NATURAL LANGUAGE PROCESSING

SUCCINCTLY

BY JOSEPH D. BOOTH

SUCCINCTLY EBOOK SERIES

Syncfusion®

Natural Language Processing Succinctly

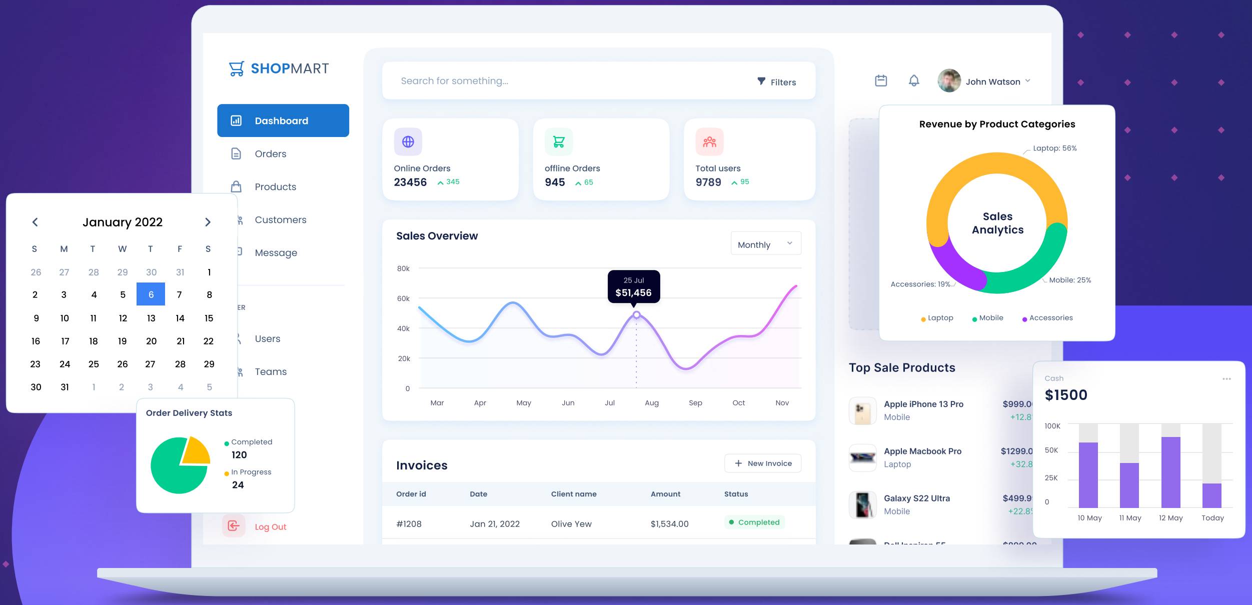
By
Joseph D. Booth

Foreword by Daniel Jebaraj

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.
Proofreader: Darren West, content producer, Syncfusion, Inc.

THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING

POWERFUL APPS



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

4.6 out of 5 stars

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

IBM SIEMENS intel VISA

Syncfusion

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Face-book to help us spread the word about the *Succinctly* series!

About the Author

Joseph D. Booth has been programming since 1981 in a variety of languages, including BASIC, Clipper, FoxPro, Delphi, Classic ASP, Visual Basic, Visual C#, and the .NET Framework. He has also worked in various database platforms, including DBASE, Paradox, Oracle, and SQL Server.

He is the author of GitHub Succinctly, Accounting Succinctly, Regular Expressions Succinctly, and Visual Studio Add-Ins Succinctly from Syncfusion, as well as six books on Clipper and FoxPro programming, network programming, and client/server development with Delphi. He has also written several third-party developer tools, including CLIPWKS, which allows developers to programmatically create and read native Lotus and Excel spreadsheet files from Clipper applications.

Joe has worked for a number of companies including Sperry Univac, MCI-WorldCom, Ronin, Harris Interactive, Thomas Jefferson University, People Metrics, and Investor Force. He is one of the primary authors of Results for Research (market research software), PEPSys (industrial distribution software), and a key contributor to AccuBuild (accounting software for the construction industry).

He has a background in accounting, having worked as a controller for several years in the industrial distribution field, but his real passion is computer programming.

In his spare time, Joe is an avid tennis player, practices yoga and martial arts, and plays with his first granddaughter, Blaire.

Chapter 1 Natural Language Processing

In Star Trek, the actors frequently talk to the computer. The computer understands their requests and immediately delivers the expected results. While this level of understanding is still quite a way off, it is one of the goals of Artificial Intelligence—for the computer to accurately understand English (and other languages) and be able to extract meaning from the words.

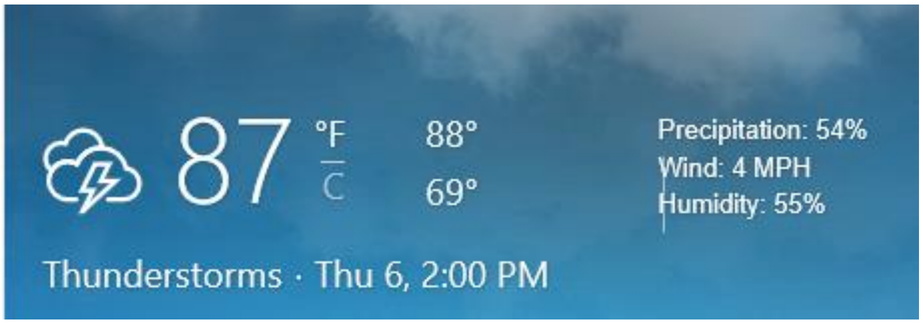
Siri, Cortana, and other "assistants"

Many smart phones now come with virtual assistants. For example, you might ask Siri, "Will I need a raincoat tomorrow?" The program will determine that your question is asking "Is it going to rain on the next calendar day at my location," or in programming pseudo-code:

```
return RainCheck(zipCode,date()+1)
```

The assistant software will determine the probability of rain, and based on that probability, return an answer, such as "There is a good chance you will need your raincoat." I asked Cortana that same question and was shown the weather chart in Figure 1.

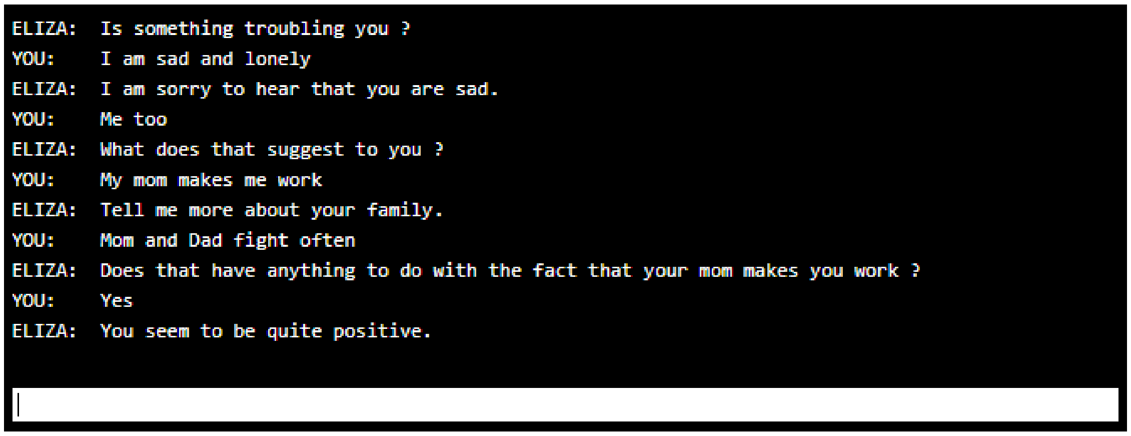
Figure 1 – Raincoat tomorrow?



Eliza

Back in 1966, Joseph Weizenbaum wrote a program called ELIZA, which was meant to simulate a Rogerian psychotherapist. A sample "session" is shown in Figure 2.

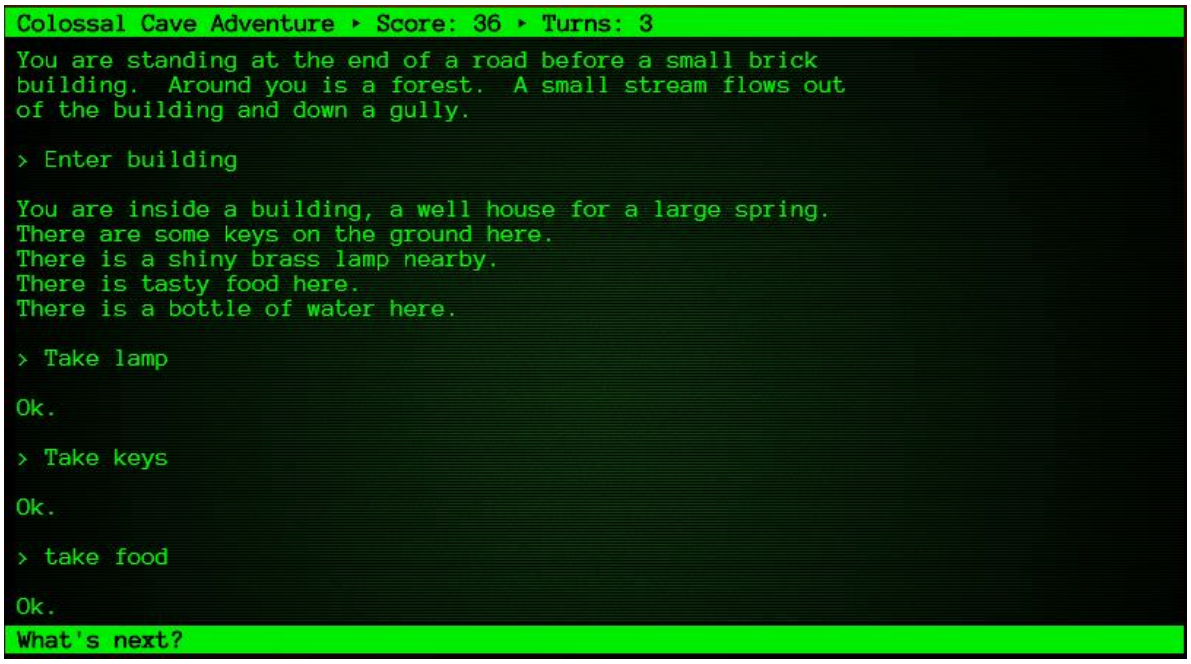
Figure 2 – Eliza session



```
ELIZA:  Is something troubling you ?
YOU:    I am sad and lonely
ELIZA:  I am sorry to hear that you are sad.
YOU:    Me too
ELIZA:  What does that suggest to you ?
YOU:    My mom makes me work
ELIZA:  Tell me more about your family.
YOU:    Mom and Dad fight often
ELIZA:  Does that have anything to do with the fact that your mom makes you work ?
YOU:    Yes
ELIZA:  You seem to be quite positive.
```

Many early computer games were influenced by ELIZA, allowing the user to enter short sentences instructing the program what to do for the next step. “Colossal Cave” was a very early computer game that allowed you to explore a “nearby cave” for treasures. It was entirely text based; you would type in a one or two-word commands to “explore” the world. Figure 3 shows the start of the program.

