

NLP Pipeline

Sheet-2

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

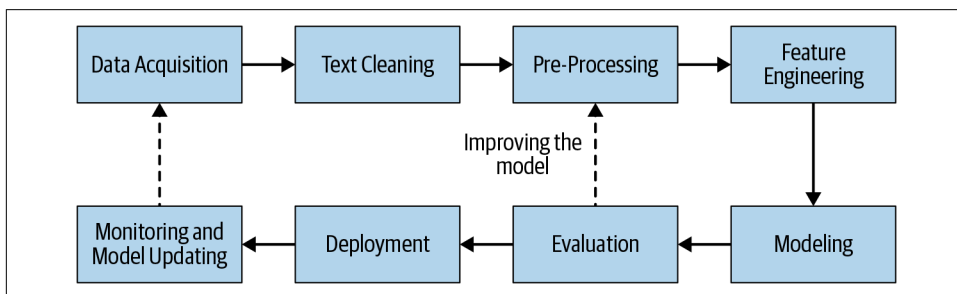


Figure 2-1. Generic NLP pipeline

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.

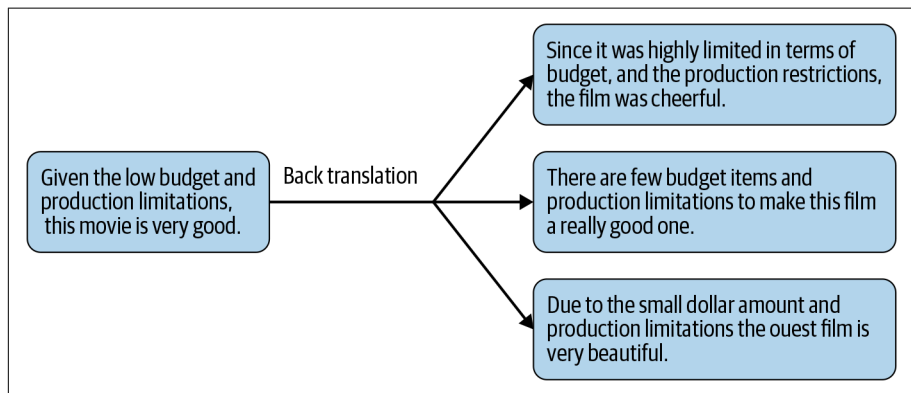


Figure 2-2. Back translation

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

2

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](#).

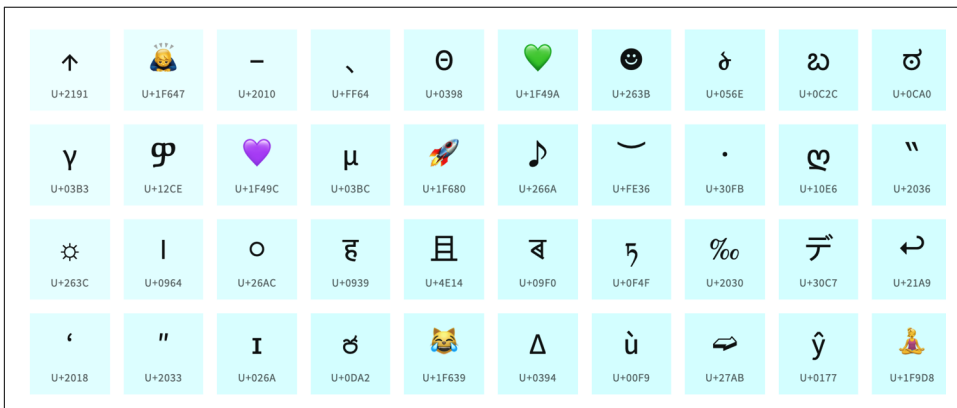


Figure 2-4. Unicode characters [18]

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  
  }  
]
```

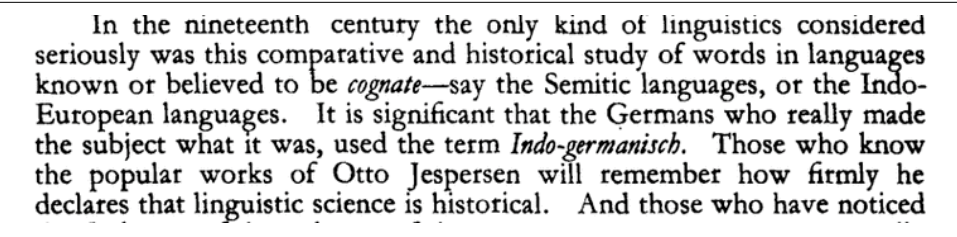
You can see the full tutorial in [21].

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].

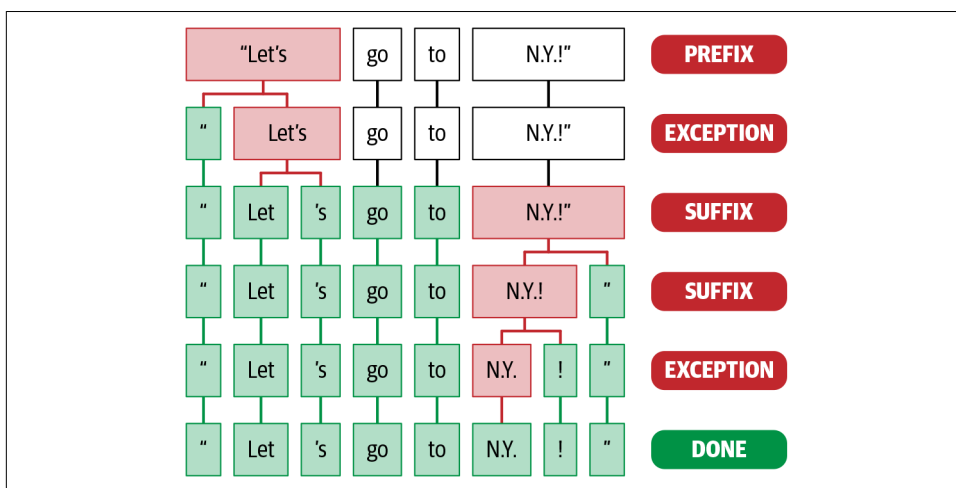


Figure 2-6. Language-specific (English here) exceptions during tokenization [31]

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.

