Natural Language Processing with Java

Natural Language Processing (NLP) is an important area of application development and its relevance in addressing contemporary problems will only increase in the future. There has been a significant increase in the demand for natural language-accessible applications supported by NLP tasks.

Natural Language Processing with Java will explore how to automatically organize text using approaches such as full-text search, proper name recognition, clustering, tagging, information extraction, and summarization. It covers concepts of NLP that even those of you without a background in statistics or natural language processing can understand.

Who this book is written for

If you are a Java programmer who wants to learn about the fundamental tasks underlying natural language processing, this book is for you. You will be able to identify and use NLP tasks for many common problems, and integrate them in your applications to solve more difficult problems. Readers should be familiar/experienced with Java software development.

What you will learn from this book

- Develop a deep understanding of the basic NLP tasks and how they relate to each other
- Discover and use the available tokenization engines
- Implement techniques for end of sentence detection
- Apply search techniques to find people and things within a document
- Construct solutions to identify parts of speech within sentences
- Use parsers to extract relationships between elements of a document
- Integrate basic tasks to tackle more complex NLP problems

Explore various approaches to organize and extract useful text from unstructured data using Java

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Introduction to NLP'
- A synopsis of the book’s content
- More information on Natural Language Processing with Java

About the Author

Richard M Reese has worked in both industry and academics. For 17 years, he worked in the telephone and aerospace industries, serving in several capacities, including research and development, software development, supervision, and training. He currently teaches at Tarleton State University, where he is able to apply his years of industry experience to enhance his classes.

Richard has written several Java and C books. He uses a concise and easy-to-follow approach to topics at hand. His books include EJB 3.1 Cookbook; books about new features of Java 7 and 8, Java Certification, and jMonkey Engine; and a book on C pointers.

I would like to thank my daughter, Jennifer, for the numerous reviews and contributions she has made. Her input has been invaluable.
Natural Language Processing with Java

Natural Language Processing (NLP) has been used to address a wide range of problems, including support for search engines, summarizing and classifying text for web pages, and incorporating machine learning technologies to solve problems such as speech recognition and query analysis. It has found use wherever documents contain useful information.

NLP is used to enhance the utility and power of applications. It does so by making user input easier and converting text to more usable forms. In essence, NLP processes natural text found in a variety of sources, using a series of core NLP tasks to transform or extract information from the text.

This book focuses on core NLP tasks that will likely be encountered in an NLP application. Each NLP task presented in this book starts with a description of the problem and where it can be used. The issues that make each task difficult are introduced so that you can understand the problem in a better way. This is followed by the use of numerous Java techniques and APIs to support an NLP task.
What This Book Covers

Chapter 1, Introduction to NLP, explains the importance and uses of NLP. The NLP techniques used in this chapter are explained with simple examples illustrating their use.

Chapter 2, Finding Parts of Text, focuses primarily on tokenization. This is the first step in more advanced NLP tasks. Both core Java and Java NLP tokenization APIs are illustrated.

Chapter 3, Finding Sentences, proves that sentence boundary disambiguation is an important NLP task. This step is a precursor for many other downstream NLP tasks where text elements should not be split across sentence boundaries. This includes ensuring that all phrases are in one sentence and supporting parts of speech analysis.

Chapter 4, Finding People and Things, covers what is commonly referred to as Named Entity Recognition. This task is concerned with identifying people, places, and similar entities in text. This technique is a preliminary step for processing queries and searches.

Chapter 5, Detecting Parts of Speech, shows you how to detect parts of speech, which are grammatical elements of text, such as nouns and verbs. Identifying these elements is a significant step in determining the meaning of text and detecting relationships within text.

Chapter 6, Classifying Texts and Documents, proves that classifying text is useful for tasks such as spam detection and sentiment analysis. The NLP techniques that support this process are investigated and illustrated.

Chapter 7, Using Parser to Extract Relationships, demonstrates parse trees. A parse tree is used for many purposes, including information extraction. It holds information regarding the relationships between these elements. An example implementing a simple query is presented to illustrate this process.

Chapter 8, Combined Approaches, contains techniques for extracting data from various types of documents, such as PDF and Word files. This is followed by an examination of how the previous NLP techniques can be combined into a pipeline to solve larger problems.
Natural Language Processing (NLP) is a broad topic focused on the use of computers to analyze natural languages. It addresses areas such as speech processing, relationship extraction, document categorization, and summation of text. However, these types of analysis are based on a set of fundamental techniques such as tokenization, sentence detection, classification, and extracting relationships. These basic techniques are the focus of this book. We will start with a detailed discussion of NLP, investigate why it is important, and identify application areas.

There are many tools available that support NLP tasks. We will focus on the Java language and how various Java Application Programmer Interfaces (APIs) support NLP. In this chapter, we will briefly identify the major APIs, including Apache's OpenNLP, Stanford NLP libraries, LingPipe, and GATE.

This is followed by a discussion of the basic NLP techniques illustrated in this book. The nature and use of these techniques is presented and illustrated using one of the NLP APIs. Many of these techniques will use models. Models are similar to a set of rules that are used to perform a task such as tokenizing text. They are typically represented by a class that is instantiated from a file. We round off the chapter with a brief discussion on how data can be prepared to support NLP tasks.

NLP is not easy. While some problems can be solved relatively easily, there are many others that require the use of sophisticated techniques. We will strive to provide a foundation for NLP processing so that you will be able to understand better which techniques are available and applicable for a given problem.
NLP is a large and complex field. In this book, we will only be able to address a small part of it. We will focus on core NLP tasks that can be implemented using Java. Throughout this book, we will demonstrate a number of NLP techniques using both the Java SE SDK and other libraries, such as OpenNLP and Stanford NLP. To use these libraries, there are specific API JAR files that need to be associated with the project in which they are being used. A discussion of these libraries is found in the Survey of NLP tools section and contains download links to the libraries. The examples in this book were developed using NetBeans 8.0.2. These projects required the API JAR files to be added to the Libraries category of the Projects Properties dialog box.

What is NLP?

A formal definition of NLP frequently includes wording to the effect that it is a field of study using computer science, artificial intelligence, and formal linguistics concepts to analyze natural language. A less formal definition suggests that it is a set of tools used to derive meaningful and useful information from natural language sources such as web pages and text documents.

Meaningful and useful implies that it has some commercial value, though it is frequently used for academic problems. This can readily be seen in its support of search engines. A user query is processed using NLP techniques in order to generate a result page that a user can use. Modern search engines have been very successful in this regard. NLP techniques have also found use in automated help systems and in support of complex query systems as typified by IBM's Watson project.