Figure 3 – Colossal Cave



```
Colossal Cave Adventure ‣ Score: 36 ‣ Turns: 3
You are standing at the end of a road before a small brick
building.  Around you is a forest.  A small stream flows out
of the building and down a gully.

> Enter building

You are inside a building, a well house for a large spring.
There are some keys on the ground here.
There is a shiny brass lamp nearby.
There is tasty food here.
There is a bottle of water here.

> Take lamp

Ok.

> Take keys

Ok.

> take food

Ok.
What's next?
```

Eliza, Colossal Cave, and similar games had no understanding, but were simply clever rule-based programming to “simulate” understanding. However, they did show the potential of using Natural Language as an interaction method.

SHRDLU

SHRDLU (named after the seventh through twelfth most common letters in English) was written in 1970 by Terry Winograd, and showed some very impressive Natural Language Processing capabilities, within a controlled environment of a block word. Figure 4 shows a sample dialog with the program.

Figure 4 – SHRDLU dialog

Person: Pick up a big red block.
Computer: OK.
Person: Grasp the pyramid.
Computer: I DON'T UNDERSTAND WHICH PYRAMID YOU MEAN.
Person: Find a block which is taller than the one you are holding and put it into the box.
Computer: BY "IT", I ASSUME YOU MEAN THE BLOCK WHICH IS TALLER THAN THE ONE I AM HOLDING.
Computer: OK.
Person: What does the box contain?
Computer: THE BLUE PYRAMID AND THE BLUE BLOCK.

The level of understanding was quite impressive, but limited to the blocks in the virtual world. While the program caused a lot of optimism, researchers began to realize just how complex modeling the real world could be. While parsing text and understanding it is a big part of NLP, building a model of all known facts is an incredibly complex task. By limiting the domain to a reasonable size, Natural Language could be used to help systems, but a computer program that understands all the nuances and complexities of the real world, and can answer any questions, is still quite a way off.

Search engines

Internet search engines, such as Google, Microsoft Bing, Yahoo, and Duck Duck Go all attempt to interpret the meaning behind your search text. However, it is curious to see how well the "questions" get answered. For example, for soccer fans, you might ask the question, "Who won last night's match?" The result could be a list of lottery game winners or multiple sporting event outcomes.

Part of the reason that companies are collecting information whenever they can about you, is to to give better answers when you search. I play a lot of tennis, so when I ask the question, "Who won last night's match?" and it is during the US Open dates, I'd really love to see the result of the tennis matches. You can see this level of personalization with advertisements, particularly on social media. If you purchase a product from Amazon, your next visit to Facebook will very likely show you ads related to your purchase. The more a computer system knows about you, the better your search results can be. Ask Google to "show me restaurants near me," and it will understand (based on the IP address of your computer) what "near me" means.

Chat bots

A chat bot is a computer program that attempts to respond to questions asked by humans. In small domains, chat bots can be very helpful, solving common problems. Many businesses rely on chat bot technology to handle simple requests, like fixing a router, or finding out what is on TV for a subscriber. Chat bots generally rely on patterns, and they attempt to provide a response if they detect that a question meets the pattern.

ALICE (Artificial Linguistic Internet Computer Entity) is an application based on a markup language called AIML (Artificial Intelligence Markup Language). This language is a structured text file that provides patterns and expected responses. Listing 1 shows a simple AIML file that looks for a pattern and provides a template response.

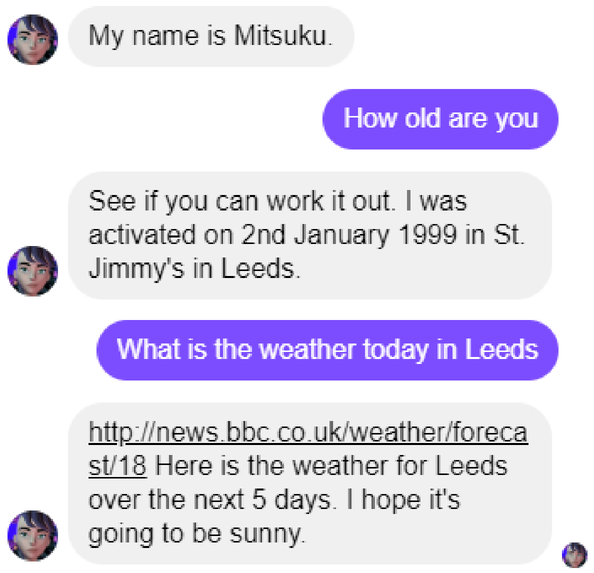
Listing 1 – Simple AIML

```
<aiml version = "1.0.1" encoding = "UTF-8"?>

   <category>

      <pattern> HELLO * </pattern>

      <template>

         Hello There!

      </template>

   </category>

</aiml>
```

This file simply says: if the user enters "HELLO" followed by any text, respond with the template response of "Hello There!" There are many additional features, such allowing the response to be one of several random templates, imbedding prior responses, etc.

By keeping the conversation space small, it is possible to create enough patterns and template responses. There is a chat bot called Mitsuku that uses AIML files to converse as an 18-year-old woman from Leeds. You can visit the Mitsuku website to see a sample chat bot in action. Figure 5 shows a sample chat with the bot.

Figure 5 – Sample chat bot


My name is Mitsuku.

How old are you

See if you can work it out. I was activated on 2nd January 1999 in St. Jimmy's in Leeds.

What is the weather today in Leeds

http://news.bbc.co.uk/weather/forecast/18 Here is the weather for Leeds over the next 5 days. I hope it's going to be sunny.

Turing test

Allan Turing, the brilliant cryptologist from World War II, suggested a “test” that could be performed to indicate if a computer has reached a certain level of intelligence. The test, named after him, simply allows people to interact via a computer keyboard with an unknown person or computer on the other side. If a computer manages to convince 30 percent of the people interacting with it that it is a human being, the test is considered passed. Note that he was not trying to determine whether the computer could "think" or not, but rather, if the computer could respond with human-like conversation.

Loebner Prize

The Loebner Prize is a competition to see if any computer software can pass the Turing test. The gold medal (and $100,000) will be awarded for a program that can pass the test using visual and audio components, while the silver medal ($25,000) is for a program that passes the test using text-only messages. The bronze medal is awarded to the most human-like program. As of the writing of this book, the gold and silver medals have never been awarded.

Why is English so hard?

English is a notoriously difficult language to learn because it has so many ambiguities and subtleties. Many words have multiple uses, and there are different ways to express the same sentence. Often, we figure out the meaning from context and other clues, which are very difficult to resolve in a computer. A few complexities of English are described in the following paragraphs.

Context changes meaning

Consider the sentence "He shot an eagle." To a game warden, this is bad news, perhaps a poacher to deal with. However, to a golfer, this is good news, since an eagle is two strokes under par.

Sarcasm

Another issue is that of sarcasm. If a reviewer writes "I really enjoyed the loud noise in the theater," a computer program might interpret that as a positive comment. A person's background knowledge that theaters are generally quiet would make it clear that the reviewer is making a negative remark.

Exceptions

In English, there are three general rules for making a word plural case.

- Add the letter **s**
- If the word ends in **y**, change the **y** to **ies**
- If the word ends in an "s" sound, add **es** to the word

However, there are many exceptions to the rules. Some words are the same whether singular or plural (sheep), other words change some letters (man/men), and others have specific plural forms (child/children)

Summary

The goal of this book is to describe the various components needed to create a system that appears to understand natural language and will provide reasonable responses to English questions. We will design a simple system to take question text and provide an answer from a specific set of data.

We will also cover some APIs from Microsoft, Cloudmersive, and Google that provide the various methods for an NLP application. With an understanding of the steps involved, you should be able to use one of the APIs to add natural language support to your application.

If your goal is to create Siri’s older, wiser sister, or pass the next Turing test, hopefully the descriptions and code presented in this book, and the available APIs, will give you a reasonable starting point.

Chapter 2 What we're building

The goal of the book is to build a simple NLP library, and use that library code as part of a question-answering NLP application. The complete source code to the library and the database we will be querying are available on the Syncfusion author's website at http://www.joebooth-consulting.com.

The first few chapters will discuss some of the key functions that are used to parse sentences into actionable structures to query a dataset. By the end of Chapter 6, you should be able to get a list of words and tags from a sentence. In Chapters 9–11, we will explore some API calls (Cloudmersive, Google, and Microsoft) that you can use to get a tagged list of words. Some of those web services offer additional NLP tasks beyond our goal of question answering, so it is worth reading them to see what else NLP can do.

In Chapters 7 and 8, we show how to take a tagged list of words and use it to ask questions and get answers back, first by building the knowledge and code to access it, and then by showing how to match the questions to the appropriate function to provide an answer. If you are simply interested in the question-answering side, feel free to skip Chapters 3–6 and use one of the web services to build your tagged word list.

In the next chapters, we will also make use of another NLP web service product from Cloudmersive. I suggest registering for an API key with Cloudmersive; it allows you up to 50,000 requests per day on the free account. In this chapter, we will show how to get set up to make the API calls and use these calls in early chapters to supplement our code.

NLP (our main class library)

This class library project will hold the parsing and tagging routines, to create structured collection from the freeform text the user enters. In addition to the parser logic, the class library will also include the dictionary of words and phrases necessary to interpret the user's text. While our example code will have self-contained word and tag lists, we will also look at using web services to assist some of the functions. There are number of companies that offer APIs to supplement the data we store internally.

Tagger

The **`tagger`** static class within the NLP project contains the list of words and parts of speech we want to understand. We have a small list: approximately 500 top verbs, adverbs, and adjectives, as well as pronouns. You can create your own word list or rely on various web services and freely available dictionaries to help look up and interpret words you find in a sentence or question.