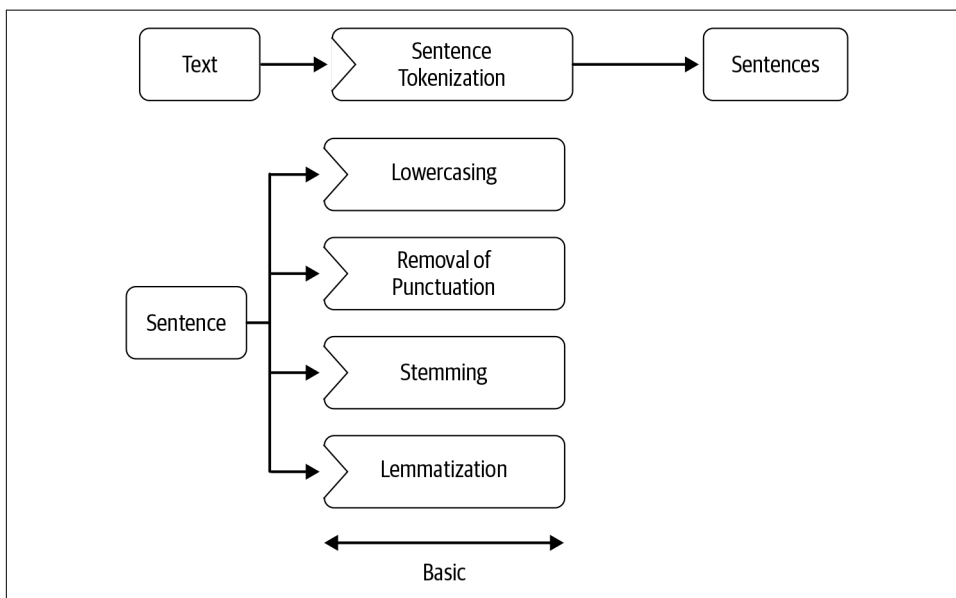


Figure 2-8. Common pre-processing steps on a blob of text

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.

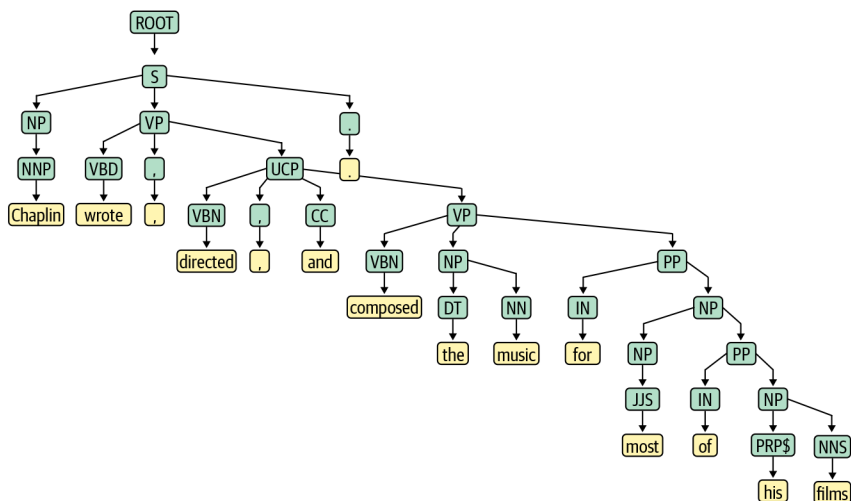
Tokenization with Lemmatization

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.

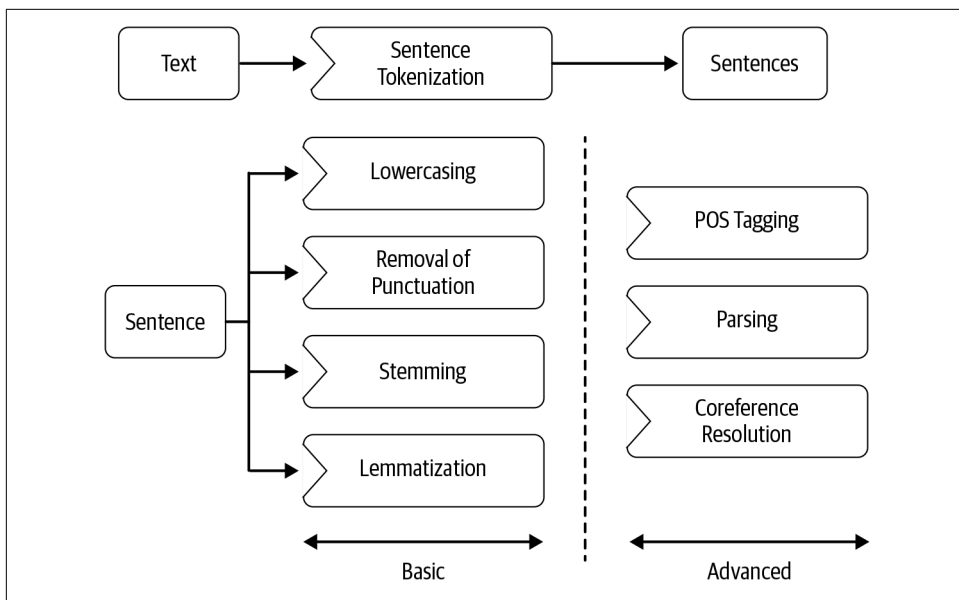


Figure 2-11. Advanced pre-processing steps on a blob of text

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.

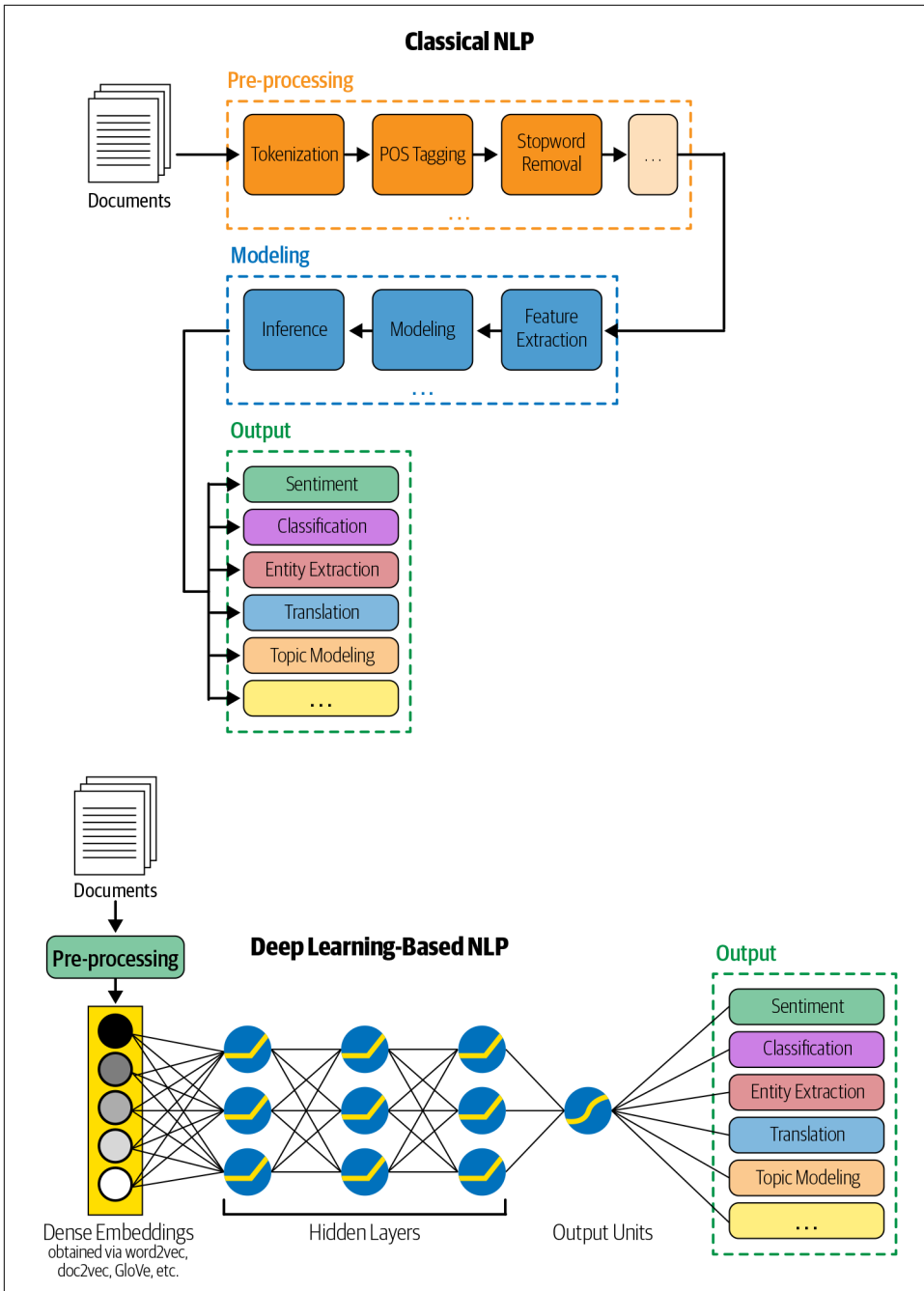


Figure 2-12. Feature engineering for classical NLP versus DL-based NLP

Classical NLP/ML Pipeline

Feature engineering is an integral step in any ML pipeline. Feature engineering steps **convert the raw data into a format that can be consumed by a machine**. These transformation functions are usually handcrafted in the classical ML pipeline, aligning to the task at hand. For example, imagine a task of sentiment classification on product reviews in e-commerce. One way to **convert the reviews into meaningful “numbers” that helps predict the reviews’ sentiments** (positive or negative) would be to count the number of positive and negative words in each review. There are statistical measures for understanding if a feature is useful for a task or not; we’ll discuss this in **Chapter 11**. The main takeaway for building classical ML models is that the **features are heavily inspired by the task at hand as well as domain knowledge** (for example, using sentiment words in the review example). One of the advantages of handcrafted features is that the model remains interpretable—it’s possible to quantify exactly how much each feature is influencing the model prediction.