When we work with a language, the terms, syntax, and semantics, are frequently encountered. The syntax of a language refers to the rules that control a valid sentence structure. For example, a common sentence structure in English starts with a subject followed by a verb and then an object such as "Tim hit the ball". We are not used to unusual sentence order such as "Hit ball Tim". Although the rule of syntax for English is not as rigorous as that for computer languages, we still expect a sentence to follow basic syntax rules.

The semantics of a sentence is its meaning. As English speakers, we understand the meaning of the sentence "Tim hit the ball". However, English and other natural languages can be ambiguous at times and a sentence's meaning may only be determined from its context. As we will see, various machine learning techniques can be used to attempt to derive the meaning of text.

As we progress with our discussions, we will introduce many linguistic terms that will help us better understand natural languages and provide us with a common vocabulary to explain the various NLP techniques. We will see how the text can be split into individual elements and how these elements can be classified.
In general, these approaches are used to enhance applications, thus making them more valuable to their users. The uses of NLP can range from relatively simple uses to those that are pushing what is possible today. In this book, we will show examples that illustrate simple approaches, which may be all that is required for some problems, to the more advanced libraries and classes available to address sophisticated needs.

**Why use NLP?**

NLP is used in a wide variety of disciplines to solve many different types of problems. Text analysis is performed on text that ranges from a few words of user input for an Internet query to multiple documents that need to be summarized. We have seen a large growth in the amount and availability of unstructured data in recent years. This has taken forms such as blogs, tweets, and various other social media. NLP is ideal for analyzing this type of information.

Machine learning and text analysis are used frequently to enhance an application's utility. A brief list of application areas follow:

- **Searching**: This identifies specific elements of text. It can be as simple as finding the occurrence of a name in a document or might involve the use of synonyms and alternate spelling/misspelling to find entries that are close to the original search string.

- **Machine translation**: This typically involves the translation of one natural language into another.

- **Summation**: Paragraphs, articles, documents, or collections of documents may need to be summarized. NLP has been used successfully for this purpose.

- **Named Entity Recognition (NER)**: This involves extracting names of locations, people, and things from text. Typically, this is used in conjunction with other NLP tasks such as processing queries.

- **Information grouping**: This is an important activity that takes textual data and creates a set of categories that reflect the content of the document. You have probably encountered numerous websites that organize data based on your needs and have categories listed on the left-hand side of the website.

- **Parts of Speech Tagging (POS)**: In this task, text is split up into different grammatical elements such as nouns and verbs. This is useful in analyzing the text further.

- **Sentiment analysis**: People's feelings and attitudes regarding movies, books, and other products can be determined using this technique. This is useful in providing automated feedback with regards to how well a product is perceived.
• **Answering queries:** This type of processing was illustrated when IBM's Watson successfully won a Jeopardy competition. However, its use is not restricted to winning game shows and has been used in a number of other fields including medicine.

• **Speech recognition:** Human speech is difficult to analyze. Many of the advances that have been made in this field are the result of NLP efforts.

• **Natural Language Generation:** This is the process of generating text from a data or knowledge source, such as a database. It can automate reporting of information such as weather reports, or summarize medical reports.

NLP tasks frequently use different machine learning techniques. A common approach starts with training a model to perform a task, verifying that the model is correct, and then applying the model to a problem. We will examine this process further in *Understanding NLP models* later in the chapter.

**Why is NLP so hard?**

NLP is not easy. There are several factors that makes this process hard. For example, there are hundreds of natural languages, each of which has different syntax rules. Words can be ambiguous where their meaning is dependent on their context. Here, we will examine a few of the more significant problem areas.

At the character level, there are several factors that need to be considered. For example, the encoding scheme used for a document needs to be considered. Text can be encoded using schemes such as ASCII, UTF-8, UTF-16, or Latin-1. Other factors such as whether the text should be treated as case-sensitive or not may need to be considered. Punctuation and numbers may require special processing. We sometimes need to consider the use of emoticons (character combinations and special character images), hyperlinks, repeated punctuation (… or ---), file extension, and usernames with embedded periods. Many of these are handled by preprocessing text as we will discuss in *Preparing data* later in the chapter.

When we **Tokenize** text, it usually means we are breaking up the text into a sequence of words. These words are called **Tokens**. The process is referred to as **Tokenization**. When a language uses whitespace characters to delineate words, this process is not too difficult. With a language like Chinese, it can be quite difficult since it uses unique symbols for words.

Words and morphemes may need to be assigned a part of speech label identifying what type of unit it is. A **Morpheme** is the smallest division of text that has meaning. Prefixes and suffixes are examples of morphemes. Often, we need to consider synonyms, abbreviation, acronyms, and spellings when we work with words.
Stemming is another task that may need to be applied. Stemming is the process of finding the word stem of a word. For example, words such as "walking", "walked", or "walks" have the word stem "walk". Search engines often use stemming to assist in asking a query.

Closely related to stemming is the process of Lemmatization. This process determines the base form of a word called its lemma. For example, for the word "operating", its stem is "oper" but its lemma is "operate". Lemmatization is a more refined process than stemming and uses vocabulary and morphological techniques to find a lemma. This can result in more precise analysis in some situations.

Words are combined into phrases and sentences. Sentence detection can be problematic and is not as simple as looking for the periods at the end of a sentence. Periods are found in many places including abbreviations such as Ms. and in numbers such as 12.834.

We often need to understand which words in a sentence are nouns and which are verbs. We are sometimes concerned with the relationship between words. For example, Coreferences resolution determines the relationship between certain words in one or more sentences. Consider the following sentence:

"The city is large but beautiful. It fills the entire valley."

The word "it" is the coreference to city. When a word has multiple meanings we might need to perform Word Sense Disambiguation to determine the meaning that was intended. This can be difficult to do at times. For example, "John went back home".

Does the home refer to a house, a city, or some other unit? Its meaning can sometimes be inferred from the context in which it is used. For example, "John went back home. It was situated at the end of a cul-de-sac."

In spite of these difficulties, NLP is able to perform these tasks reasonably well in most situations and provide added value to many problem domains. For example, sentiment analysis can be performed on customer tweets resulting in possible free product offers for dissatisfied customers. Medical documents can be readily summarized to highlight the relevant topics and improved productivity.