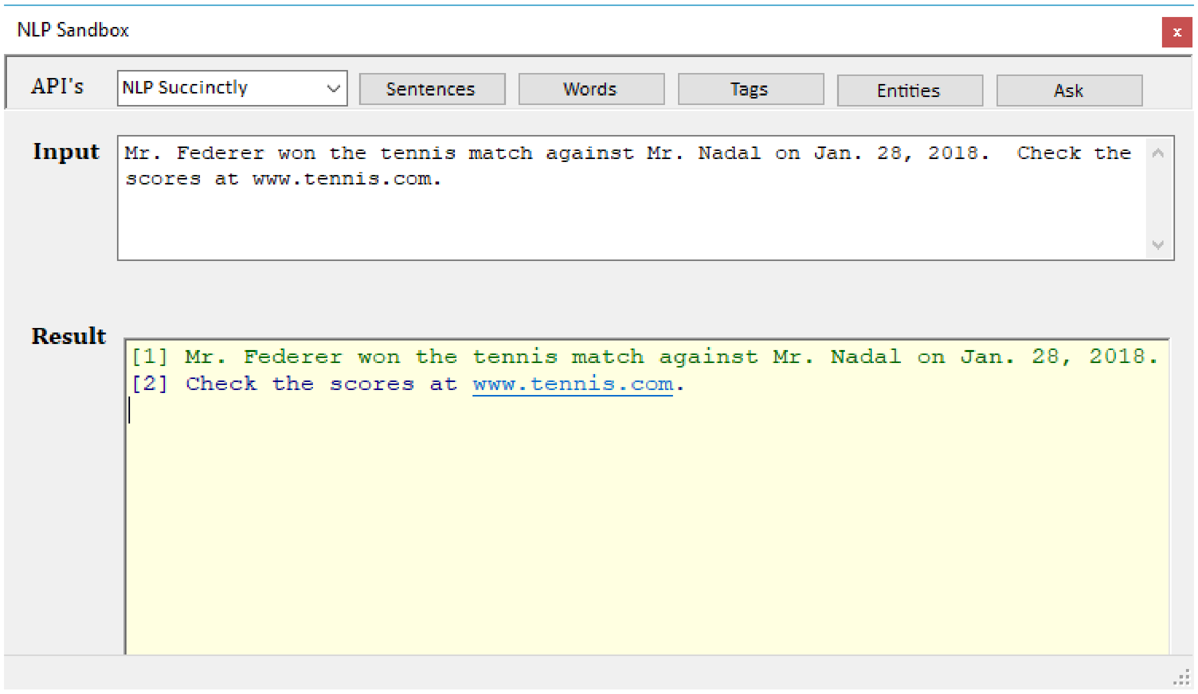
DataSet (sample dataset)

Our sample dataset is simply a collection of facts (in our example, tennis major tournaments) stored as a collection of objects. If your dataset is small and static, this approach could work. However, for a large or frequently changing dataset, you would most likely pull the data from a database backend using SQL queries.

Playground

Playground is a simple project that allows you to test out the API calls. It is a Windows application that allows you to enter some text, and then either parse it, or ask questions of the dataset. Figure 6 shows the Playground window.

Figure 6 – Windows NLP testing playground



```
Mr. Federer won the tennis match against Mr. Nadal on Jan. 28, 2018.  Check the
scores at www.tennis.com.
```

```
[1] Mr. Federer won the tennis match against Mr. Nadal on Jan. 28, 2018.
[2] Check the scores at www.tennis.com.
```

Web service

Cloudmersive is a web services company that offers several different web services to solve problems that often plague developers. These include OCR, data validation, and document conversion, and my favorite, the Natural Language API calls. In their own words:

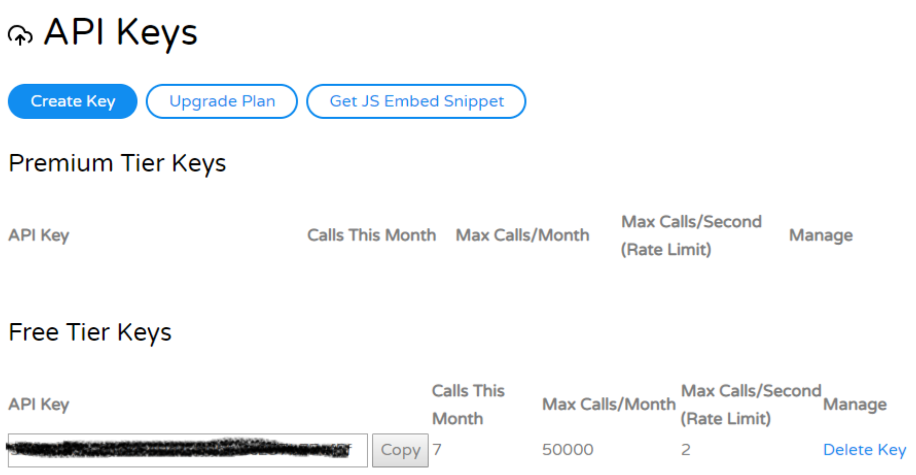
The Cloudmersive Natural Language Processing APIs let you perform part of speech tagging, entity identification, sentence parsing, language detection, text analytics, and much more to help you understand the meaning of unstructured text across a range of programming languages - Node.JS, Python, C#, Java, PHP, Objective-C, and Ruby.

Getting registered

To register with Cloudmersive (and get 50,000 free web service calls per day), go to this webpage and create a login account.

Once you've done that, next time you log in, you will be directed to the Management Center, where you can manage your API keys. You will need an API key to call the web services. Figure 7 shows the API Keys management page.

Figure 7 – API keys



API Keys

Premium Tier Keys

API Key	Calls This Month	Max Calls/Month	Max Calls/Second (Rate Limit)	Manage

Free Tier Keys

API Key	Calls This Month	Max Calls/Month	Max Calls/Second (Rate Limit)	Manage
[redacted] Copy	7	50000	2	Delete Key

Installing

Once you have an API key, you can use NuGet to install the package into your application.

```
Install-Package Cloudmersive.APIClient.NET.NLP -Version 1.2.3
```

You will need to add a few references:

```
using Cloudmersive.APIClient.NET.NLP.Api;
using Cloudmersive.APIClient.NET.NLP.Client;
using Cloudmersive.APIClient.NET.NLP.Model;
```

Now, let’s make our first test call.

What language is the message written in?

One useful feature is the ability to detect what language an input text string is in. Let's look at how the Cloudmersive web service provides that functionality. Listing 2 shows a sample API call.

Listing 2 – Language detection API

```
static public LanguageDetectionResponse DetectLanguage(string text)
{
    // Configure API key authorization: Apikey
    Configuration.Default.AddApiKey("Apikey", APIKey);

    var apiInstance = new LanguageDetectionApi();       // Which API to
call
    try
      {
         // Detect language of text
        LanguageDetectionResponse result =
                    apiInstance.LanguageDetectionPost(text);
         return result;
      }
  catch (Exception e)
      {
         return null;
      }
}
```

This is the general structure of the API calls. First, set your API key, and then create an instance of the API you want to call. Finally, make the call within a `try...catch` block (since you are `POST`ing to a web service) and return the result of `NULL`.

You can visit the website to explore the various API calls provided. For convenience, Chapter 9 has sample C# code to call the APIs from Cloudmersive for the internal NLP functions we discuss in the next few chapters.

Getting started

To have the computer appear to process and understand text, there are a few concepts we should be comfortable with before we begin.

Expected usage

Knowing what kind of data your application will work with and what kinds of inputs and questions the user is likely to use can go a long way towards making your application better able to figure out the question and answer. Siri and other "intelligent assistants" already know a lot about the user, simply by having access to the information in the phone or device. If you want to know tomorrow's weather, Siri can access the location information in your phone to determine the location you are asking about.

If you are adding Natural Language support to a personnel system, you will expect questions about employees, applicants, roles, etc. The question, "*Who is the president?"* would be answered with the company's president name, not the president of the United States. A scheduling system would answer questions about appointments, free time, etc. If I tell a scheduler system I want to play tennis with Roger tomorrow, I would hope it would find the Roger in my contacts, and not put me on the court with Roger Federer.

Knowing your expected usage and data allows you to make assumptions to help return the most likely response. English is highly ambiguous, so any context we can provide to help resolve the ambiguities will improve the appearance of understanding.

Domain size

It is overly ambitious to assume that we could create a massive knowledge base that would understand every conceivable fact in the world. But if we keep the knowledge domain small, such as vendors and products a business uses, or football teams and stats in the NFL, we could create a reasonable system to handle basic English-language queries.

Regular expressions

Many of the parsing techniques in the subsequent chapters will use *regular expressions* (or *regex*), which are patterns used to perform string searching. A regex is a compact string that indicates how data should be searched. It is very powerful, very cryptic, and very easy to get wrong. To get a sense of regular expressions, let's explore a simple example.

Imagine you wanted to find the word "CAT" in a text string. Pretty easy, right? The regular expression is simply **`CAT`**. However, your needs are a bit more complex: you need any three-letter word beginning with **`C`** and ending with **`T`**. You can use **`C.T`** (the period represents any character). But wait, that finds unexpected things, like **`C#T`**. No problem—change the expression to **`C[a-z]T`**. Only vowels allowed? Then let’s go with **`C[aeiou]T`**.

Regular expression can handle a lot, such as the following expression: **`^[0-9]{5}([- /]?[0-9]{4})?$.`** This expression asks for five numbers and an optional space or dash, optionally followed by four more digits (such as a United States Postal Service zip code).

Such a regular expression could read through a corpus of text and try to identify possible zip codes, phone numbers, email addresses, or URLs. A regular expression is looking for text patterns that match. The expressions can get very involved. The cryptic expression in Listing 3 shows one way to validate that text looks like a valid file name.

Listing 3 – File name regex

```
^(([a-zA-Z]:|\\)\\)?(((\.)|(\.\.)|([^\\/:\*\?"\|<>\. ](([^\\/:\*\?"\|<>\.
])|([^\\/:\*\?"\|<>]*[^\\/:\*\?"\|<>\. ]))?))\\)*[^\\/:\*\?"\|<>\.
](([^\\/:\*\?"\|<>\. ])|([^\\/:\*\?"\|<>]*[^\\/:\*\?"\|<>\. ]))?$
```

It is very powerful, but can be very cryptic and hard to read. You do not need to necessarily understand how regexes work, but you will see references to two key regex methods throughout the book.

Regex.Split()

This method works just like the regular string **`Split`** function, but uses a regular expression to split the string. For example, using **`StringSplit`**, we could use the following code to split by punctuation characters.

```
char[] EndChars = new char[] { '.', '!', '?', ';' };

string[] Sentences = text.Split(EndChars,

                     StringSplitOptions.RemoveEmptyEntries);
```

Using the **`Regex.Split`** function, we can perform the same functionality with the following code:

```
string RegexEndofSentence = @"[.!?;]";

string[] Sentences = Regex.Split(text, RegexEndofSentence);
```

You will see examples using **`Regex.Split`** in the code when the splitting criteria is a bit more complex than simple string splitting.

Regex.IsMatch()

This method compares the text with a regular expression pattern and returns a Boolean value whether the text matches the search rules in the regex expression. For example, a simple Regex to test a time string (such as 10:00) is:

```
@"^([0-1][0-9]|[2][0-3]):([0-5][0-9])$"
```

The following code snippet will set the **`IsTime`** flag to **`true`** if the value in the word string looks like a time value.

```
bool IsTime =(Regex.IsMatch(Word_,@"^([0-1][0-9]|[2][0-3]):([0-5][0-9])$"));
```

Summary

Know your application and keep your domain size small—this will help reduce the ambiguities and give you a better chance to get meaningful results from the inputted text.

