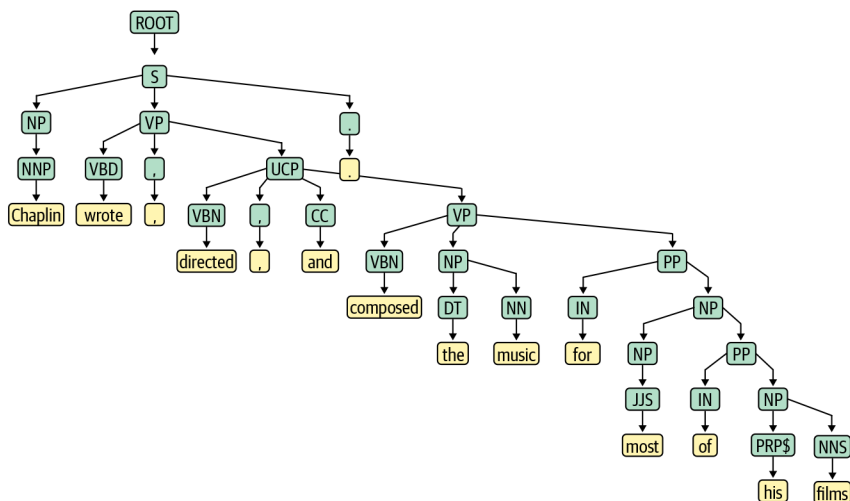
## Tokenization with Lemmatization

Chaplin wrote, directed, and composed the music for most of his films.

## POS Tagging

Chaplin wrote, directed, and composed the music for most of his films.

## Parse Tree



## Coreference Resolution

Chaplin wrote, directed, and composed the music for most of his films.

Figure 2-10. Output from different stages of NLP pipeline processing

Finally, we have to consider the step-by-step procedures of pre-processing in each case, as summarized in Figure 2-11.

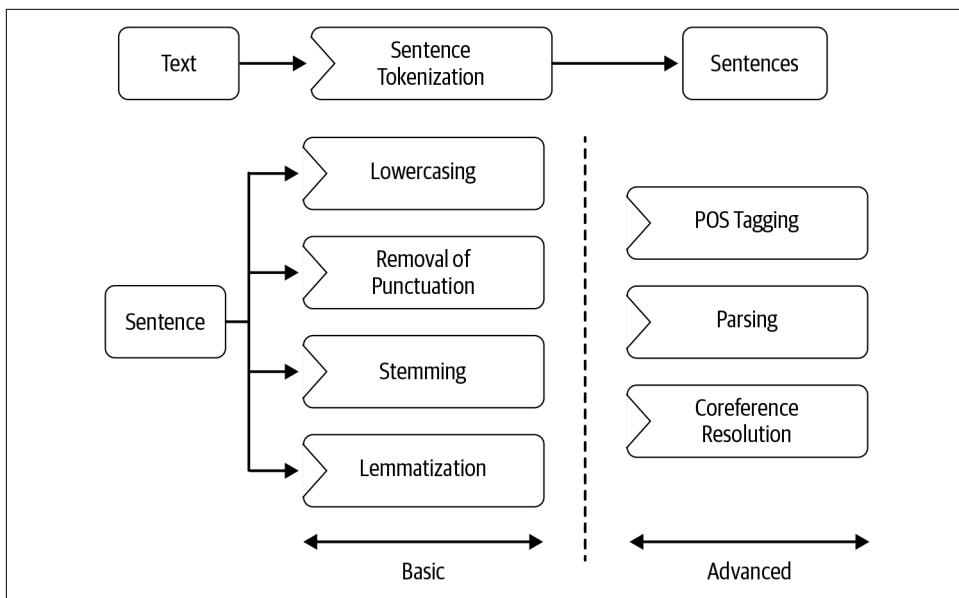


Figure 2-11. Advanced pre-processing steps on a blob of text

For example, POS tagging cannot be preceded by stop word removal, lowercasing, etc., as such processing affects POS tagger output by changing the grammatical structure of the sentence. How a particular pre-processing step is helping a given NLP problem is another question that is specific to the application, and it can only be answered with a lot of experimentation. We'll discuss more specific pre-processing required for different NLP applications in upcoming chapters. For now, let's move on to the next step: feature engineering.

## Feature Engineering

So far, we've seen different pre-processing steps and where they can be useful. When we use ML methods to perform our modeling step later, we'll still need a way to feed this pre-processed text into an ML algorithm. *Feature engineering* refers to the set of methods that will accomplish this task. It's also referred to as *feature extraction*. The goal of feature engineering is to capture the characteristics of the text into a numeric vector that can be understood by the ML algorithms. We refer to this step as “text representation” in this book, and it's the topic of [Chapter 3](#). We also detail feature extraction in the context of developing a complete NLP pipeline and iterating to improve performance in [Chapter 11](#). Here, we'll briefly touch on two different approaches taken in practice for feature engineering in (1) a classical NLP and traditional ML pipeline and (2) a DL pipeline. [Figure 2-12](#) (adapted from [42]) distinguishes the two approaches.