DL Pipeline

The main drawback of classical ML models is the feature engineering. Handcrafted feature engineering becomes a bottleneck for both model performance and the model development cycle. **A noisy or unrelated feature can potentially harm the model’s performance by adding more randomness to the data**. Recently, with the advent of DL models, this approach has changed. In the DL pipeline, the raw data (after pre-processing) is directly fed to a model. **The model is capable of “learning” features from the data**. Hence, these features are more in line with the task at hand, so they generally give improved performance. But, since all these features are learned via model parameters, the model loses interpretability. **It’s very hard to explain a DL model’s prediction, which is a disadvantage in a business-driven use case**. For example, when identifying an email as ham or spam, it might be worth knowing which word or phrases played the significant role in making the email ham or spam. While this is easy to do with handcrafted features, it’s not easy in the case of DL models.

As we’ve already mentioned, feature engineering is heavily task specific, so we discuss it throughout the book in the context of textual data and a range of tasks. With a high-level understanding of feature engineering, now let’s take a look at the next step in the pipeline, which we call *modeling*.

Modeling

We now have some amount of data related to our NLP project and a clear idea of what sort of cleaning up and pre-processing needs to be done and what features are to be extracted. The next step is about how to **build a useful solution** out of this. At the start, when we have limited data, we can use simpler methods and rules. Over

time, with more data and a better understanding of the problem, we can add more complexity and improve performance. We'll cover this process in this section.

Start with Simple Heuristics

At the very start of building a model, ML may not play a major role by itself. Part of that could be due to a lack of data, but human-built heuristics can also provide a great start in some ways. Heuristics may already be part of your system, either implicitly or explicitly. For instance, in email spam-classification tasks, we may have a blacklist of domains that are used exclusively to send spam. This information can be used to filter emails from those domains. Similarly, a blacklist of words in an email that denote a high chance of spam could also be used for this classification.

Such heuristics can be found in a range of tasks, especially at the start of applying ML. In an e-commerce setting, we may use a heuristic based on the number of purchases for ordering search results and show products belonging to the same category as recommendations while we collect data that could be used to build a larger, collaborative, filtering-based system that can recommend products using a range of other characteristics based on what customers with similar buying profiles purchased.

Another popular approach to incorporating heuristics in your system is using regular expressions. Let's say we're developing a system to extract different forms of information from text documents, such as dates and phone numbers, names of people who work in a given organization, etc. While some information, such as email IDs, dates, and telephone numbers can be extracted using normal (albeit complex) regular expressions, Stanford NLP's TokensRegex [43] and spaCy's rule-based matching [20] are two tools that are useful for defining advanced regular expressions to capture other information, such as people who work in a specific organization. Figure 2-13 shows an example of spaCy's rule-based matcher in action.

This shows a pattern that looks for text containing the lemma "match," appearing as a noun, optionally preceded by an adjective, and followed by any word form of lemma "be." Such patterns are an advanced form of regular expressions, which require some of the NLP pre-processing steps we saw earlier in this chapter. In the absence of large amounts of training data, and when we have some domain knowledge, we can start building systems by encoding this knowledge in the form of rules/heuristics. Even when we're building ML-based models, we can use such heuristics to handle special cases—for example, cases where the model has failed to learn well. Thus, simple heuristics can give us a good starting point and be useful in ML models. Now, assuming we built such a heuristics-based system, where do we go from there?

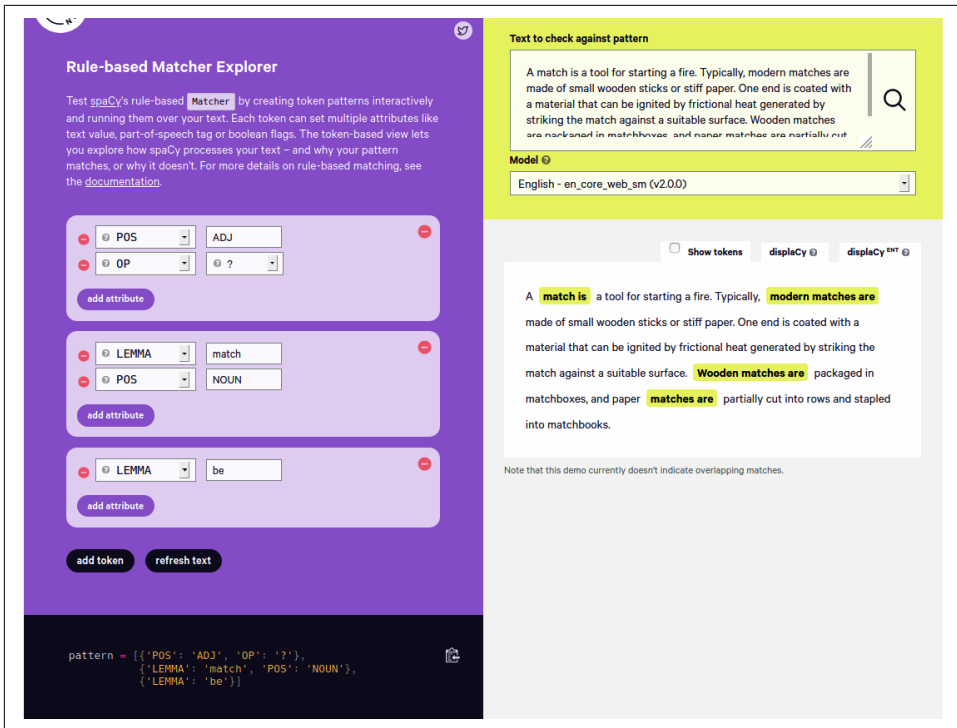


Figure 2-13. spaCy's rule-based matcher

Building Your Model

While a set of simple heuristics is a good start, as our system matures, adding newer and newer heuristics may result in a complex, rule-based system. Such a system is hard to manage, and it can be even harder to diagnose the cause of errors. We need a system that's easier to maintain as it matures. Further, as we collect more data, our ML model starts beating pure heuristics. At that point, a common practice is to combine heuristics directly or indirectly with the ML model. There are two broad ways of doing that:

Create a feature from the heuristic for your ML model

When there are many heuristics where the behavior of a single heuristic is deterministic but their combined behavior is fuzzy in terms of how they predict, it's best to use these heuristics as features to train your ML model. For instance, in the email spam-classification example, we can add features, such as the number of words from the blacklist in a given email or the email bounce rate, to the ML model.