If you want to roll your own parsing routines, you should spend a bit of time exploring the regex syntax, which is very useful for parsing text. You can download my book on regular expressions here.

Chapter 3 Extracting Sentences

At first glance, it would seem that breaking a text into individual sentences is a trivial task. Simply use the .NET **`Split()`** method, as shown in Listing 4.

Listing 4 – Simple split

```
char[] CharSep = new char[] { '.', '!','?',';' };
string[] Sentences = theText.Split(CharSep,
                                   StringSplitOptions.RemoveEmptyEntries);
```

The code says to add a new element to the string array as soon as a period, exclamation point, question mark, or semi-colon is found. If an element is empty, don't include it in the result.

However, English has its own set of punctuation rules, and the period character is used for a lot more than just an end-of-sentence indicator. Some uses of the period character include:

- Abbreviations (Mr., Mrs.)
- Acronyms (I.B.M., I.D. card)
- Currency ($32.55)
- Website URLs (www.cnn.com)
- IP Addresses (127.0.0.1)
- Ellipsis

In addition, the English rules of grammar state that if an acronym or abbreviation ends a sentence, you should not add a second period.

In NLP parlance, the problem of extracting sentences from text is called *sentence boundary disambiguation*. There are multiple approaches to the problem, and we are going to explore two of them in this chapter.

Imagine that we needed to break the following text into sentences.

```
Mr. Federer won the tennis match against Mr. Nadal on Jan. 28, 2018. Check
the scores at www.tennis.com.
```

Our expected outcome would be two sentences, telling who won and how to check out the scores. Let's create a static class, **`SimpleSentenceSplit`**, to handle our parsing requirements.

Approach 1 – Substitute text prior to splitting

One approach we can take is to search the text for a list of abbreviations that contain a period and replace the period character with a placeholder. Next, we'll call the **`Split`** function. Finally, before saving the sentences, we replace the placeholder with the period.

The first step would be to take our test sentence, and change it to the following text:

`Mr~ Federer won the tennis match against Mr~ Nadal on Jan~ 28, 2018. Check the scores at www~tennis~com.`

Once the text is converted, the **`Split`** function produces the following result.

```
[0] Mr~ Federer won the tennis match against Mr~ Nadal on Jan~ 28, 2018.
[1] Check the scores at www~tennis~com.
```

We now loop through the "sentences" and replace the ~ character with the period.

The sentence parser would declare a few settings as constant variables when the class is first called. This includes your end-of-sentence characters and any anticipated abbreviation.

Listing 5 – Static variables

```
static public class NLP
  {
     public static char[] EndCharacters =new char[] {'.', '!','?', ';' };
     public static string[] Abbreviations =
         new string[] { "Mr.", "Mrs.", "Dr.","Ms.","Sr.","Jr.","etc.",
                        "Sun.","Mon.","Tue.","Wed.","Thu.","Fri.","Sat.",
                        "Jan.","Feb.","Mar.","Apr.","May.","Jun.","Jul.",
                        "Aug.","Sep.","Oct.","Nov.","Dec.",
                        "www.",".com",".org",".net"};
```

You can add your own abbreviations; the list shown in Listing 5 is just a sample of the possible words you might encounter in the text. The code to do the parsing is shown in Listing 6.

Listing 6 – Parsing method

```
public static List<string> SimpleSentenceSplit(string paragraph)
    {
      char[] CharSep = new char[] { '.', '!', '?', ';' };
      foreach (string curAbbr in Abbreviations)
      {
           paragraph = Regex.Replace(paragraph,curAbbr,
                            curAbbr.Replace(".","~"),
                            RegexOptions.IgnoreCase);
      }
      List<string> Sentences_ = new List<string>();
      string curSentence;
      string[] Sentences = paragraph.Split(CharSep,
                         StringSplitOptions.RemoveEmptyEntries);
      foreach (string sentence in Sentences)
      {
           curSentence = sentence.Replace("~", ".").Trim();
           Sentences_.Add(curSentence);
      }
      return Sentences_;
    }
```

Note: C# doesn't have a case-insensitive* `Replace` *method, so the code uses the* `Regex.Replace` *method to provide case insensitive replacement.

This approach will work reasonably well, but does require some understanding of the likely abbreviations your text can expect. You can adjust the delimiters and abbreviations to fine-tune this parsing strategy. If your goal is to parse a reasonably consistent set of text, this simple approach could give you a usable sentence splitter.

Approach 2 – Checking if the end of the sentence is valid

A second approach is to perform the split first. Then, check each potential "sentence" to see if it is a valid end of sentence, or if the period is used for another purpose and the sentence should be continued. If we apply this approach to our test input, we will get the following array back.

```
[0] Mr.
[1] Federer won the tennis match against Mr.
[2] Nadal on Jan.
[3] 28, 2018.
[4]
[5] Check the scores at www.
[6] tennis.
[7] com.
```

The first key to this approach is adapting the **`split`** function to be sure to include the delimiter character, since we will need that to assemble the final resulting sentences. Listing 7 shows the regular expression to split the string, but uses backtracking to keep the delimiter character.

Listing 7 – Regex to keep delimiter

```
string punctuation = @"(\S.+?[.!?\u2047])(?=\s+|$)";
```

Tip: If you are targeting languages other than English, you can add additional delimiters between the [ ] characters, such as the* `\U2047` *for the double question mark.

We can now write our code to extract the sections from the input text string. The first snippet of the code is shown in Listing 8.

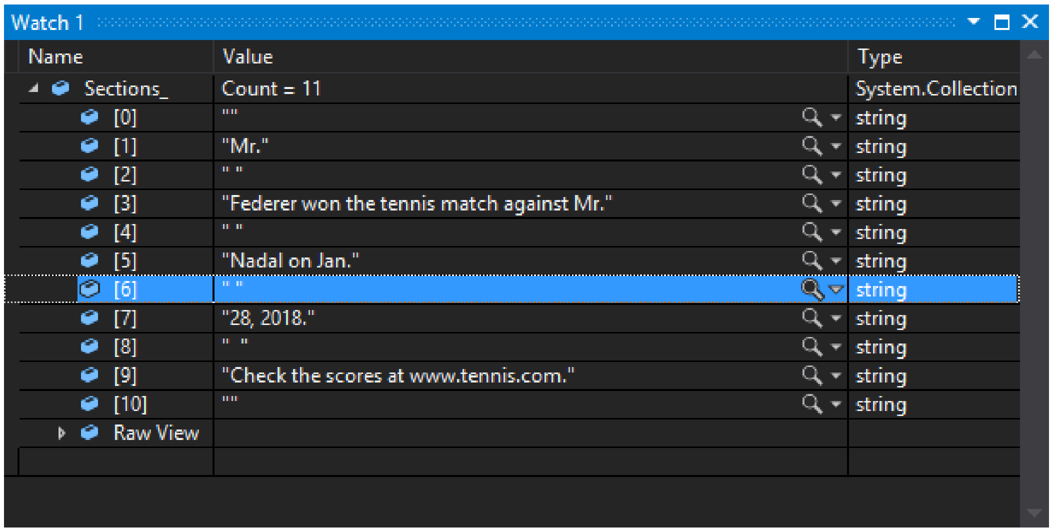
Listing 8 – Collect sections

```
static public List<string> ExtractSentences(string Paragraph)
{
      List<string> Sentences_ = new List<string>();
      List<string> Sections_ = new List<string>();
if (!string.IsNullOrEmpty(Paragraph))
```

```
  {
      // Split by new line character
      List<string> FirstPass =
                   Regex.Split(Paragraph, @"((?:\r ?\n |\r)+)",
                   RegexOptions.IgnorePatternWhitespace).
                   Where(s => s != Environment.NewLine &&
                   !string.IsNullOrEmpty(s)).ToList<string>();
      foreach (string curSentence in FirstPass)
      {
           string[] chunks = Regex.Split(curSentence, punctuation);
           Sections_.AddRange(chunks.ToList<string>());
      }
  }
```

After this code runs, the **Sections_ string** list contains the split elements, including the delimiter character. Figure 8 shows the content of **Sections_ list**.

Figure 8 - First pass



```
Watch 1
Name            Value                                        Type
◢ Sections_     Count = 11                                   System.Collection
    [0]         ""                                           string
    [1]         "Mr."                                        string
    [2]         " "                                          string
    [3]         "Federer won the tennis match against Mr."   string
    [4]         " "                                          string
    [5]         "Nadal on Jan."                              string
    [6]         " "                                          string
    [7]         "28, 2018."                                  string
    [8]         "  "                                         string
    [9]         "Check the scores at www.tennis.com."        string
    [10]        ""                                           string
  ▷ Raw View
```

We now make a second pass, checking to see if the current list element is a sentence or a different usage of the delimiter. Our first step is to build a regex pattern to look for potential uses of the delimiter, other than end of sentence.

```
            Regex rAbbrevs = new
Regex(@"\b(dr|mrs|mr|ms|assn|dept|corp|rte|ave|blvd|hwy"+
       "|www|com|edu|gov|jan|feb|mar|apr|may|jun|jul|aug|sep|"+
        "oct|nov|dec)[.!?] *$", RegexOptions.IgnoreCase);

string acronyms = @"^(?:[A-Z]\.){2,}$";
```

You can add your own expected abbreviations to this list. Knowing the type of questions and input text will be very helpful.

We will also use a bit of LINQ to remove the empty strings from our collected list of sections.

```
Sections_ = Sections_.Where(s => s.Length > 0).ToList<string>();
```

Processing the sections

We now loop through the sections we collected and try to determine whether the section is by itself or part of another word. Listing 9 shows the code.

Listing 9 - Parse collected sections

```
for (int x = 0; x < Sections_.Count; x++)
    {
        string curPart = Sections_[x];          // Current word
        string nextPart = "";                   // Next word
        if (x + 1 < Sections_.Count)
           {
              nextPart = Sections_[x + 1];
              if (nextPart.Length > 0
                  && (rAbbrevs.IsMatch(curPart) || curPart.Length == 1))
                 {
                     Sections_[x + 1] = curPart + nextPart;
                 }
              else
                 {
                    int pv = Sentences_.Count - 1;
                    if (pv >= 0 && Regex.IsMatch(Sentences_[pv], acronyms))
                     {
                         Sentences_[pv] += curPart;
                     }
                    else
                    {
                         Sentences_.Add(curPart);
                    }
                 }
         }
  else
         {
            // Last item, add to the sentence stack
            if (curPart.Trim().Length > 0)
               {
                   Sentences_.Add(curPart);
               }
        }
    }
```

We basically compare the current and next phrase, so in our example.

[1] Mr.

[2] Federer won the tennis match against Mr.