Summarization is the process of producing a short description of different units. These units can include multiple sentences, paragraphs, a document, or multiple documents. The intent may be to identify those sentences that convey the meaning of the unit, determine the prerequisites for understanding a unit, or to find items within these units. Frequently, the context of the text is important in accomplishing this task.
Survey of NLP tools

There are many tools available that support NLP. Some of these are available with the Java SE SDK but are limited in their utility for all but the simplest types of problems. Other libraries such as Apache's OpenNLP and LingPipe provide extensive and sophisticated support for NLP problems.

Low-level Java support includes string libraries, such as `String`, `StringBuilder`, and `StringBuffer`. These classes possess methods that perform searching, matching, and text replacement. **Regular expressions** use special encoding to match substrings. Java provides a rich set of techniques to use regular expressions.

As discussed earlier, tokenizers are used to split text into individual elements. Java provides supports for tokenizers with:

- The `String` class' `split` method
- The `StreamTokenizer` class
- The `StringTokenizer` class

There also exists a number of NLP libraries/APIs for Java. A partial list of Java-based NLP APIs are found in the following table. Most of these are open source. In addition, there are a number of commercial APIs available. We will focus on the open source APIs:

<table>
<thead>
<tr>
<th>API</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Architecture for Text Engineering</td>
<td><a href="http://gate.ac.uk/">http://gate.ac.uk/</a></td>
</tr>
<tr>
<td>Learning Based Java</td>
<td><a href="http://cogcomp.cs.illinois.edu/page/software_view/11">http://cogcomp.cs.illinois.edu/page/software_view/11</a></td>
</tr>
<tr>
<td>LingPipe</td>
<td><a href="http://alias-i.com/lingpipe/">http://alias-i.com/lingpipe/</a></td>
</tr>
<tr>
<td>Mallet</td>
<td><a href="http://mallet.cs.umass.edu/">http://mallet.cs.umass.edu/</a></td>
</tr>
</tbody>
</table>
Many of these NLP tasks are combined to form a **pipeline**. A pipeline consists of various NLP tasks, which are integrated into a series of steps to achieve some processing goal. Examples of frameworks that support pipelines are GATE and Apache UIMA.

In the next section, we will cover several NLP APIs in more depth. A brief overview of their capabilities will be presented along with a list of useful links for each API.

**Apache OpenNLP**

The Apache OpenNLP project addresses common NLP tasks and will be used throughout this book. It consists of several components that perform specific tasks, permit models to be trained, and support for testing the models. The general approach, used by OpenNLP, is to instantiate a model that supports the task from a file and then executes methods against the model to perform a task.

For example, in the following sequence, we will tokenize a simple string. For this code to execute properly, it must handle the `FileNotFoundException` and `IOException` exceptions. We use a try-with-resource block to open a `FileInputStream` instance using the `en-token.bin` file. This file contains a model that has been trained using English text:

```java
try (InputStream is = new FileInputStream(
    new File(getModelDir(), "en-token.bin")){
    // Insert code to tokenize the text
    }
    } catch (FileNotFoundException ex) {
        ...
    } catch (IOException ex) {
        ...
    }
```

An instance of the `TokenizerModel` class is then created using this file inside the try block. Next, we create an instance of the `Tokenizer` class, as shown here:

```java
TokenizerModel model = new TokenizerModel(is);
Tokenizer tokenizer = new TokenizerME(model);
```

The `tokenize` method is then applied, whose argument is the text to be tokenized. The method returns an array of `String` objects:

```java
String tokens[] = tokenizer.tokenize("He lives at 1511 W." + "Randolph.");
```
A for-each statement displays the tokens as shown here. The open and close brackets are used to clearly identify the tokens:

```java
for (String a : tokens) {
    System.out.print("[" + a + "] ");
} System.out.println();
```

When we execute this, we will get output as shown here:

```
[He] [lives] [at] [1511] [W.] [Randolph] [.]
```

In this case, the tokenizer recognized that w. was an abbreviation and that the last period was a separate token demarking the end of the sentence.

We will use the OpenNLP API for many of the examples in this book. OpenNLP links are listed in the following table:

<table>
<thead>
<tr>
<th>OpenNLP</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td><a href="https://opennlp.apache.org/">https://opennlp.apache.org/</a></td>
</tr>
<tr>
<td>Documentation</td>
<td><a href="https://opennlp.apache.org/documentation.html">https://opennlp.apache.org/documentation.html</a></td>
</tr>
<tr>
<td>Download</td>
<td><a href="https://opennlp.apache.org/cgi-bin/download.cgi">https://opennlp.apache.org/cgi-bin/download.cgi</a></td>
</tr>
<tr>
<td>Wiki</td>
<td><a href="https://cwiki.apache.org/confluence/display/OPENNLP/Index%3bjsessionid=32B408C73729ACCCDD071D9EC354FC54">https://cwiki.apache.org/confluence/display/OPENNLP/Index%3bjsessionid=32B408C73729ACCCDD071D9EC354FC54</a></td>
</tr>
</tbody>
</table>

**Stanford NLP**

The Stanford NLP Group conducts NLP research and provides tools for NLP tasks. The Stanford CoreNLP is one of these toolsets. In addition, there are other tool sets such as the Stanford Parser, Stanford POS tagger, and the Stanford Classifier. The Stanford tools support English and Chinese languages and basic NLP tasks, including tokenization and name entity recognition.

These tools are released under the full GPL but it does not allow them to be used in commercial applications, though a commercial license is available. The API is well organized and supports the core NLP functionality.

There are several tokenization approaches supported by the Stanford group. We will use the PTBTokenizer class to illustrate the use of this NLP library. The constructor demonstrated here uses a Reader object, a LexedTokenFactory<T> argument, and a string to specify which of the several options is to be used.
The LexedTokenFactory is an interface that is implemented by the CoreLabelTokenFactory and WordTokenFactory classes. The former class supports the retention of the beginning and ending character positions of a token, whereas the latter class simply returns a token as a string without any positional information. The WordTokenFactory class is used by default.