Pre-process your input to the ML model

If the heuristic has a really high prediction for a particular kind of class, then it's best to use it before feeding the data in your ML model. For instance, if for certain words in an email, there's a 99% chance that it's spam, then it's best to classify that email as spam instead of sending it to an ML model.

Additionally, we have NLP service providers, such as Google Cloud Natural Language [44], Amazon Comprehend [45], Microsoft Azure Cognitive Services [46], and IBM Watson Natural Language Understanding [47], which provide off-the-shelf APIs to solve various NLP tasks. If your project has an NLP problem that's addressed by these APIs, you can start by using them to get an estimate of the feasibility of the task and how good your existing dataset is. Once you're comfortable that the task is feasible and conclude that the off-the-shelf models give reasonable results, you can move toward building custom ML models and improving them.

Building THE Model

We've seen examples of getting started building an NLP system by using heuristics or existing APIs, or by building our own ML models. We start with a baseline approach and work toward improving it. We may have to do many iterations of the model-building process to "build THE model" that gives good performance and is also production-ready. We cover some of the approaches to address this issue here:

Ensemble and stacking

In our experience, a common practice is not to have a single model, but to use a collection of ML models, often dealing with different aspects of the prediction problem. There are two ways of doing this: we can feed one model's output as input for another model, thus sequentially going from one model to another and obtaining a final output. This is called *model stacking*.ⁱ Alternatively, we can also pool predictions from multiple models and make a final prediction. This is called *model ensembling*. Figure 2-14 demonstrates both of these procedures.

In this figure, training data is used to build Models 1, 2, and 3. Outputs of these models are then combined to be used in a meta-model (a model that uses other models) to predict the final outcome. For example, in the email spam-classification case, we can assume that we run three different models: a heuristic-based score, Naive Bayes, and LSTM. The output of these three models is then fed into the meta-model based on logistic regression, which then gives the chances of the email being spam or not. As the product grows in terms of its features, the model will also grow in complexity. So, we may eventually end up using a

i. This is different from the vertical stacking done in neural networks like LSTM.

combination of all of these—i.e., heuristics, machine learning, and stacked and ensemble models—as part of a large product.

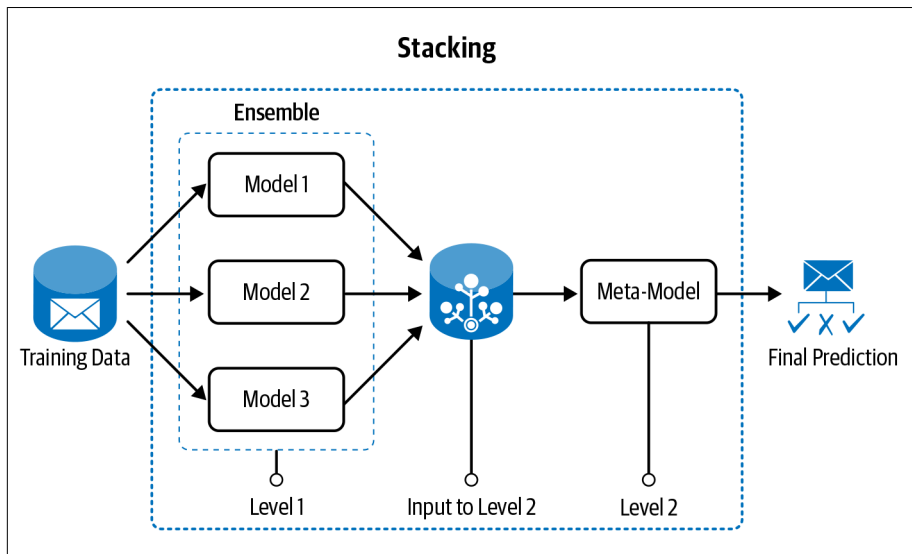


Figure 2-14. Model ensemble and stacking

Better feature engineering

For both API-based and custom-built models, feature engineering is an important step, and it evolves throughout the process. A better feature engineering step may lead to better performance. For instance, if there are a lot of features, then we use feature selection to find a better model. We detail strategies for iterating feature engineering to achieve an optimal setting in Chapter 11.

Transfer learning

Apart from model stacking or ensemble, there is a newer trend that's becoming popular in the NLP community—*transfer learning*, which we introduced in Chapter 1. Often, the model needs external knowledge beyond the dataset for the task to understand the language and the problem well. Transfer learning tries to transfer preexisting knowledge from a big, well-trained model to a newer model at its initial phase. Afterward, the new model slowly adapts to the task at hand. This is analogous to a teacher transferring wisdom and knowledge to a student. Transfer learning provides a better initialization, which helps in the downstream tasks, especially when the dataset for the downstream task is smaller. In these cases, transfer learning yields better results than just initializing a downstream model from scratch with random initialization. As an example, for email spam classification, we can use BERT to fine-tune the email dataset. We cover BERT in greater detail in Chapters 4 through 6.

Reapplying heuristics

No ML model is perfect. Hence, ML models still make mistakes. It's possible to revisit these cases again at the end of the modeling pipeline to **find any common pattern in errors and use heuristics to correct them**. We can also apply domain-specific knowledge that is not automatically captured in the data to refine the model predictions. An analogy for reapplying heuristics would be our model as a trapeze artist performing great feats and these rules as the safety net so the artist doesn't fall off.

Between the stage of having no data, when we fully rely on heuristics, to a lot of data, where we can try a range of modeling techniques, we encounter a situation where we have a small amount of data, which is often **not sufficient to build good ML models**. In such scenarios, one approach to follow is **active learning**, where we can use **user feedback or other such sources to continuously collect new data to build better models**. We'll discuss this in detail in **Chapter 4**. As we've just seen, modeling strategies depend heavily on the data at hand. **Table 2-1** provides a range of decision paths given our data volume and quality, based on our experience.