`Mr.` is found by the regular expression, and the second element gets added to it,

Mr. Federer won the tennis match against Mr.

The process continues until all sections are processed. You can add your own rules, such as: if the two sections are numeric (such as currency or IP address), combine them. This allows you to fine-tune the sentence extraction based on your application.

Cloudmersive API call

If you've installed the Cloudmersive API, you can use the code in Listing 10 to extract sentences from a paragraph.

Listing 10 – Cloudmersive sentence extraction

```
static public List<string> ExtractSentencesFromString(string Paragraph)
{
    Configuration.Default.AddApiKey("Apikey", APIKey);
    var apiInstance = new SentencesApi();
    try
       {
         // Extract sentences from string
         string result = apiInstance.SentencesPost(Paragraph);
         string[] Sentences_ = result.Replace("\\n", "").
                  Split(new String[] { "\\r" },
                  StringSplitOptions.RemoveEmptyEntries);
         for(int x=0;x<Sentences_.Length;x++)
            {
                Sentences_[x] = Sentences_[x].Replace("\"", "");
            }
            return Sentences_.ToList<string>();
        }
        catch ()
        {
            return null;
        }
 }
```

Note: Be sure to review the string that comes back from the API to determine which characters you might need. In the example shown in Listing 10, rather than return a string, we are returning a list of strings, after splitting on new line characters and removing the extra quotes placed on both ends of the paragraph.

Sample paragraph

Try the following sample paragraph to confirm how well the program performs at splitting sentences.

Kerri won her match 6-2,6-2. Rachel/Dori also won 6-4,6-3. Dr. Schmidt of Frog Hollow Assn. was on hand to watch. I.B.M. provided the scoring software. The players paid $18.00 to play.

This should test to see how well the parser handles various uses of the period character.

IsQuestion

Once you have the sentences, you might want to determine if the sentence is asking a question (particularly in the case of a system to provide answers). Listing 11 below shows a simple function to determine if the sentence is asking a question.

Listing 11 – IsQuestion

```
public static bool IsQuestion(string text)
 {
    bool isQuestion = text.Trim().EndsWith("?");   // Assumes English only
    if (!isQuestion)
       {
         isQuestion = Regex.IsMatch(text, @"(Who|What|Where|When|How)\s.*",
                      RegexOptions.IgnoreCase);
        }
        return isQuestion;

 }
```

This function would allow you to read each retrieved sentence from the input text and determine which ones need an answer.

Summary

We touched upon the basics of extracting sentences, but did not touch upon all the nuances that can occur. For example, we ignored emoticons and only provided code to handle English. Sentence boundary disambiguation is a complex problem to solve completely, but the code should give you a basic idea of how it works.

Chapter 4 Extracting Words

Fortunately, splitting a sentence into words (called *tokenization* in NLP parlance) is a bit easier of a task than sentence splitting. Delimiters, such as spaces and commas, generally don't have other purposes within a sentence. A simple solution to extracting words is shown in the following example, using the **Char** class from .NET and some LINQ code.

LINQ and the Char class

The **Char** class has methods to determine what a char is, such as letter, number, punctuation mark, etc. Our first step is to find all the punctuation characters in our text, with the following LINQ query.

```
var ListOfSeparators =
sentence.Where(Char.IsPunctuation).Distinct().ToList();
```

If you want to add additional "separators," such as symbols, you can simply append to the **ListOfSeparators**:

```
ListOfSeparators.AddRange(sentence.Where(Char.IsSymbol).Distinct().ToList());
```

You can also add you own word delimiters to the list, in case of any unusual separators that might be common in your application.

Once you've determined your separators, simply perform a **Split()** using the character list you've just built.

```
var words = sentence.Split().Select(x => x.Trim(ListOfSeparators.ToArray()));
```

Let's run our tennis example from the earlier chapter:

Mr. Federer won the tennis match against Mr. Nadal on Jan. 28, 2018.

We will get the following array of words back.

```
[ 'Mr' , 'Federer' , 'won' , 'the' , 'tennis' , 'match' , 'against' , 'Mr' ,
'Nadal' , 'on' ,'Jan' ,'28', '2018' ]
```

Regular expressions

You can also use regular expressions for a compact way to get the list of words. The regular expression engine will generally perform slower than other .NET approaches, but unless you are dealing with a massive amount of text, an end user would not notice the performance difference within an application. Using regular expressions allows the regex engine to decide how to split words, rather than custom splitting. Listing 12 shows how you can split the words using regular expressions.

Listing 12 – Regular expression words

```
List<string> words_ = new List<string>();
var FoundWords = Regex.Matches(sentence, @"\w+[^\s]*\w+|\w");
foreach (Match IndividualWord in FoundWords)
    {
       words_.Add(IndividualWord.Value);
    }
```

Contractions

Contractions, which are a short way to represent two words, are an interesting construct. For example:

- Did not > Didn't
- He will > He'll

Depending on your preferences, you might want to expand contractions to have a better chance of interpreting the text. Listing 13 is a function that will take a list of words and return a new list with the contractions (if any found) expanded.

Listing 13 – Expand contractions

```
        static public List<string> ExpandContractions(List<string> words_)
        {
            List<string> ans_ = new List<string>();
            Regex rgx = new Regex(@"^((\w+)\'(ve|ll|t|d|re))$");
            for (int x = 0; x < words_.Count; x++)
            {
                MatchCollection matches_ = rgx.Matches(words_[x]);
                if (matches_.Count > 0)
                {
                    int lst = matches_[0].Groups.Count - 1;
                    ans_.Add(matches_[0].Groups[lst-1].Value);  // Get word
                    string cont = matches_[0].Groups[lst].Value.ToLower();
                    if (cont == "ve") { ans_.Add("have"); }
                    if (cont == "ll") { ans_.Add("will"); }
                    if (cont == "t") { ans_.Add("not"); }
                    if (cont == "d") { ans_.Add("could"); }
                    if (cont == "re") { ans_.Add("are"); }
                }
                else
                {
                    ans_.Add(words_[x]);
                }
            }
            return ans_;
        }
```

So the sentence "*I could've been a contender"* becomes the following words.

- I
- could
- have
- been
- a
- contender

By breaking the contraction into two words, it helps reduce the work that application needs to do to parse the sentence and attempt to determine its meaning.

Summary

Splitting the words is a simple task, but now you are left with a list of words. In the next chapter, we will start using the word list to help the system determine what the user is asking or telling us.

Chapter 5 Tagging

Now that we have a list of words from the sentence, our next step is to tag the words, essentially providing a part of speech code indicating how the word is being used. Children are taught this in school, words are verbs, nouns, adjectives, etc. However, many words cannot be classified, since how the word is used in context, determines its likely part of speech. One well-known example is the following expression:

Time flies like an arrow; fruit flies like a banana.

This example shows that in the first segment, *flies* is a verb, while in the second, *flies* is a noun and *fruit* is an adjective. Tagging is the processing of using any many clues as possible to come up with the most likely set of tag values for each word in the sentence.

Choosing a tag set

One of the first steps, however, is to choose what labels or tags to use. Two popular tag sets are the Penn Treebank tag set and the Universal POS tag set. The Penn Treebank tag set defines 36 tags that are geared around the English language (plus 12 tags for things like punctuation and currency). The Universal POS tag set was derived from multiple tag sets to try to create a set that can work across languages. In this book, we are going to use the Penn Treebank tags. The tag sets are described in Appendix A and B.

To keep our code simple, we are only going to use a few common tags.

Table 1 - Subset of Tags

Tag	Meaning	Example
DT	Determiner	the, an
CD	Cardinal Number	8, 25, 2016
IN	Preposition	in, on,
JJ	Adjective	great, fast, slower
NN	Noun	person, town, card
NNS	Plural noun	people, cars, files
NNP	Proper noun	Washington, Federer
PRP	Pronoun	him, her, me
RB	Adverb	quickly, patiently

Tag	Meaning	Example
VB	Verb	win, write, talks
VBD	Verbs (past tense)	won, ate, slept
VBG	Verbs (present)	winning, speaking
WP	WH-pronoun	Who, where, when…

Different APIs may use different tag sets; Google's API calls use the Universal tag set, and Microsoft Cognitive services use the Penn Treebank tags. In the chapter where we cover the Google API calls, I will provide code to map the Google tags to the matching Penn Treebank tag.

Everything's a noun

When we start the process of tagging, we begin by assuming everything is a noun. Since nouns make up most English words, it seems a reasonable starting point. This means that if we cannot make a more accurate tag, we will treat it as a noun.

Regular expressions

There are some common patterns in words that we can use to change the tag from the default noun, to a more likely tag. For example, if a word has five or more letters, and ends with a consonant, followed by "ed" (such as saved, accumulated, or assumed), we could make a guess that the word is more likely to be a past-tense verb, rather than a noun. By using regular expressions to look for these patterns, we can possibly determine a better tag.

Table 2 shows some sample regular expressions to try to assign a better tag, based on these word patterns.

Table 2 – Regular expression for tagging

Tag	Expression	Meaning
VBG	**`(?i)^[a-z]{2,}ing$`**	Present tense verb
VBD	**`(?i)^[a-z]{2,}[^aeiou]ed$`**	Past tense verb
CD	**`^(((\d{1,3})(,\d{3})*)|(\d+))(.\d+)?$`**	Numbers
NNS	**`(?i)^[a-z]{2,}[s|ss|sh|ch|x|z|]es$`**	Plural nouns
NNS	**`(?i)^[a-z]{1,}[aeiou]ves$`**	Plural nouns
POS	**`(?i)^[a-z]{2,}\'s$`**	Possessive

Tag	Expression	Meaning
DT	**(?i)^[the|an|a]$**	Determiners
CC	**(?i)^[for|and|nor|but|or|yet|so]$**	Conjunctions
PRP	**(?i)^[I|me|we|us|you|they|him|her|them]$**	Pronouns
RB	**@"(?i)^[a-z]{2,}ly$**	Adverbs

Listing 14 is the start of our **Tagger** class, which make every tag a noun, and uses some regular expressions to identify words that we should consider other tags.