The CoreLabelTokenFactory class is used in the following example. A StringReader is created using a string. The last argument is used for the option parameter, which is null for this example. The Iterator interface is implemented by the PTBTokenizer class allowing us to use the hasNext and next methods to display the tokens:

```java
PTBTokenizer ptb = new PTBTokenizer(
    new StringReader("He lives at 1511 W. Randolph."),
    new CoreLabelTokenFactory(), null);
while (ptb.hasNext()) {
    System.out.println(ptb.next());
}
```

The output is as follows:

```
He
lives
at
1511
W.
Randolph.
```

We will use the Stanford NLP library extensively in this book. A list of Stanford links is found in the following table. Documentation and download links are found in each of the distributions:

<table>
<thead>
<tr>
<th>Stanford NLP</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td><a href="http://nlp.stanford.edu/index.shtml">http://nlp.stanford.edu/index.shtml</a></td>
</tr>
<tr>
<td>CoreNLP</td>
<td><a href="http://nlp.stanford.edu/software/corenlp.shtml#Download">http://nlp.stanford.edu/software/corenlp.shtml#Download</a></td>
</tr>
<tr>
<td>Parser</td>
<td><a href="http://nlp.stanford.edu/software/lex-parser.shtml">http://nlp.stanford.edu/software/lex-parser.shtml</a></td>
</tr>
<tr>
<td>POS Tagger</td>
<td><a href="http://nlp.stanford.edu/software/tagger.shtml">http://nlp.stanford.edu/software/tagger.shtml</a></td>
</tr>
<tr>
<td>java-nlp-user</td>
<td><a href="https://mailman.stanford.edu/mailman/listinfo/java-nlp-user">https://mailman.stanford.edu/mailman/listinfo/java-nlp-user</a></td>
</tr>
</tbody>
</table>
LingPipe

LingPipe consists of a set of tools to perform common NLP tasks. It supports model training and testing. There are both royalty free and license versions of the tool. The production use of the free version is limited.

To demonstrate the use of LingPipe, we will illustrate how it can be used to tokenize text using the `Tokenizer` class. Start by declaring two lists, one to hold the tokens and a second to hold the whitespace:

```java
List<String> tokenList = new ArrayList<>();
List<String> whiteList = new ArrayList<>();
```

Next, declare a string to hold the text to be tokenized:

```java
String text = "A sample sentence processed \n by \t the " + "LingPipe tokenizer.";
```

Now, create an instance of the `Tokenizer` class. As shown in the following code block, a static `tokenizer` method is used to create an instance of the `Tokenizer` class based on a Indo-European factory class:

```java
Tokenizer tokenizer = IndoEuropeanTokenizerFactory.INSTANCE.tokenizer(text.toCharArray(), 0, text.length());
```

The `tokenize` method of this class is then used to populate the two lists:

```java
tokenizer.tokenize(tokenList, whiteList);
```

Use a for-each statement to display the tokens:

```java
for(String element : tokenList) {
    System.out.print(element + " ");
}
System.out.println();
```
The output of this example is shown here:

A sample sentence processed by the LingPipe tokenizer

A list of LingPipe links can be found in the following table:

<table>
<thead>
<tr>
<th>LingPipe</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td><a href="http://alias-i.com/lingpipe/index.html">http://alias-i.com/lingpipe/index.html</a></td>
</tr>
<tr>
<td>Tutorials</td>
<td><a href="http://alias-i.com/lingpipe/demos/tutorial/read-me.html">http://alias-i.com/lingpipe/demos/tutorial/read-me.html</a></td>
</tr>
<tr>
<td>JavaDocs</td>
<td><a href="http://alias-i.com/lingpipe/docs/api/index.html">http://alias-i.com/lingpipe/docs/api/index.html</a></td>
</tr>
<tr>
<td>Download</td>
<td><a href="http://alias-i.com/lingpipe/web/install.html">http://alias-i.com/lingpipe/web/install.html</a></td>
</tr>
<tr>
<td>Core</td>
<td><a href="http://alias-i.com/lingpipe/web/download.html">http://alias-i.com/lingpipe/web/download.html</a></td>
</tr>
<tr>
<td>Models</td>
<td><a href="http://alias-i.com/lingpipe/web/models.html">http://alias-i.com/lingpipe/web/models.html</a></td>
</tr>
</tbody>
</table>

**GATE**

**General Architecture for Text Engineering (GATE)** is a set of tools written in Java and developed at the University of Sheffield in England. It supports many NLP tasks and languages. It can also be used as a pipeline for NLP processing.

It supports an API along with GATE Developer, a document viewer that displays text along with annotations. This is useful for examining a document using highlighted annotations. GATE Mimir, a tool for indexing and searching text generated by various sources, is also available. Using GATE for many NLP tasks involves a bit of code. GATE Embedded is used to embed GATE functionality directly in code. Useful GATE links are listed in the following table:

<table>
<thead>
<tr>
<th>Gate</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td><a href="https://gate.ac.uk/">https://gate.ac.uk/</a></td>
</tr>
<tr>
<td>Docs</td>
<td><a href="https://gate.ac.uk/documentation.html">https://gate.ac.uk/documentation.html</a></td>
</tr>
<tr>
<td>JavaDocs</td>
<td><a href="http://jenkins.gate.ac.uk/job/GATE-Nightly/javadoc/">http://jenkins.gate.ac.uk/job/GATE-Nightly/javadoc/</a></td>
</tr>
<tr>
<td>Download</td>
<td><a href="https://gate.ac.uk/download/">https://gate.ac.uk/download/</a></td>
</tr>
<tr>
<td>Wiki</td>
<td><a href="http://gatewiki.sf.net/">http://gatewiki.sf.net/</a></td>
</tr>
</tbody>
</table>
Introduction to NLP

UIMA

The Organization for the Advancement of Structured Information Standards (OASIS) is a consortium focused on information-oriented business technologies. It developed the Unstructured Information Management Architecture (UIMA) standard as a framework for NLP pipelines. It is supported by the Apache UIMA.

Although it supports pipeline creation, it also describes a series of design patterns, data representations, and user roles for the analysis of text. UIMA links are listed in the following table:

<table>
<thead>
<tr>
<th>Apache UIMA</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td><a href="https://uima.apache.org/">https://uima.apache.org/</a></td>
</tr>
<tr>
<td>Documentation</td>
<td><a href="https://uima.apache.org/documentation.html">https://uima.apache.org/documentation.html</a></td>
</tr>
<tr>
<td>JavaDocs</td>
<td><a href="https://uima.apache.org/d/uimaj-2.6.0/apidocs/index.html">https://uima.apache.org/d/uimaj-2.6.0/apidocs/index.html</a></td>
</tr>
<tr>
<td>Download</td>
<td><a href="https://uima.apache.org/downloads.cgi">https://uima.apache.org/downloads.cgi</a></td>
</tr>
<tr>
<td>Wiki</td>
<td><a href="https://cwiki.apache.org/confluence/display/UIMA/Index">https://cwiki.apache.org/confluence/display/UIMA/Index</a></td>
</tr>
</tbody>
</table>