Table 2-1. Data attributes and associated decision paths

Data attribute	Decision path	Examples
Large data volume	Can use techniques that require more data, like DL. Can use a richer set of features as well. If the data is sufficiently large but unlabeled, we can also apply unsupervised techniques.	If we have a lot of reviews and metadata associated with them, we can build a sentiment-analysis tool from scratch .
Small data volume	Need to start with rule-based or traditional ML solutions that are less data hungry. Can also adapt cloud APIs and generate more data with weak supervision. We can also use transfer learning if there's a similar task that has large data.	This often happens at the start of a completely new project .
Data quality is poor and the data is heterogeneous in nature	More data cleaning and pre-processing might be required.	This entails issues like code mixing (different languages being mixed in the same sentence) , unconventional language, transliteration, or noise (like social media text).
Data quality is good	Can directly apply off-the-shelf algorithms or cloud APIs more easily.	Legal text or newspapers .
Data consists of full-length documents	Choose the right strategy for breaking the document into lower levels , like paragraphs, sentences, or phrases, depending on the problem.	Document classification, review analysis , etc.

So far, we've seen an overview of different forms of modeling that can be useful in an NLP pipeline and what modeling path to choose based on the data we have. **Supervised learning, especially classification, is the most common modeling process** you'll

encounter in the NLP projects you'll be building in an industry scenario. We'll discuss classification models in [Chapter 4](#) and models used for different application scenarios in NLP in Chapters 5 through 7. Now, let's take a look at the next step in the pipeline: evaluation.

Evaluation

A key step in the NLP pipeline is to measure **how good the model** we've built is. "Goodness" of a model can have multiple meanings, but the most common interpretation is the **measure of the model's performance on unseen data**. Success in this phase depends on two factors: (1) **using the right metric for evaluation**, and (2) **following the right evaluation process**. Let's first focus on 1. Depending on the NLP task or problem, the evaluation metrics can vary. They can also vary depending on the phase: the model building, deployment, and production phases. Whereas in the first two phases, we typically use ML metrics, in the final phase, we also include business metrics to measure business impact.

Also, evaluations are of two types: intrinsic and extrinsic. Intrinsic focuses on **intermediary objectives**, while extrinsic focuses on **evaluating performance on the final objective**. For example, consider a spam-classification system. The ML metric will be precision and recall, while the business metric will be "the amount of time users spent on a spam email." Intrinsic evaluation will focus on measuring the system performance using precision and recall. Extrinsic evaluation will focus on measuring the time a user wasted because a spam email went to their inbox or a genuine email went to their spam folder.

Intrinsic Evaluation

In this section, we'll look at some intrinsic evaluation metrics that are commonly used to measure NLP systems. For most metrics in this category, we assume a test set where we have the *ground truth* or *labels* (human annotated, correct answers). Labels could be binary (e.g., 0/1 for text classification), one-to-two words (e.g., names for named entity recognition), or large text itself (e.g., text translated by machine translation). **The output of the NLP model on a data point is compared against the corresponding label for that data point, and metrics are calculated based on the match (or mismatch) between the output and label.** For most NLP tasks, the comparison can be automated, hence intrinsic evaluation can be automated. For some cases, like machine translation or summarization, it's not always possible to automate evaluation since comparison is not subjective.

[Table 2-2](#) lists various metrics used for intrinsic evaluation across various NLP tasks. For a more detailed discussion of the metrics, refer to the corresponding reference.

15

Table 2-2. Popular metrics and NLP applications where they're used

Metric	Description	Applications
Accuracy [48]	Used when the output variable is categorical or discrete. It denotes the fraction of times the model makes correct predictions as compared to the total predictions it makes.	Mainly used in classification tasks, such as sentiment classification (multiclass), natural language inference (binary), paraphrase detection (binary), etc.
Precision [48]	Shows how precise or exact the model's predictions are, i.e., given all the positive (the class we care about) cases, how many can the model classify correctly?	Used in various classification tasks, especially in cases where mistakes in a positive class are more costly than mistakes in a negative class, e.g., disease predictions in healthcare.
Recall [48]	Recall is complementary to precision. It captures how well the model can recall positive class, i.e., given all the positive predictions it makes, how many of them are indeed positive?	Used in classification tasks, especially where retrieving positive results is more important, e.g., e-commerce search and other information-retrieval tasks.
F1 score [49]	Combines precision and recall to give a single metric, which also captures the trade-off between precision and recall, i.e., completeness and exactness. F1 is defined as $(2 \times \text{Precision} \times \text{Recall}) / (\text{Precision} + \text{Recall})$.	Used simultaneously with accuracy in most of the classification tasks. It is also used in sequence-labeling tasks, such as entity extraction, retrieval-based questions answering, etc.
AUC [48]	Captures the count of positive predictions that are correct versus the count of positive predictions that are incorrect as we vary the threshold for prediction.	Used to measure the quality of a model independent of the prediction threshold. It is used to find the optimal prediction threshold for a classification task.
MRR (mean reciprocal rank) [50]	Used to evaluate the responses retrieved given their probability of correctness. It is the mean of the reciprocal of the ranks of the retrieved results.	Used heavily in all information-retrieval tasks, including article search, e-commerce search, etc.
MAP (mean average precision) [51]	Used in ranked retrieval results, like MRR. It calculates the mean precision across each retrieved result.	Used in information-retrieval tasks.
RMSE (root mean squared error) [48]	Captures a model's performance in a real-value prediction task. Calculates the square root of the mean of the squared errors for each data point.	Used in conjunction with MAPE in the case of regression problems, from temperature prediction to stock market price prediction.
MAPE (mean absolute percentage error) [52]	Used when the output variable is a continuous variable. It is the average of absolute percentage error for each data point.	Used to test the performance of a regression model. It is often used in conjunction with RMSE.
BLEU (bilingual evaluation understudy) [53]	Captures the amount of n-gram overlap between the output sentence and the reference ground truth sentence. It has many variants.	Mainly used in machine-translation tasks. Recently adapted to other text-generation tasks, such as paraphrase generation and text summarization.
METEOR [54]	A precision-based metric to measure the quality of text generated. It fixes some of the drawbacks of BLEU, such as exact word matching while calculating precision. METEOR allows synonyms and stemmed words to be matched with the reference word.	Mainly used in machine translation.

Metric	Description	Applications
ROUGE [55]	Another metric to compare quality of generated text with respect to a reference text. As opposed to BLEU, it measures recall.	Since it measures recall, it's mainly used for summarization tasks where it's important to evaluate how many words a model can recall.
Perplexity [56]	A probabilistic measure that captures how confused an NLP model is. It's derived from the cross-entropy in a next word prediction task. The exact definition can be found at [56].	Used to evaluate language models. It can also be used in language-generation tasks, such as dialog generation.

Apart from the list of metrics shown in Table 2-2, there are few more metrics and visualizations that are often used for solving NLP problems. While we've covered these topics briefly here, we encourage you to follow the references and learn more about these metrics.

In the case of classification tasks, a commonly used visual evaluation method is a *confusion matrix*. It allows us to inspect the actual and predicted output for different classes in the dataset. The name stems from the fact that it helps to understand how “confused” the classification model is in terms of identifying different classes. A confusion matrix is in turn used to compute metrics such as precision, recall, F1 score, and accuracy. We'll see how to use confusion matrices in Chapter 4.