Listing 14 – Tagger class

```
static public class Tagger
{
    static private Dictionary<string, string> Taglist = new
Dictionary<string, string>
    {
       { @"(?i)^[a-z]{2,}ing$","VBG" },                      // Present
verb
       { @"(?i)^[a-z]{2,}[^aeiou]ed$","VBD" },               // Past verb
       { @"^(((\d{1,3})(,\d{3})*)|(\d+))(.\d+)?$", "CD" },   // Numbers,
       { @"(?i)^[a-z]{2,}[s|ss|sh|ch|x|z|]es$","NNS" },      // Plural
nouns
       { @"(?i)^[a-z]{1,}[aeiou]ves$","NNS" },               // Plural
nouns
       { @"(?i)^[a-z]{2,}\'s$","POS" },                      // Possessive
       { @"(?i)^(the|an|a)$","DT" },                         // Determiners
       { @"(?i)^(for|and|nor|but|or|yet|so)$","DT" },        //
Conjunctions
       { @"(?i)^(I|me|we|us|you|they|him|her|them)$","PRP"}, // Pronouns
       { @"(?i)^[a-z]{2,}ly$","RB" },                        // Adverbs
    };
    static public string TagWord(string Word_)
    {
       string tag = "NN";                  // Assume its a noun
       foreach (var RegEx in Taglist)
       {
         if (Regex.IsMatch(Word_, RegEx.Key))
            {
               tag = RegEx.Value;
               break;
            }
       }
       return tag;
    }
}
```

We assume the word is a noun, but check the regular expression to see if there is a better tag. The regular expression check improves our tagging performance, but we need to add more coding to get a likely set of tags we can interpret.

Tip: If a word is caught in the regular expression code, there is no need to repeat the word in your dictionary. Be sure to consider the regex "catches" before building your dictionary.

Dictionary lookup

Although regular expressions will help, there are many cases where the regular expression is going to get it wrong. To improve our tagging work, we are going to create a dictionary of common words (top 500 verbs, top 500 adjectives, etc.). Because we know our intended use (question-answering), we can get by with a smaller dictionary of words. For a large, more free-form question answering, you would most likely need a much larger word list (or rely on one of the NLP APIs available).

Listing 15 shows a fragment of the dictionary collection.

Listing 15 – Dictionary

```
static private Dictionary<string, string>
        MyDictionary = new Dictionary<string, string>
        {
            {"aboard","RB"},
            {"almost","RB"},
            {"always","RB"},
            {"and/or","CC"},
            {"bad","JJ"},
            {"became","VBD"},
            {"began","VBD"},
            {"best","JJS"},

            . . .

        }
```

Note: The source code, with a more complete word list, is available at https://github.com/SyncfusionSuccinctlyE-Books/Natural-Language-Processing-Succinctly.

To use this dictionary, we are going to add additional code after our regular expression lookup, but before we return the final tag. Listing 16 shows the code to look the word up in the dictionary.

Listing 16 – Look up tag in dictionary

```
string WordLower = Word_.ToLower();
```

```
    if (MyDictionary.ContainsKey(WordLower))
       {
         tag= MyDictionary[WordLower];
       }
```

Fine-tuning our nouns

Microsoft has a feature (borrowed from Entity framework) that can identify a noun as singular or plural. The following code snippet shows how we can add this service to our NLP library.

```
private static PluralizationService ps;
   static Tagger()
   {
       CultureInfo ci = new CultureInfo("en-us");
       ps = PluralizationService.CreateService(ci);
   }
```

Be sure to add the following using statements to your code.

```
using System.Data.Entity.Design.PluralizationServices;
using System.Globalization;
```

We can now use this feature, and a simple regular expression, to fine-tune our noun tag. Listing 17 shows some code that attempts to distinguish between singular and plural nouns, and to take a guess as to whether the noun is a proper noun.

Listing 17 – Fine-tune noun tag

```
            if (ps.IsPlural(WordLower)  && tag=="NN")
            {
                tag = "NNS";
            }
            // If nothing found, maybe it is a proper noun?
            if (tag=="NN" && Regex.IsMatch(Word_, @"[A-Z]{1}[a-z\-]{2,}"))
            {
                tag = "NNP";
            }
            return tag;
```

We have assumed a noun (**NN**) and marked it as plural (**NNS**) based on the pluralization service. Finally, we use a regular expression to assume that a word beginning with a capital letter, following by two or more lowercase letters (and an optional hyphen), is probably a proper name (**NNP**). While this simple regular expression won't handle all names, it should help us make a reasonable guess as to the type of noun.

Adding words

Your application may have its own unique vocabulary, so the class library has two additional methods to allow you to add words to the dictionary, or regular expressions to the regex list. Listing 18 shows two methods that allow you to customize your dictionaries.

Listing 18 – Add to dictionaries

```
        public  enum ErrorCode
            {   ADDED_OK,
                INVALID_REGEX,
                INVALID_TAG,
                DUPLICATE_ENTRY
            }
        static public ErrorCode AddExpression(string pattern, string tag)
        {
            ErrorCode ans = ErrorCode.ADDED_OK;
            if (TreebankTagList.IndexOf(tag) < 0)
            {
                return ErrorCode.INVALID_TAG;
            }
            try
            {
                new Regex(pattern);
            }
            catch {
                ans = ErrorCode.INVALID_REGEX;
            }
            return ans;
        }

        static public ErrorCode AddToDictionary(string Word_, string tag,
                      bool Overwrite=false)
        {
            ErrorCode ans = ErrorCode.ADDED_OK;
            if (TreebankTagList.IndexOf(tag)<0)
            {
                return ErrorCode.INVALID_TAG;
            }
            string WordLower = Word_.ToLower();
            if (MyDictionary.ContainsKey(WordLower))
            {
                if (Overwrite)
                {
                    MyDictionary[WordLower] = tag;
                    return ErrorCode.ADDED_OK;
                }
                return ErrorCode.DUPLICATE_ENTRY;
            }
            MyDictionary.Add(WordLower, tag);
            return ans;
        }
```

The routine allows you to access the tagger's internal structures and returns some error codes if something occurs (such as a bad regex expression or duplicate entries).

Not quite done

At this point, we’ve tagged the individual words and have created a reasonable set of tags. However, we can fine-tune the list even more by considering the entire list of words and tags, not just the individual words.

Not all sentences will play that nicely with the tags we've assigned. For example, a question like *“Who spoke at the play?”* would produce the following set of tags:

Table 3 - First pass

Word	Tag
Who	`WP`
spoke	`NN`
at	`IN`
the	`DT`
play	`VB`

Since **spoke** is not in our dictionary, it is assumed to be a noun. Trying to determine how to answer a question based on that parsing would be difficult to do. This is where our next step comes into play: tweaking. ☺

To improve our word tags, we need to rely on more than simply looking up words and assigning the appropriate parts of speech. We are going to implement a very simple algorithm, loosely based on the Brill tagger. The Brill tagger algorithm was invented by Eric Brill in 1993 as part of his Ph.D. thesis. Essentially, it says to make your best guess, then correct any thing you might have gotten wrong.

Brill steps

Once we've assigned the tags, we need to process "rules" to see if any tags should be switched. Our rules structure is shown in the following table: It compares two sequential words and tags. The first tag gets switched to the replacement tag if the condition specified is met.

Table 4 shows a few example rules.

Table 4 – Sample tag swap rules

Tag	Change to	Condition
NN	**VB**	After any question word, assume a verb **WP**, **NN** (if **WP** is following by a **NN**)
VB	**NN**	Assume a noun after any determiner **DT**, **VB** (if a **VB** follows a **DT**)
VB	**NN**	Determine one or more adjectives, verb **DT**, **JJ+**,**VB**

The rule syntax is based on regular expressions. For example, the third rule indicates that a verb should be changed to a noun if it occurs after a determiner, and one or more adjectives. For example, the word *play* is generally considered a verb. However, if the sentence started with "*An exciting play*," the rule parser would determine that instead of **DT, JJS, VB** (determiner, superlative adjective, verb), it should be **DT, JJS, NN.**

Using the rules shown in Table 4, our new set of tags for our problematic sentence, "*Who spoke at the play*," become:

Table 5 - First pass

Word	Tag
Who	**WP**
spoke	**VB**
at	**IN**
the	**DT**
play	**NN**

This represents an interpretation of the input sentence that is much more likely to be accurate. Listing 19 shows the code to adjust the collection of words and tags based on the rules table.

Listing 19 – Transformation rules

```
static private Dictionary<string, string>
       TransformRules = new Dictionary<string, string>  {
            { "WP,NN"     ,"NN=VB" },   // Who spoke at the rally?
            { "DT,VB"     ,"VB=NN" },   // The play really resonated
            { "DT,JJ,VB"  ,"JJ=NN" },   // The rich pay more taxes
            { "DT,JJ+,VB" ,"VB=NN" }    // The amazing score was great
        };
```

Listing 20 shows the code to apply these rules prior to returning the tag list.

Listing 20 – Apply rules to tag list

```
static public List<string> RevisedTags(List<string> Words_,List<string>
Tags_)
{
    List<string> revised = Tags_;
    for(int x=0;x<revised.Count-1;x++)
    {
       string curTag = revised[x];
       foreach (KeyValuePair<string, string> pair in TransformRules)
          {
               string[] tagList = pair.Key.Split(',');
               if (tagList[0] == curTag)
               {
                   int EndPos = revised.IndexOf(tagList.Last(), x + 1);
                   if (EndPos>0)                    {
                       string[] newTags = pair.Value.Split('=');
                       string ThisTag = newTags[0];
                       string Becomes = newTags[1];
                       int ThisPos = revised.IndexOf(ThisTag, x + 1);
                       int NumTagsBetween = (EndPos - x) - 1;
                       if (NumTagsBetween < 1 && tagList.Count() == 2)
                       {
                            revised[ThisPos] = Becomes;
                            break;
                       }
                       if (tagList.Count() == 3)
                       {
                          string TagsBetween = "";
                          for (int y = x + 1; y < EndPos; y++)
                          {
                              TagsBetween += revised[y] + ",";
                          }
                          if (MatchMiddlePattern(tagList[1],TagsBetween))
                          {
                              revised[ThisPos] = Becomes;
                             break;
                          }
                       }
                   }
               }
            }
        }
       return revised;
  }
```

Listing 21 shows the pattern-matching code that deals with patterns using regular expression syntax.

Listing 21 – Pattern matching

```
static private bool MatchMiddlePattern(string Pattern, string tagsBetween)
  {
     bool ans = false;
     string lastchar = Pattern.Substring(Pattern.Length - 1);
     // Regex search patterns
     if ("+*?".IndexOf(lastchar)>=0)
        {
           string regex = @"^(" + Pattern.Replace(lastchar, "," +
                                  lastchar) + ")$";
           ans = Regex.IsMatch(tagsBetween, regex);
        }
     else
       {
           ans = tagsBetween.StartsWith(Pattern);
       }
    return ans;
 }
```

The code reads each tag from your list. If the current tag is found in the transform rules, it then looks to see if the ending tag is found within your tag list as well. If both tags are found, we then compare the tags in between the two tags. The tag in-between borrows syntax from regular expressions, such as the options shown in Table 6.