Overview of text processing tasks

Although there are numerous NLP tasks that can be performed, we will focus only on a subset of these tasks. A brief overview of these tasks is presented here, which is also reflected in the following chapters:

- Finding Parts of Text
- Finding Sentences
- Finding People and Things
- Detecting Parts of Speech
- Classifying Text and Documents
- Extracting Relationships
- Combined Approaches

Many of these tasks are used together with other tasks to achieve some objective. We will see this as we progress through the book. For example, tokenization is frequently used as an initial step in many of the other tasks. It is a fundamental and basic step.
Finding parts of text

Text can be decomposed into a number of different types of elements such as words, sentences, and paragraphs. There are several ways of classifying these elements. When we refer to parts of text in this book, we are referring to words, sometimes called tokens. Morphology is the study of the structure of words. We will use a number of morphology terms in our exploration of NLP. However, there are many ways of classifying words including the following:

- **Simple words**: These are the common connotations of what a word means including the 17 words of this sentence.
- **Morphemes**: These are the smallest units of a word that is meaningful. For example, in the word "bounded", "bound" is considered to be a morpheme. Morphemes also include parts such as the suffix, "ed".
- **Prefix/Suffix**: This precedes or follows the root of a word. For example, in the word graduation, the "ation" is a suffix based on the word "graduate".
- **Synonyms**: This is a word that has the same meaning as another word. Words such as small and tiny can be recognized as synonyms. Addressing this issue requires word sense disambiguation.
- **Abbreviations**: These shorten the use of a word. Instead of using Mister Smith, we use Mr. Smith.
- **Acronyms**: These are used extensively in many fields including computer science. They use a combination of letters for phrases such as FORmula TRANslation for FORTRAN. They can be recursive such as GNU. Of course, the one we will continue to use is NLP.
- **Contractions**: We'll find these useful for commonly used combinations of words such as the first word of this sentence.
- **Numbers**: A specialized word that normally uses only digits. However, more complex versions can include a period and a special character to reflect scientific notation or numbers of a specific base.

Identifying these parts is useful for other NLP tasks. For example, to determine the boundaries of a sentence, it is necessary to break it apart and determine which elements terminate a sentence.

The process of breaking text apart is called tokenization. The result is a stream of tokens. The elements of the text that determine where elements should be split are called Delimiters. For most English text, whitespace is used as a delimiter. This type of a delimiter typically includes blanks, tabs, and new line characters.
Tokenization can be simple or complex. Here, we will demonstrate a simple tokenization using the String class’ split method. First, declare a string to hold the text that is to be tokenized:

```java
String text = "Mr. Smith went to 123 Washington avenue.";
```

The split method uses a regular expression argument to specify how the text should be split. In the next code sequence, its argument is the string \s+. This specifies that one or more whitespaces be used as the delimiter:

```java
String tokens[] = text.split("\s+");
```

A for-each statement is used to display the resulting tokens:

```java
for(String token : tokens) {
    System.out.println(token);
}
```

When executed, the output will appear as shown here:

Mr.
Smith
went
to
123
Washington
avenue.

In Chapter 2, Finding Parts of Text, we will explore the tokenization process in depth.

### Finding sentences

We tend to think of the process of identifying sentences as a simple process. In English, we look for termination characters such as a period, question mark, or exclamation mark. However, as we will see in Chapter 3, Finding Sentences, this is not always that simple. Factors that make it more difficult to find the end of sentences include the use of embedded periods in such phrases as "Dr. Smith" or "204 SW. Park Street".

This process is also called **Sentence Boundary Disambiguation (SBD)**. This is a more significant problem in English than it is in languages such as Chinese or Japanese that have unambiguous sentence delimiters.
Identifying sentences is useful for a number of reasons. Some NLP tasks, such as POS tagging and entity extraction, work on individual sentences. Question-answering applications also need to identify individual sentences. For these processes to work correctly, sentence boundaries must be determined correctly.

The following example demonstrates how sentences can be found using the Stanford DocumentPreprocessor class. This class will generate a list of sentences based on either simple text or an XML document. The class implements the Iterable interface allowing it to be easily used in a for-each statement.

Start by declaring a string containing the sentences, as shown here:

```java
String paragraph = "The first sentence. The second sentence.";
```

Create a StringReader object based on the string. This class supports simple read type methods and is used as the argument of the DocumentPreprocessor constructor:

```java
Reader reader = new StringReader(paragraph);
DocumentPreprocessor documentPreprocessor = new DocumentPreprocessor(reader);
```

The DocumentPreprocessor object will now hold the sentences of the paragraph. In the next statement, a list of strings is created and is used to hold the sentences found:

```java
List<String> sentenceList = new LinkedList<String>();
```

Each element of the documentPreprocessor object is then processed and consists of a list of the HasWord objects, as shown in the following block of code. The HasWord elements are objects that represent a word. An instance of StringBuilder is used to construct the sentence with each element of the hasWordList element being added to the list. When the sentence has been built, it is added to the sentenceList list:

```java
for (List<HasWord> element : documentPreprocessor) {
    StringBuilder sentence = new StringBuilder();
    List<HasWord> hasWordList = element;
    for (HasWord token : hasWordList) {
        sentence.append(token).append(" ");
    }
    sentenceList.add(sentence.toString());
}
```

A for-each statement is then used to display the sentences:

```java
for (String sentence : sentenceList) {
    System.out.println(sentence);
}
```
The output will appear as shown here:

The first sentence.
The second sentence.

The SBD process is covered in depth in Chapter 3, Finding Sentences.

Finding people and things

Search engines do a pretty good job of meeting the needs of most users. People frequently use a search engine to find the address of a business or movie show times. A word processor can perform a simple search to locate a specific word or phrase in a text. However, this task can get more complicated when we need to consider other factors such as whether synonyms should be used or if we are interested in finding things closely related to a topic.

For example, let's say we visit a website because we are interested in buying a new laptop. After all, who doesn't need a new laptop? When you go to the site, a search engine will be used to find laptops that possess the features you are looking for. The search is frequently conducted based on previous analysis of vendor information. This analysis often requires text to be processed in order to derive useful information that can eventually be presented to a customer.