Ranking tasks like information search and retrieval mostly uses ranking-based metrics, such as MRR and MAP, but usual classification metrics can be used, too. In the case of retrieval, we care mainly about recall, so recall at various ranks is calculated. For example, for information retrieval, a common metric is “Recall at rank K”; it looks for the presence of ground truth in top K retrieved results. If present, it's a success.

When it comes to text-generation tasks, there are a number of metrics that are used, depending on the task. Even though BLEU and METEOR are good metrics for machine translation, they may not be good metrics when applied to other generation tasks. For example, in the case of dialog generation, the ground truth is one of the correct answers, but there could be many variations in responses that are not listed. In cases like this, precision-based metrics such as BLEU and METEOR will completely fail to capture the task performance faithfully. For these reasons, perplexity is one metric that's used extensively to understand a model's text-generation ability.

However, any evaluation scheme for text generation is not perfect. This is because there could be multiple sentences that have the same meaning, and it's not possible to have all the variations listed as ground truth. Therefore, the text generated and the ground truth can have the same meaning but be different sentences. This makes automated evaluation a difficult process. For example, say we build a machine-translation model that converts sentences from French to English. Consider the following sentence in French: “J'ai mangé trois filberts.” In English, this means, “I ate three filberts.”

So, we put this sentence as the label. Say our model generates the following English translation: “I ate three hazelnuts.” Since the output does not match the label, automated evaluation will say the output is incorrect. But this *evaluation* is incorrect because English speakers are known to refer to filberts as hazelnuts. Even if we add this sentence as a possible label, our model could still generate “I have eaten three hazelnuts” as output. Yet again, the automated evaluation will say the model got it wrong since the output does not match either of the two labels. This is where human evaluation comes into play. But human evaluation can be expensive both in terms of time and money.

Extrinsic Evaluation

Like we said earlier, extrinsic evaluation focuses on evaluating the model performance on the final objective. In industrial projects, any AI model is built with the aim of solving a business problem. For example, a regression model is built with the aim of ranking the emails of the users and bringing the most important emails to the top of the inbox, thereby helping the users of an email service save time. Consider a scenario where the regression model does well on the ML metrics but doesn't really save a lot of time for the email service users, or where a question-answering model does very well on intrinsic metrics but fails to address a large number of questions in the production environment. Would we call such models successful? No, because they failed to achieve their business objectives. While this is not an issue for researchers in academia, for practitioners in industry, it's very important.

The way to carry out extrinsic evaluation is to set up the business metrics and the process to measure them correctly at the start of the project. We'll see examples of the right business metrics in later chapters.

We might ask: if extrinsic evaluation is what matters, why do intrinsic evaluation at all? The reason we must do intrinsic evaluation before extrinsic evaluation is that extrinsic evaluation often includes project stakeholders outside the AI team—sometimes even end users. Intrinsic evaluation can be done mostly by the AI team itself. This makes extrinsic evaluation a much more expensive process as compared to intrinsic evaluation. Therefore, intrinsic evaluation is used as a proxy for extrinsic evaluation. Only when we get consistently good results in intrinsic evaluation should we go for extrinsic evaluation.

Another thing to remember is that bad results in intrinsic evaluation often imply bad results in extrinsic evaluation. However, the converse may not be true. That is, we can have a model that does very well in intrinsic evaluation but does badly in extrinsic evaluation, but it's unlikely that a model that does well in extrinsic evaluation did poorly during intrinsic evaluation. The reasons for poor performance in extrinsic evaluation could be many, from setting up the wrong metrics to not having suitable

data or having wrong expectations. We touched on some of these in [Chapter 1](#) and will discuss them in more detail in [Chapter 11](#).

So far, we've seen some metrics commonly used for intrinsic evaluation and also discussed the importance of extrinsic evaluation to measure the performance of NLP models. There are some more metrics that are task specific, which are not seen across different NLP application scenarios. We'll discuss such evaluation measures in detail as we cover these specific applications in upcoming chapters. With this, now let's look at the next components of the pipeline: model deployment, monitoring, and updating.

Post-Modeling Phases

Once our model has been tried and tested, we move on to the **post-modeling phase: deploying, monitoring, and updating the model**. We'll cover these briefly in this section.

Deployment

In most practical application scenarios, the NLP module we're implementing is a part of a larger system (e.g., a spam-classification system in a larger email application). Thus, working through the processing, modeling, and evaluation pipeline is only a part of the story. Eventually, once we're happy with one final solution, **it needs to be deployed in a production environment as a part of a larger system**. Deployment entails plugging the NLP module into the broader system. It may also involve making sure input and output data pipelines are in order, as well as making sure our NLP module is scalable under heavy load.

An **NLP module is typically deployed as a web service**. Let's say we designed a web service that takes a text as input and returns the email's category (spam or non-spam) as output. Now, **each time someone gets a new email, it goes to the microservice, which classifies the email text**. This, in turn, can be used to make a decision about what to do with the email (either show it or send it to the spam folder). In certain circumstances, like batch processing, the NLP module is deployed in the larger task queue. As an example, take a look at task queues in Google Cloud [57] or AWS [58]. We'll cover deployment in more detail in [Chapter 11](#).

Monitoring

Like with any software engineering project, extensive software testing has to be done before final deployment, and the **model performance is monitored constantly after deploying**. Monitoring for NLP projects and models has to be handled differently than a regular engineering project, as we need to **ensure that the outputs produced by our models daily make sense**. If we're automatically training the model frequently, we

have to make sure that the models behave in a reasonable manner. Part of this is done through a performance dashboard showing the model parameters and key performance indicators. We'll discuss this more in [Chapter 11](#).

Model Updating

Once the model is deployed and we start gathering new data, we'll iterate the model based on this new data to stay current with predictions. We cover this model update for each task throughout the book, especially in Chapters 4 through 7 and [Chapter 11](#). As a start, [Table 2-3](#) gives some guidance on how to approach the model updating process for different post-deployment scenarios.

Table 2-3. Project attribute and associated decision paths

Project attribute	Decision paths	Examples
More training data is generated post-deployment.	Once deployed, extracted signals can be used to automatically improve the model. Can also try online learning to train the model automatically on a daily basis.	Abuse-detection systems where users flag data.
Training data is not generated post-deployment.	Manual labeling could be done to improve evaluation and the models. Ideally, each new model has to be manually built and evaluated.	A subset of a larger NLP pipeline with no direct feedback.
Low model latency is required, or model has to be online with near-real-time response.	Need to use models that can be inferred quickly. Another option is to create memoization strategies like caching or have substantially bigger computing power.	Systems that need to respond right away, like any chatbot or an emergency tracking system.
Low model latency is not required, or model can be run in an offline fashion.	Can use more advanced and slower models. This can also help in optimizing costs where feasible.	Systems that can be run on a batch process, like retail product catalog analysis.