Table 6 – Tag patterns

Middle pattern	Description
No tag	Tags follow immediately after one another
Tag *	0 or more occurrences of the tag
Tag +	1 or more occurrences of the tag
Tag ?	0 or 1 occurrence of the tag

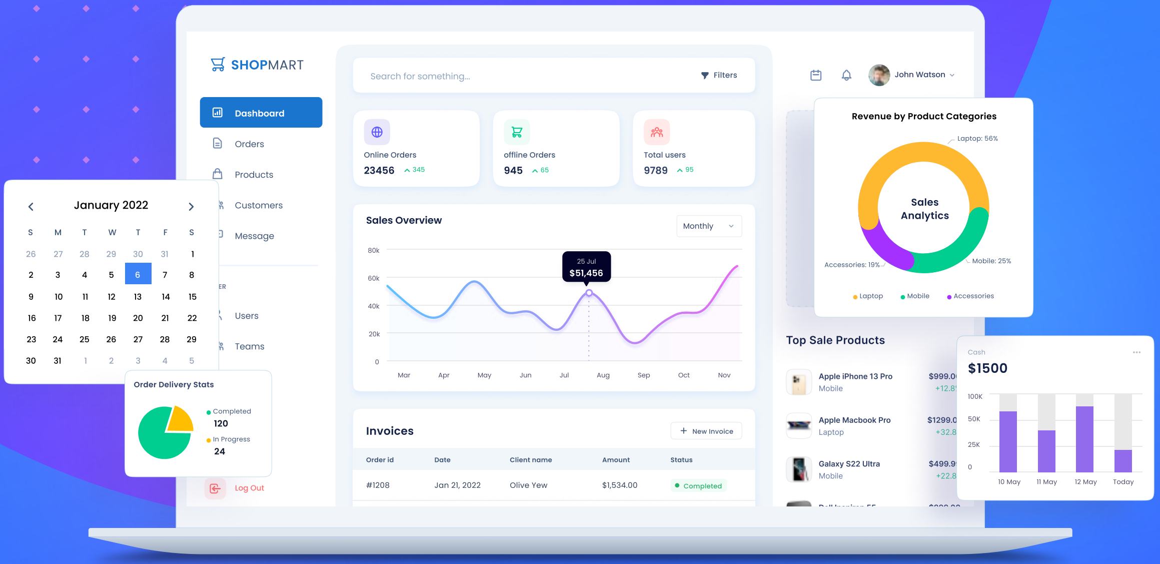
This is a very simple example of a Brill-based tagging routine, combining regular expressions and word lookups to provide a reasonable interpretation of the text the user entered.

Summary

This chapter gave a simple tagger example, with a very small set of words and rules. Implementing a tagger capable of handling more complex sentences would require a larger word set, more tags, and several additional rules. For example, you could adapt the dictionary to return all possible tags a word could have, or you could consider the tense of word or whether it's singular or plural. English is an ambiguous and complex language, and a tagger to understand all those nuances would be a very ambitious undertaking (leave it to the big guys).

THE WORLD'S BEST
UI COMPONENT SUITE
FOR BUILDING
POWERFUL APPS

4.6 out of 5 stars



GET YOUR **FREE** .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

IBM SIEMENS intel VISA

Syncfusion

Chapter 6 Entity Recognition

Once you have a tagged collection of words, you should scan the list for named entities. A named entity could be something like a city, a person, or a corporation. Essentially, it is likely a noun or adjective and noun that has meaning for your application. If you were processing a travel application, and wanted the user to be able to say, "I want to fly from Philadelphia to Orlando," your system should consider airport cities as important entities to extract from the sentence.

In addition to named entities, there are certain “nouns” that have special meaning, such as email addresses, phone numbers, or credit card numbers. Recognizing these entities can improve your application’s ability to determine what the text contains.

The more information we can gather about the word, the better we will be able to extract meaning and find answers to the user's English questions.

Note: Your application might need to expand upon the entity types, depending on what type of questions you need answered. If you are developing a human resources application, for example, you might want to include ID number (Social Security number), phone numbers and email addresses.

Entity types

Table 7 contains a list of entity type tags we might consider within our application.

Table 7 - Entity type tags

Entity tag	Description
`PERSON`	Person’s name
`FAC`	Buildings, airports, etc.
`ORG`	Organizations like companies and agencies
`GPE`	Geographic entities like states and countries
`EVENT`	Named events such as French Open and Mardi Gras
`DEMOGRAPHIC`	Fields about a person such as phone number and email
`DATE`	A date value, such as mm/dd/yyyy or tomorrow
`TIME`	Hours and minutes, clock time
`MONEY`	Currency values

Named entities

The simplest way to find named entities is by looking them up in a dictionary object. If you create a simple list of strings, comparing them against the words in the tag list is a simple matter. We might want to add some new tags, such as Airport, Person, City, Organization, or Event for our travel application. Listing 22 shows the code to traverse the tag and word list, and if possible, update the tag to identify it as an entity.

Listing 22 – Check tags for specific entities

```
static private Dictionary<string, string> NamedEntities =
        new Dictionary<string, string>
        {
            { "US OPEN"          ,"EVENT" },
            { "FRENCH OPEN"      ,"EVENT" },
            { "AUSTRALIAN OPEN"  ,"EVENT" },
            { "WIMBLEDON"        ,"EVENT" },
            { "MARDI GRAS"       ,"EVENT" },
            { "MICROSOFT"        ,"ORG" },
            { "GOOGLE"           ,"ORG" }
        };


static public void UpdateNamedEntities(List<string> Words_, List<string>
Tags_)
{
     var WordArray = Words_.ToArray();
     foreach (KeyValuePair<string, string> pair in NamedEntities)
     {
         string[] tags = pair.Key.ToString().Split(' ');
         int nSize = tags.Length;
         int nStart = Array.FindIndex(WordArray, ne => ne.Equals(tags[0],
                     StringComparison.InvariantCultureIgnoreCase));
         if (nStart >= 0)  {
            if (nStart + nSize <= Words_.Count)    {
                 bool FoundIt = true;
                 int TagPos = 1;
                 for (int x = nStart + 1; x < nStart + nSize; x++) {
                    if (Words_[x].ToUpper() != tags[TagPos].ToUpper()) {
                        FoundIt = false;
                        break;
                     }
                     TagPos++;
                  }
                  if (FoundIt)
                  {
                       Tags_[nStart] = pair.Value;
                        Words_[nStart] = pair.Key;
                        if (nSize > 1 && (nSize + nStart <= Words_.Count))
                        {
                            Tags_.RemoveRange(nStart + 1, nSize - 1);
```

```
                              Words_.RemoveRange(nStart + 1, nSize - 1);
                         }
                    }
                }
                 }
            }
        }
```

Patterns

In addition to a word list, we are also going to create a dictionary of patterns (via regular expressions) to help us identity "entities." In general, if a word matches one of the defined patterns, we are going to assign it as an entity.

Listing 23 shows some of the patterns we are looking for.

Listing 23 - Regex list

```
static public Dictionary<string, string> Patterns =
      new Dictionary<string, string>
{
   {@"^\$(\d{1,3}(\,\d{3})*|(\d+))(\.\d{2})?$","CURRENCY"  },
   {@"(?i)^[a-zA-Z0-9\-\.]+\.(com|org|net|mil|edu)$","URL"   },



   {@"^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|
     (([a-zA-Z0-9\-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$","EMAIL" },
   {@"^((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}$","PHONE" },
   {@"^((\d{2})|(\d))\/((\d{2})|(\d))\/((\d{4})|(\d{2}))$","DATE" },
   {@"^(\d{4})$","YEAR"  },
   {@"(?i)^(Mr|Ms|Miss|Ms)$","TITLE"},
   {@"^(\d+)$","CD"},



   {@" (?i)^(Jan(uary)?|Feb(ruary)?|Mar(ch)?|Apr(il)?|May|Jun(e)?|
Jul(y)?|Aug(ust)?|Sep(tember)?|Sept|Oct(ober)?|Nov(ember)?|
Dec(ember)?)$","MONTH" }
};
```

Again, knowing your type of application will help indicate which regular expressions you should look for. If your system is a Human Resources application, identifying Social Security numbers is a good idea. For a payment system, you might want to identify the credit cards. If you are processing emails, emoticons would be useful.

Listing 24 shows the code that loops through your words and updates any tags back on the regular expression patterns you've defined.

Listing 24 – Regex pattern matching

```
for (int x = 0; x < Words_.Count; x++)
{
     string curWord = Words_[x];
     foreach (KeyValuePair<string, string> pair in Patterns)
     {
        if (Regex.IsMatch(curWord, pair.Key))
        {
            Tags_[x]= pair.Value;
             break;
         }
     }

}
```

There are a couple things to keep in mind. First, be sure to put your regular expressions in order, so that the **`YEAR`** tag (four digits) is checked before the cardinal regex (any number of digits). Second, you might need additional checks in your code. For example, before we consider a four-digit number to be a year, we want to confirm it falls between 1960 and 2029. Listing 25 shows just such a check.

Listing 25 – Confirm year

```
int yyyy = Convert.ToInt32(Word_);
           if (yyyy < 1960 || yyyy > 2029)
              {
                 continue;
              }
```

In a Human Resources application, you might want to consider that a four-digit number should only be considered a year if it's lower than the current year (asking someone for their birth date).

A good source for regular expressions is the Regex Library website. They have a large number of user contributed regular expressions.

Rule-based lookup

Another approach for identifying named entities, is to look for tag patterns that suggest an entity, rather than individual words. For example, the tag **`TITLE`** followed by one or more tags of **`NNP`** (proper name) suggests a person. Anytime we see that pattern, we should create a single-person entity. Our pattern dictionary would be very similar to the rules we used when we tweaked the tag list while tagging words.

Listing 26 – Entity patterns

```
static public Dictionary<string, string> EntityPatterns = new
Dictionary<string, string>
        {
            {"TITLE|NNP|NNP","PERSON" },
            {"TITLE|NNP","PERSON" }

        };
```

This rule says: if we find the pattern of title followed by one or two proper names, we combine it into a single tagged word, and tag it as a **`PERSON`**. So, Mr. Joseph Booth as three tags, becomes a single **`PERSON`** tag, with the value of **`Mr. Joseph Booth`**.

The code to process the pattern list is shown in Listing 27.

Listing 27 - Process tag patterns

```
foreach (KeyValuePair<string, string> pair in TagPatterns)
{
    string[] tags = pair.Key.ToString().Split('|');
    string EntityValue = "";
    int nSize = tags.Length;
    int nStart = Tags_.IndexOf(tags[0]);
    if (nStart >= 0)
    {
        if (nStart + nSize <= Tags_.Count)
        {
            EntityValue += Words_[nStart];
            bool FoundIt = true;
            int TagPos = 1;
            for (int x = nStart + 1; x < nStart + nSize; x++)
            {
                if (Tags_[x] != tags[TagPos])
                {
                   FoundIt = false;
                   break;
                 }
                 EntityValue += " " + Words_[x];
                 TagPos++;
            }
            if (FoundIt)
            {
               Tags_[nStart] = pair.Value;
               Words_[nStart] = EntityValue;
               if (nSize > 1 && (nSize + nStart <= Words_.Count))
               {
                  Tags_.RemoveRange(nStart + 1, nSize - 1);
                  Words_.RemoveRange(nStart + 1, nSize - 1);
                }
             }
           }
       }
   }
```

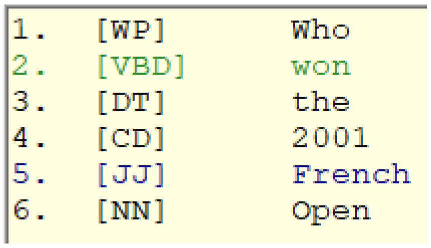
Example question

Let’s say the user asks the following question:

Who won the 2001 French Open?