The presentation may be in the form of facets. These are normally displayed on the left-hand side of a web page. For example, the facets for laptops might include categories such as an Ultrabook, Chromebook, or hard disk size. This is illustrated in the following figure, which is part of an Amazon web page:
Some searches can be very simple. For example, the `String` class and related classes have methods such as the `indexOf` and `lastIndexOf` methods that can find the occurrence of a `String` class. In the simple example that follows, the index of the occurrence of the target string is returned by the `indexOf` method:

```java
String text = "Mr. Smith went to 123 Washington avenue.";
String target = "Washington";
int index = text.indexOf(target);
System.out.println(index);
```

The output of this sequence is shown here:

```
22
```

This approach is useful for only the simplest problems.

When text is searched, a common technique is to use a data structure called an inverted index. This process involves tokenizing the text and identifying terms of interest in the text along with their position. The terms and their positions are then stored in the inverted index. When a search is made for the term, it is looked up in the inverted index and the positional information is retrieved. This is faster than searching for the term in the document each time it is needed. This data structure is used frequently in databases, information retrieval systems, and search engines.

More sophisticated searches might involve responding to queries such as: "Where are good restaurants in Boston?" To answer this query we might need to perform entity recognition/resolution to identify the significant terms in the query, perform semantic analysis to determine the meaning of the query, search and then rank candidate responses.

To illustrate the process of finding names, we use a combination of a tokenizer and the OpenNLP `TokenNameFinderModel` class to find names in a text. Since this technique may throw an `IOException`, we will use a `try-catch` block to handle it. Declare this block and an array of strings holding the sentences, as shown here:

```java
try {
    String[] sentences = {
        "Tim was a good neighbor. Perhaps not as good a Bob " +
        "Haywood, but still pretty good. Of course Mr. Adam " +
        "took the cake!"};
    // Insert code to find the names here
} catch (IOException ex) {
    ex.printStackTrace();
}
```
Before the sentences can be processed, we need to tokenize the text. Set up the tokenizer using the `Tokenizer` class, as shown here:

```java
Tokenizer tokenizer = SimpleTokenizer.INSTANCE;
```

We will need to use a model to detect sentences. This is needed to avoid grouping terms that may span sentence boundaries. We will use the `TokenNameFinderModel` class based on the model found in the `en-ner-person.bin` file. An instance of `TokenNameFinderModel` is created from this file as follows:

```java
TokenNameFinderModel model = new TokenNameFinderModel(
        new File("C:\OpenNLP Models", "en-ner-person.bin"));
```

The `NameFinderME` class will perform the actual task of finding the name. An instance of this class is created using the `TokenNameFinderModel` instance, as shown here:

```java
NameFinderME finder = new NameFinderME(model);
```

Use a for-each statement to process each sentence as shown in the following code sequence. The `tokenize` method will split the sentence into tokens and the `find` method returns an array of `Span` objects. These objects store the starting and ending indexes for the names identified by the `find` method:

```java
for (String sentence : sentences) {
    String[] tokens = tokenizer.tokenize(sentence);
    Span[] nameSpans = finder.find(tokens);
    System.out.println(Arrays.toString(
            Span.spansToStrings(nameSpans, tokens)));
}
```

When executed, it will generate the following output:

```
[Tim, Bob Haywood, Adam]
```

The primary focus of *Chapter 4, Finding People and Things*, is name recognition.

## Detecting Parts of Speech

Another way of classifying the parts of text is at the sentence level. A sentence can be decomposed into individual words or combinations of words according to categories, such as nouns, verbs, adverbs, and prepositions. Most of us learned how to do this in school. We also learned not to end a sentence with a preposition contrary to what we did in the second sentence of this paragraph.
Detecting the Parts of Speech (POS) is useful in other tasks such as extracting relationships and determining the meaning of text. Determining these relationships is called Parsing. POS processing is useful for enhancing the quality of data sent to other elements of a pipeline.

The internals of a POS process can be complex. Fortunately, most of the complexity is hidden from us and encapsulated in classes and methods. We will use a couple of OpenNLP classes to illustrate this process. We will need a model to detect the POS. The POSModel class will be used and instanced using the model found in the en-pos-maxent.bin file, as shown here:

```java
POSModel model = new POSModelLoader().load(
    new File("../OpenNLP Models/" "en-pos-maxent.bin"));
```

The POSTaggerME class is used to perform the actual tagging. Create an instance of this class based on the previous model as shown here:

```java
POSTaggerME tagger = new POSTaggerME(model);
```

Next, declare a string containing the text to be processed:

```java
String sentence = "POS processing is useful for enhancing the " + "quality of data sent to other elements of a pipeline.";
```

Here, we will use a whitespace tokenizer to tokenize the text:

```java
String tokens[] = WhitespaceTokenizer.INSTANCE.tokenize(sentence);
```

The tag method is then used to find those parts of speech, which stored the results in an array of strings:

```java
String[] tags = tagger.tag(tokens);
```

The tokens and their corresponding tags are then displayed:

```java
for(int i=0; i<tokens.length; i++) {
    System.out.print(tokens[i] + "[" + tags[i] + "] ");
}
```

When executed, the following output will be produced:

```
```

Each token is followed by an abbreviation, contained within brackets, for its part of speech. For example, NNP means that it is a proper noun. These abbreviations will be covered in Chapter 5, Detecting Parts of Speech, which is devoted to exploring this topic in depth.
Introduction to NLP

Classifying text and documents
Classification is concerned with assigning labels to information found in text or documents. These labels may or may not be known when the process occurs. When labels are known, the process is called classification. When the labels are unknown, the process is called clustering.

Also of interest in NLP is the process of categorization. This is the process of assigning some text element into one of the several possible groups. For example, military aircraft can be categorized as either fighter, bomber, surveillance, transport, or rescue.

Classifiers can be organized by the type of output they produce. This can be binary, which results in a yes/no output. This type is often used to support spam filters. Other types will result in multiple possible categories.

Classification is more of a process than many of the other NLP tasks. It involves the steps that we will discuss in Understanding NLP models later in the chapter. Due to the length of this process, we will not illustrate the process here. In Chapter 6, Classifying Text and Documents, we will investigate the classification process and provide a detailed example.

Extracting relationships
Relationship extraction identifies relationships that exist in text. For example, with the sentence "The meaning and purpose of life is plain to see", we know that the topic of the sentence is "The meaning and purpose of life". It is related to the last phrase that suggests that it is "plain to see".

Humans can do a pretty good job at determining how things are related to each other, at least at a high level. Determining deep relationships can be more difficult. Using a computer to extract relationships can also be challenging. However, computers can process large datasets to find relationships that would not be obvious to a human or that could not be done in a reasonable period of time.