Working with Other Languages

So far, our discussion has assumed that we're dealing mostly with English text. Depending on the task at hand, we may need to build models and solutions for other languages as well. How we approach this will change based on what language we're dealing with. The pipeline for some languages may be very similar to English, whereas some languages and scenarios may require us to rethink how we approach the problem. We've compiled some action points for dealing with different languages in [Table 2-4](#), based on our experiences working on projects that involve non-English language processing.

Table 2-4. Language attribute and action plan

Language attribute	Example and languages	Action
High-resource languages	Languages that have both ample data as well as pre-built models. Examples include English, French, and Spanish.	Possible to use pre-trained DL models. Easier to use.
Low-resource languages	Languages that have limited data and recent digital adoption. May not have pre-built models. Examples include Swahili, Burmese, and Uzbek.	Depending on the task, may need to label more data as well as explore individual components.
Morphologically rich	Linguistic and grammatical information like subject, object, predicate, tense, and mode are not separate words, but are joined together. Examples include Latin, Turkish, Finnish, and Malayalam.	If the language is not resource rich, we'll need to explore morphological analyzers that exist for the language. In the worst case, manual rules to handle certain cases might be needed.
Vocabulary variation heavy	Nonstandard spellings and high word variation. For Arabic and Hindi, the spellings are nonstandard.	If the language is not resource rich, then we may need to first normalize the words/spellings before training any model. This may not be needed for languages with large datasets, as they can still learn of vocabulary variation.
CJK languages	These languages are derived from ancient Chinese characters. They're not alphabet based and have several thousand characters for basic literacy and over 40,000 characters for larger coverage. Thus, they have to be handled differently. They include Chinese, Japanese, and Korean, hence the name CJK.	Use specific tokenization schemes in these languages. Given that an ample amount of CJK data is available, it's possible to build NLP models for various tasks from scratch. There are also pre-trained models for them. Transfer learning from models trained in other languages beyond CJK may not be useful in this case.

Next, we'll turn our attention to a case study that will put all these steps together.

Case Study

So far, we've seen different stages of an NLP pipeline. At each stage, we discussed what it's about, why it's useful, and how it fits into the general framework of an NLP pipeline. However, we tackled these individual stages separately, away from the over-all context. How do all these stages work together in a real-world NLP system pipeline? Let's see a case study, using Uber's tool to improve customer care: Customer Obsession Ticketing Assistant (COTA).

Uber operates in 400+ cities worldwide, and based on the number of people who use Uber every day, we can expect that their customer support teams receive several hundreds of thousands of tickets on different issues each day. There are a couple of solutions to choose from for a given ticket. The goal of COTA is to rank these solutions

and pick the best possible one. Uber developed COTA using ML and NLP techniques to enable better customer support and quick and efficient resolution of such tickets. Figure 2-15 shows the pipeline in Uber's COTA and the various NLP components in it.

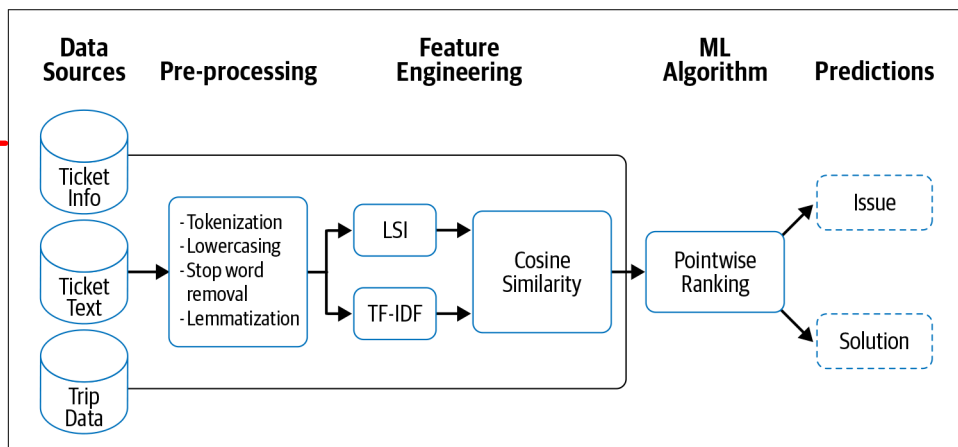


Figure 2-15. NLP pipeline for ranking tickets in a ticketing system by Uber [59]

The information needed to identify the ticket issue and select a solution in this system comes from three sources, as shown in the figure. Ticket text is, as the name indicates, textual content, which is where NLP comes into the picture. After cleaning up the text by removing HTML tags (not shown in the figure), the pre-processing steps consist of tokenization, lowercasing, stop word removal, and lemmatization. We saw how to do all of these earlier in this chapter. After pre-processing, the ticket text is represented as a collection of words (known as a *bag of words* and discussed in detail in Chapter 3).

The next step in this pipeline is feature engineering. The bag of words we obtained earlier is fed to two NLP modules—TF-IDF (term frequency and inverse document frequency) and LSI (latent semantic indexing)—which are used to understand the meaning of a text using this bag of words representation. This process comes under the NLP task called topic modeling, which we'll discuss in Chapter 7. Exactly how Uber uses these NLP tasks in this context is an interesting idea: Uber collects the historical tickets for each solution from their database, forms a bag-of-words vector representation for each solution, and creates a topic model based on these representations. An incoming ticket is then mapped to this topic space of solutions, creating a vector representation for the ticket. Cosine similarity is a common measure of similarity between any two vectors. It is used to create a vector where each element indicates the ticket text's similarity to one solution. Thus, at the end of this feature engineering step, we end up with a representation indicating the ticket text's similarity to all possible solutions.

In the next stage, modeling, this representation is combined with ticket information and trip data to build a ranking system that shows the three best solutions for the ticket. Under the hood, this ranking model consists of a binary classification system, which classifies each ticket-solution combination as a match or mismatch. The matches are then ranked based on a scoring function. [59] describes more details on the implementation of this system pipeline.

The next step in our pipeline is evaluation. How does evaluation work in this context? While the evaluation of model performance itself can be done in terms of an intrinsic evaluation measure such as MRR, the overall effectiveness of this approach is evaluated extrinsically. It's estimated that COTA's quick ticket resolution saves Uber tens of millions of dollars every year.

As we learned earlier, a model is not built just once. COTA, too, was continually experimented with and improved upon. After exploring a range of DL architectures, the best solution that was ultimately chosen resulted in a 10% greater accuracy compared to the previous version with the binary classification-based ranking system. The process does not end here, though. As we can see from the COTA team's article [59], it's a continuous process of model deployment, monitoring, and updating.

Wrapping Up

In this chapter, we saw the different steps involved in developing an NLP pipeline for a given project description and saw a detailed case study of a real-world application. We also saw how a traditional NLP pipeline and a DL-based NLP pipeline differ from each other and learned what to do when working with non-English languages. Aside from the case study, we looked at these steps in a more general manner in this chapter. Specific details for each step will depend on the task at hand and the purpose of our implementation. We'll look at a few task-specific pipelines from Chapter 4 onward, describing in detail what's unique as well as common across different tasks while designing such pipelines. In the next chapter, we'll tackle the question of text representation that we mentioned briefly earlier in this chapter.

References

- [1] Iderhoff, Nicolas. [nlp-datasets: Alphabetical list of free/public domain datasets with text data for use in Natural Language Processing \(NLP\)](#), (GitHub repo). Last accessed June 15, 2020.