There are numerous relationships possible. These include relationships such as where something is located, how two people are related to each other, what are the parts of a system, and who is in charge. Relationship extraction is useful for a number of tasks including building knowledge bases, performing analysis of trends, gathering intelligence, and performing product searches. Finding relationships is sometimes called Text Analytics.
There are several techniques that we can use to perform relationship extractions. These are covered in more detail in Chapter 7, Using a Parser to Extract Relationships. Here, we will illustrate one technique to identify relationships within a sentence using the Stanford NLP StanfordCoreNLP class. This class supports a pipeline where annotators are specified and applied to text. Annotators can be thought of as operations to be performed. When an instance of the class is created, the annotators are added using a Properties object found in the java.util package.

First, create an instance of the Properties class. Then assign the annotators as follows:

```java
Properties properties = new Properties();
properties.put("annotators", "tokenize, ssplit, parse");
```

We used three annotators, which specify the operations to be performed. In this case, these are the minimum required to parse the text. The first one, tokenize, will tokenize the text. The ssplit annotator splits the tokens into sentences. The last annotator, parse, performs the syntactic analysis, parsing, of the text.

Next, create an instance of the StanfordCoreNLP class using the properties' reference variable:

```java
StanfordCoreNLP pipeline = new StanfordCoreNLP(properties);
```

Next, an Annotation instance is created, which uses the text as its argument:

```java
Annotation annotation = new Annotation("The meaning and purpose of life is plain to see.");
```

Apply the annotate method against the pipeline object to process the annotation object. Finally, use the prettyPrint method to display the result of the processing:

```java
pipeline.annotate(annotation);
pipeline.prettyPrint(annotation, System.out);
```

The output of this code is shown as follows:

```
Sentence #1 (11 tokens):
The meaning and purpose of life is plain to see.
```
Introduction to NLP

[Text=The CharacterOffsetBegin=0 CharacterOffsetEnd=3 PartOfSpeech=DT] [Text=meaning CharacterOffsetBegin=4 CharacterOffsetEnd=11 PartOfSpeech=NN] [Text=and CharacterOffsetBegin=12 CharacterOffsetEnd=15 PartOfSpeech=CC] [Text=purpose CharacterOffsetBegin=16 CharacterOffsetEnd=23 PartOfSpeech=NN] [Text=of CharacterOffsetBegin=24 CharacterOffsetEnd=26 PartOfSpeech=IN] [Text=life CharacterOffsetBegin=27 CharacterOffsetEnd=31 PartOfSpeech=NN] [Text=is CharacterOffsetBegin=32 CharacterOffsetEnd=34 PartOfSpeech=VBZ] [Text=plain CharacterOffsetBegin=35 CharacterOffsetEnd=40 PartOfSpeech=JJ] [Text=to CharacterOffsetBegin=41 CharacterOffsetEnd=43 PartOfSpeech=TO] [Text=see CharacterOffsetBegin=44 CharacterOffsetEnd=47 PartOfSpeech=VB] [Text=. CharacterOffsetBegin=48 CharacterOffsetEnd=48 PartOfSpeech=.] (ROOT (S (NP (NP (DT The) (NN meaning) (CC and) (NN purpose))) (PP (IN of) (NP (NN life)))) (VP (VBZ is) (ADJP (JJ plain) (S (VP (TO to) (VP (VB see))))))) (. .))) root(ROOT-0, plain-8) det(meaning-2, The-1) nsubj(plain-8, meaning-2) conj_and(meaning-2, purpose-4) prep_of(meaning-2, life-6) cop(plain-8, is-7) aux(see-10, to-9) xcomp(plain-8, see-10)
The first part of the output displays the text along with the tokens and POS. This is followed by a tree-like structure showing the organization of the sentence. The last part shows relationships between the elements at a grammatical level. Consider the following example:

 prep_of(meaning-2, life-6) 

This shows how the preposition, "of", is used to relate the words "meaning" and "life". This information is useful for many text simplification tasks.

Using combined approaches
As suggested earlier, NLP problems often involve using more than one basic NLP task. These are frequently combined in a pipeline to obtain the desired results. We saw one use of a pipeline in the previous section, Extracting relationships.

Most NLP solutions will use pipelines. We will provide several examples of pipelines in Chapter 8, Combined Approaches.

Understanding NLP models
Regardless of the NLP task being performed or the NLP tool set being used, there are several steps that they all have in common. In this section, we will present these steps. As you go through the chapters and techniques presented in this book, you will see these steps repeated with slight variations. Getting a good understanding of them now will ease the task of learning the techniques.

The basic steps include:

- Identifying the task
- Selecting a model
- Building and training the model
- Verifying the model
- Using the model

We will discuss each of these tasks in the following sections.
Identifying the task

It is important to understand the problem that needs to be solved. Based on this understanding, a solution can be devised that consists of a series of steps. Each of these steps will use an NLP task.

For example, suppose we want to answer a query such as "Who is the mayor of Paris?" We will need to parse the query into the POS, determine the nature of the question, the qualifying elements of the question, and eventually use a repository of knowledge, created using other NLP tasks, to answer the question.

Other problems may not be quite as involved. We might only need to break apart text into components so that the text can be associated with a category. For example, a vendor's product description may be analyzed to determine the potential product categories. The analysis of the description of a car would allow it to be placed into categories such as sedan, sports car, SUV, or compact.

Once you have an idea of what NLP tasks are available, you will be better able to match it with the problem you are trying to solve.

Selecting a model

Many of the tasks that we will examine are based on models. For example, if we need to split a document into sentences, we need an algorithm to do this. However, even the best sentence boundary detection techniques have problems doing this correctly every time. This has resulted in the development of models that examine the elements of text and then use this information to determine where sentence breaks occur.

The right model can be dependent on the nature of the text being processed. A model that does well for determining the end of sentences for historical documents might not work well when applied to medical text.

Many models have been created that we can use for the NLP task at hand. Based on the problem that needs to be solved, we can make informed decisions as to which model is the best. In some situations, we might need to train a new model. These decisions frequently involve trade-offs between accuracy and speed. Understanding the problem domain and the required quality of results permits us to select the appropriate model.
Building and training the model

Training a model is the process of executing an algorithm against a set of data, formulating the model, and then verifying the model. We may encounter situations where the text that needs to be processed is significantly different from what we have seen and used before. For example, using models trained using journalistic text might not work well when processing tweets. This may mean that the existing models will not work well with this new data. When this situation arises, we will need to train a new model.

To train a model, we will often use data that has been "marked up" in such a way that we know the correct answer. For example, if we are dealing with POS tagging, then the data will have POS elements (such as nouns and verbs) marked in the data. When the model is being trained, it will use this information to create the model. This dataset is called a corpus.

Verifying the model

Once the model has been created, we need to verify it against a sample set. The typical verification approach is to use a sample set where the correct responses are known. When the model is used with this data, we are able to compare its result to the known good results and assess the quality of the model. Often, only part of a corpus is used for training while the other part is used for verification.

Using the model

Using the model is simply applying the model to the problem at hand. The details are dependent on the model being used. This was illustrated in several of the earlier demonstrations, such as in the Detecting Parts of Speech section where we used the POS model as contained in the en-pos-maxent.bin file.

Preparing data

An important step in NLP is finding and preparing data for processing. This includes data for training purposes and the data that needs to be processed. There are several factors that need to be considered. Here, we will focus on the support Java provides for working with characters.

We need to consider how characters are represented. Although we will deal primarily with English text, other languages present unique problems. Not only are there differences in how a character can be encoded, the order in which text is read will vary. For example, Japanese orders its text in columns going from right to left.
There are also a number of possible encodings. These include ASCII, Latin, and Unicode to mention a few. A more complete list is found in the following table. Unicode, in particular, is a complex and extensive encoding scheme:

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>A character encoding using 128 (0-127) values.</td>
</tr>
<tr>
<td>Latin</td>
<td>There are several Latin variations that uses 256 values. They include various combination of the umlaut, such as ã, and other characters. Various versions of Latin have been introduced to address various Indo-European languages, such as Turkish and Esperanto.</td>
</tr>
<tr>
<td>Big5</td>
<td>A two-byte encoding to address the Chinese character set.</td>
</tr>
<tr>
<td>Unicode</td>
<td>There are three encodings for Unicode: UTF-8, UTF-16, and UTF-32. These use 1, 2, and 4 bytes, respectively. This encoding is able to represent all known languages in existence today, including newer languages such as Klingon and Elvish.</td>
</tr>
</tbody>
</table>

Java is capable of handling these encoding schemes. The `javac` executable's `–encoding` command-line option is used to specify the encoding scheme to use. In the following command line, the Big5 encoding scheme is specified:

```
javac –encoding Big5
```

Character processing is supported using the primitive data type `char`, the `Character` class, and several other classes and interfaces as summarized in the following table:

<table>
<thead>
<tr>
<th>Character type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Primitive data type.</td>
</tr>
<tr>
<td>Character</td>
<td>Wrapper class for char.</td>
</tr>
<tr>
<td>CharBuffer</td>
<td>This class support a buffer of char providing methods for get/put characters or a sequence of characters operations.</td>
</tr>
<tr>
<td>CharSequence</td>
<td>An interface implemented by CharBuffer, Segment, String, StringBuffer and StringBuilder. It supports read-only access to a sequence of chars.</td>
</tr>
</tbody>
</table>

Java also provides a number of classes and interfaces to support strings. These are summarized in the following table. We will use these in many of our examples. The `String`, `StringBuilder`, and `StringBuffer` classes provide similar string processing capabilities but differ in whether they can be modified and whether they are thread-safe. The `CharacterIterator` interface and the `StringCharacterIterator` class provide techniques to traverse character sequences.
The `Segment` class represents a fragment of text.

<table>
<thead>
<tr>
<th>Class/Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>An immutable string.</td>
</tr>
<tr>
<td>StringBuffer</td>
<td>Represents a modifiable string. It is thread-safe.</td>
</tr>
<tr>
<td>StringBuilder</td>
<td>Compatible with the <code>StringBuffer</code> class but is not thread-safe.</td>
</tr>
<tr>
<td>Segment</td>
<td>Represents a fragment of text in a character array. It provides rapid access to character data in an array.</td>
</tr>
<tr>
<td>CharacterIterator</td>
<td>Defines an iterator for text. It supports bidirectional traversal of text.</td>
</tr>
<tr>
<td>StringCharacterIterator</td>
<td>A class that implements the <code>CharacterIterator</code> interface for a <code>String</code>.</td>
</tr>
</tbody>
</table>

We also need to consider the file format if we are reading from a file. Often data is obtained from sources where the words are annotated. For example, if we use a web page as the source of text, we will find that it is marked up with HTML tags. These are not necessarily relevant to the analysis process and may need to be removed.

The **Multi-Purpose Internet Mail Extensions (MIME)** type is used to characterize the format used by a file. Common file types are listed in the following table. Either we need to explicitly remove or alter the markup found in a file or use specialized software to deal with it. Some of the NLP APIs provide tools to deal with specialized file formats.

<table>
<thead>
<tr>
<th>File format</th>
<th>MIME type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>plain/text</td>
<td>Simple text file</td>
</tr>
<tr>
<td>Office Type Document</td>
<td>application/msword</td>
<td>Microsoft Office</td>
</tr>
<tr>
<td></td>
<td>application/vnd.oasis.opendocument.text</td>
<td>Open Office</td>
</tr>
<tr>
<td>PDF</td>
<td>application/pdf</td>
<td>Adobe Portable Document Format</td>
</tr>
<tr>
<td>HTML</td>
<td>text/html</td>
<td>Web pages</td>
</tr>
<tr>
<td>XML</td>
<td>text/xml</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>Database</td>
<td>Not applicable</td>
<td>Data can be in a number of different formats</td>
</tr>
</tbody>
</table>

Many of the NLP APIs assume that the data is clean. When it is not, it needs to be cleaned lest we get unreliable and misleading results.
Summary
In this chapter we introduced NLP and its uses. We found that it is used in many places to solve many different types of problems ranging from simple searches to sophisticated classification problems. The Java support for NLP in terms of core string support and advanced NLP libraries were presented. The basic NLP tasks were explained and illustrated using code. We also examined the process of training, verifying, and using models.

In this book, we will lay the foundation for using the basic NLP tasks using both simple and more sophisticated approaches. You may find that some problems require only simple approaches and when that is the case, knowing how to use the simple techniques may be more than adequate. In other situations, a more complex technique may be needed. In either case, you will be prepared to identify what tool is needed and be able to choose the appropriate technique for the task.

In the next chapter, we will examine the process of tokenization in depth and see how it can be used to find parts of text.
Where to buy this book
You can buy Natural Language Processing with Java from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers. Click here for ordering and shipping details.