- [2] Google. [“Dataset Search”](#). Last accessed June 15, 2020.
- [3] Miller, George A. “WordNet: A Lexical Database for English.” *Communications of the ACM* 38.11 (1995): 39–41.
- [4] NTLTK documentation. [“WordNet Interface”](#). Last accessed June 15, 2020.

- [5] Xie, Qizhe, Zihang Dai, Eduard Hovy, Minh-Thang Luong, and Quoc V. Le. “Unsupervised Data Augmentation for Consistency Training”. (2019).
- [6] Wikipedia. “Fat-finger error”. Last modified January 26, 2020.
- [7] Snorkel. “Programmatically Building and Managing Training Data”. Last accessed June 15, 2020.
- [8] Ratner, Alexander, Stephen H. Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. “Snorkel: Rapid Training Data Creation with Weak Supervision.” *The VLDB Journal* 29 (2019): 1–22.
- [9] Bach, Stephen H., Daniel Rodriguez, Yintao Liu, Chong Luo, Haidong Shao, Cassandra Xia, Souvik Sen et al. “Snorkel DryBell: A Case Study in Deploying Weak Supervision at Industrial Scale”. (2018).
- [10] Wei, Jason W., and Kai Zou. “Eda: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks”, (2019).
- [11] GitHub repository for [10]. Last accessed June 15, 2020.
- [12] Ma, Edward. *nplaug: Data augmentation for NLP*, (GitHub repo). Last accessed June 15, 2020.
- [13] Shioulin and Nisha. “A Guide to Learning with Limited Labeled Data”. April 2, 2019.
- [14] eForms. “Blank Invoice Templates”. Last accessed June 15, 2020.
- [15] Amazon.com. “Amazon Elements Vitamin B12 Methylcobalamin 5000 mcg - Normal Energy Production and Metabolism, Immune System Support - 2 Month Supply (65 Berry Flavored Lozenges)”. Last accessed June 15, 2020.
- [16] Beautiful Soup. Last accessed June 15, 2020.
- [17] Scrapy.org. *Scrapy*. Last accessed June 15, 2020.
- [18] Unicode. Last accessed June 15, 2020.
- [19] Dickinson, Markus, Chris Brew, and Detmar Meurers. *Language and Computers*. New Jersey: John Wiley & Sons, 2012. ISBN: 978-1-405-18305-5
- [20] Explosion.ai. “Rule-based matching”. Last accessed June 15, 2020.
- [21] Microsoft documentation. “Quickstart: Check spelling with the Bing Spell Check REST API and Python”. Last accessed June 15, 2020.
- [22] Stamy, Matthew. *PyPDF2: A utility to read and write PDFs with Python*, (GitHub repo). Last accessed June 15, 2020.
- [23] pdfminer. *pdfminer.six: Community maintained fork of pdfminer*, (GitHub repo). Last accessed June 15, 2020.

- [24] FilingDB. “What’s so hard about PDF text extraction?” Last accessed June 15, 2020.
- [25] Tesseract-OCR. “Tesseract Open Source OCR Engine (main repository)”, (GitHub repo). Last accessed June 15, 2020.
- [26] Python-tesseract documentation [Python-tesseract](#). Last accessed June 15, 2020.
- [27] Firth, John Rupert. “Personality and Language in Society.” *The Sociological Review* 42.1 (1950): 37–52.
- [28] pyenchant. [Spellchecking library for python](#), (GitHub repo). Last accessed June 15, 2020.
- [29] KBNL Research. [ochre: Toolbox for OCR post-correction](#), (GitHub repo). Last accessed June 15, 2020.
- [30] “Natural Language ToolKit”. Last accessed June 15, 2020.
- [31] Explosion.ai. “[spaCy 101: Everything you need to know](#)”. Last accessed June 15, 2020.
- [32] Evang, Kilian, Valerio Basile, Grzegorz Chrupała, and Johan Bos. “Elephant: Sequence Labeling for Word and Sentence Segmentation.” *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (2013): 1422–1426.
- [33] Porter, Martin F. “An Algorithm For Suffix Stripping.” *Program: electronic library and information systems* 14.3 (1980): 130–137.
- [34] Padmanabhan, Arvind. “[Lemmatization](#)”. October 11, 2019.
- [35] Explosion.ai. “[spaCy](#)”. Last accessed June 15, 2020.
- [36] Polyglot documentation. [Polyglot Python library](#). Last accessed June 15, 2020.
- [37] Mair, Victor. “[Singlish: alive and well](#)”. May 14, 2016.
- [38] Jurafsky, Dan and James H. Martin. [Speech and Language Processing, Third Edition \(Draft\)](#), 2018.
- [39] Explosion.ai. “[spaCy: Industrial-Strength Natural Language Processing in Python](#)”. Last accessed June 15, 2020.
- [40] DeepAI. “[Parsey Mcparseface API](#)”. Last accessed June 15, 2020.
- [41] Stanford CoreNLP. [Stanford CoreNLP – Natural language software](#). Last accessed June 15, 2020.
- [42] Ghaffari, Parsa. “[Leveraging Deep Learning for Multilingual Sentiment Analysis](#)”. July 14, 2016.

- [43] The Stanford Natural Language Processing Group. “[Stanford TokensRegex](#)”. Last accessed June 15, 2020.
- [44] Google. “[Cloud Natural Language](#)”. Last accessed June 15, 2020.
- [45] Amazon. “[AWS Comprehend](#)”. Last accessed June 15, 2020.
- [46] Microsoft. “[Azure Cognitive Services documentation](#)”. Last accessed June 15, 2020.
- [47] IBM. “[Watson Natural Language Understanding](#)”. Last accessed June 15, 2020.
- [48] Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. *The Elements of Statistical Learning*, Second Edition. New York: Springer, 2001. ISBN: 978-0-387-84857-0
- [49] Wikipedia. “[F1 score](#)”. Last modified April 18, 2020.
- [50] Wikipedia. “[Mean reciprocal rank](#)”. Last modified December 6, 2018.
- [51] Wikipedia. “[Evaluation measures \(information retrieval\)](#)”. Last modified February 12, 2020.
- [52] Wikipedia. “[Mean absolute percentage error](#)”. Last modified February 6, 2020.
- [53] Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu. “BLEU: A Method for Automatic Evaluation of Machine Translation.” *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (2002): 311–318.
- [54] Banerjee, Satanjeev and Alon Lavie. “METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments.” *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization* (2005): 65–72.
- [55] Lin, Chin-Yew. “ROUGE: A Package for Automatic Evaluation of Summaries.” *Text Summarization Branches Out* (2004): 74–81.
- [56] Wikipedia. “[Perplexity](#)”. Last modified February 13, 2020.
- [57] Google Cloud. “[Quickstart for Cloud Tasks queues](#)”. Last accessed June 15, 2020.
- [58] Amazon. “[Amazon Simple Queue Service](#)”. Last accessed June 15, 2020.
- [59] Zheng, Huaixiu., Yi-Chia Wang, and Piero Molino. “[COTA: Improving Uber Customer Care with NLP & Machine Learning](#)”. January 3, 2018.