After tagging, the result is shown in Figure 9.

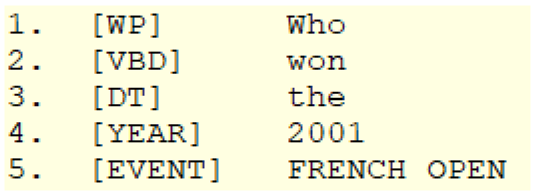
Figure 9 – Tagging results



```
1.   [WP]        Who
2.   [VBD]       won
3.   [DT]        the
4.   [CD]        2001
5.   [JJ]        French
6.   [NN]        Open
```

After we apply the named entity recognition, the result is shown in Figure 10.

Figure 10 – After Named entities



```
1.   [WP]        Who
2.   [VBD]       won
3.   [DT]        the
4.   [YEAR]      2001
5.   [EVENT]     FRENCH OPEN
```

Based on that sentence structure, we can determine a course of action: look up the winner of the French Open from 2001.

Remembering

Once you get information from a question, your application should remember key components. A user querying the tennis results might see dialog like this:

Who won the 2001 French Open?

Gustavo Kuerten won the men's side and Jennifer Capriati won on the women's side.

Who lost?

Àlex Corretja lost to Gustavo Kuerten and Kim Clijsters lost to Jennifer Capriati.

Since your code needs to know the year and name of the tournament, it should remember those key variables from the previous questions. The better you know your data and types of questions, the better you'll be able to determine which pieces of information to retain and plug in for later questions.

Prompting

You might also want to consider the system asking for additional information, in case something in missing in the sentence. For example:

Who won the French Open?

You could assume the question refers to the most recent French Open, or you could prompt back with a question to obtain the missing information: **In what year?**

A lot of these decisions about how to plan the interaction will help determine how well received your application is. For example, in our tennis application, many people refer to the player by their last names. We could consider adding the last name to our named entity stack, so someone asking "Did Nadal win the French Open in 2014?" would come back with an answer without knowing Nadal's first name (*Raphael*).

Summary

After tagging and applying named entities and some chunking logic, we should be able to come up with enough information to determine what the user wants to know. In the next few chapters, we will see how to build our knowledge base, and after that, how to provide an answer to the user.

Chapter 7 Knowledge Base

As we mentioned earlier, building a knowledge base of every known fact would be an incredibly complex and massive undertaking. However, if we can restrict our knowledge base to a small set of data, we can create a system that allows end-users to query that knowledge with English questions.

Many applications have small data sets that can be queried. For example, you might want to ask an employee system questions like: *"Who is up for a review this month?"* or *"Show me employees working for Miss Blaire."* An ordering system might answer questions such as *"When did order #1234 ship?"* or *"Which unshipped orders are older than one week?"*

Example domain

The first step in building the knowledge base, is to determine the data subset you want to use. For our example code, we are going to use the list of tennis champions at the grand slam events since 1968. This gives use a small base (50 years x 4 events x 2 genders) of 400 rows of data. Such a small base allows us to store the data in memory and use LINQ to process it. Larger systems might rely on SQL server or possibly third-party API calls.

Our sample data contains the following columns:

- Event
- Gender
- Year
- Champion Seed
- Champion Country
- Champion
- Runner-up Seed
- Runner-up Country
- Runner-up
- Score in the Final

The data was obtained from data world and released to the public domain. The link to this dataset is here.

Data World is a website of collaborative data sets contributed by users. Be sure to review the licensing, as each contributor may have their own licensing and attribution rules.

Base classes

To use the data, we are going to create a couple of base classes and then load the data into an enumerated list, so we can rely on LINQ queries to search. If you have a much larger data set, or data stored in a SQL database, you will probably use SQL queries or Entity Framework to retrieve the data.

Player class

This class represents the tennis players from the dataset. Listing 28 shows the base **Player** class. We rely on a very simple **Split** function to take player's name and return the first and last name.

Listing 28 – Player class

```
public class Player
    {
        public string FullName { get; set; }
        public string Country { get; set; }
        public int Seed { get; set;  }
        public string FirstName
        {
            get
            {
                string[] names = FullName.Split(' ');
                return names[0];
            }
        }
        public string LastName
        {
            get
            {
                string[] names = FullName.Split(' ');
                return names[names.Length - 1];
            }
        }
   }
```

If you are building a data set that refers to people's names, the names should probably be treated as named entities. By exposing our **NamedEntities** collection, we can add code to identify the players when their names appear in a question text.

Tournament class

We also have a class that represents the tournament. Listing 29 shows the **Tournament** base class.

Listing 29 – Tournament class

```
public class Tournament
    {
        public string Name { get; set; }
        public int Year { get; set; }
        public string Gender { get; set; }
        public string FinalScore { get; set; }

        public Player Winner { get; set; }
        public Player RunnerUp { get; set; }
        public int SetsPlayed
        {
            get
            {
                string[] ans_ = FinalScore.Split(',');
                return ans_.Length;
            }
        }
    }
```

Loading our data

I took the data set, and using some Excel formulas, create a text file containing the data as a pipe-delimited file. (The final score column contained commas, so the pipe delimiter was used instead.) Some sample rows are shown in Figure 11.

Figure 11 – Pipe-delimited tennis results

```
AUS|M|1969|Rod Laver|1|AUS|Andrés Gimeno|9|ESP|6-3, 6-4, 7-5
AUS|M|1970|Arthur Ashe|4|USA|Dick Crealy|12|AUS|6-4, 9-7, 6-2
AUS|M|1971|Ken Rosewall|2|AUS|Arthur Ashe|3|USA|6-1, 7-5, 6-3
```

Now we'll create a static class called **TennisMajors**, and declare a collection to hold the tournament results.

```
        private static List<Tournament> TennisResults;

        static TennisMajors() {
            LoadDataSet();

        }
```

The class will load the data set once at startup when the class is first referenced. Listing 30 shows the class to load the data file into the collection.

Listing 30 – Load data set

```
static private void LoadDataSet()
   {
      List<string> Players = new List<string>();
      TennisResults = new List<Tournament>();
      string[] lines = System.IO.File.ReadAllLines(@"Tennis.txt");
      for(var x=0;x<lines.Length;x++)
        {
          string[] CurrentData = lines[x].Split('|');
          Tournament CurTournament = new Tournament
```

```
               {
                  Name = CurrentData[0].ToUpper().Trim(),
                  Gender = CurrentData[1].ToUpper().Trim(),
                  Year = Convert.ToInt16(CurrentData[2]),
                  FinalScore = CurrentData[9],
                  Winner = new Player(),
                  RunnerUp = new Player()
               };
               CurTournament.Winner.FullName = CurrentData[3].Trim();
               CurTournament.Winner.Seed = Convert.ToInt16(CurrentData[4]);
               CurTournament.Winner.Country = CurrentData[5].Trim();

               CurTournament.RunnerUp.FullName = CurrentData[6].Trim();
               CurTournament.RunnerUp.Seed =
Convert.ToInt16(CurrentData[7]);
               CurTournament.RunnerUp.Country = CurrentData[8].Trim();
               TennisResults.Add(CurTournament);
               Players.Add(CurTournament.Winner.FullName);
               Players.Add(CurTournament.RunnerUp.FullName);
         }

       var UniquePlayers = Players.Distinct();
       foreach (string FullName in UniquePlayers)
       {
            Entities.NamedEntities.Add(FullName, "PERSON");
       }
}
```

The code loads the dataset from the text file and grabs the player's names as it loads the data. The player's names are then loaded to the `Entities` collection in the base NLP object to give us a collection of named entities from the tennis players. So now, a question such as:

Who did Serena Williams beat in the 2014 US Open?

Will tag **Serena Williams** as **PERSON**, **2014** as a **YEAR**, and **US Open** as an **EVENT**.

Single answers

Based on our dataset and expectations, we should provide the following functions at a minimum. If the year is 0, it will assume the first tournament. Most of these functions are simple LINQ queries against the collection.

GetResults(TournamentName, Year)

Listing 31 shows a method to look up the tournament, year, and optionally the gender, and return the winner, loser, and number of sets. We are returning the data as a delimited string, and allowing the program using the data the format the response.

Listing 31 – Get results

```
static Tournament GetResults(string TournamentName, int Year, string
Gender)
   {
     Tournament Fnd = TennisResults.FirstOrDefault(x => x.Year == Year
                      && x.Name == TournamentName && x.Gender==Gender);
      return Fnd;
}
```

If no results are found, the method returns **NULL**; otherwise, it will return the tournament object requested. Using this method, we can easily create some simple wrappers (or allow the query program to make direct use of the object).

WhoWon(TournamentName, Year, Gender)

WhoWon is simply a wrapper around the **GetResults** call, returning the winner full name from the **GetResults** call. Listing 32 shows the **WhoWon** method.

Listing 32 – Who won

```
static public string WhoWon(string TournamentName, int Year, string Gender)
{
   string ans_ = "";
   Tournament Results_ = GetResults(TournamentName, Year, Gender);
   if (Results_ != null)
      {
         ans_ = Results_.Winner.FullName;
       }
       return ans_;
}
```

WhoLost(TournamentName, Year, Gender)

WhoLost is the same code, except that it returns the name of the runner up instead.

FinalScore(TournamentName, Year, Gender)

FinalScore (Listing 33) simply pulls the **FinalScore** property from the **Tournament** object.

Listing 33 – Final score

```
static public string FinalScore(string TournamentName, int Year, string
Gender)
{
   string ans_ = "";
   Tournament Results_ = GetResults(TournamentName, Year, Gender);
   if (Results_ != null)
      {
         ans_ = Results_.FinalScore;
      }
   return ans_;
}
```

The wrapper functions are not necessary, but simply hide the details of the **`Tournament`** object behind the scenes.

Aggregate answers

You might also want to answer some questions about the history of the tournament, again by using LINQ queries against the collection.

MostWins(TournamentName, Gender)

`MostWins` is a function using a LINQ query that determines for a particular tournament, who has won it the most times. Listing 34 shows the method.

Listing 34 – Most wins

```
static public string MostWins(string TournamentName, string Gender)
{
    string Winningest = "";
    var ans_ = TennisResults.Where(x => x.Name.ToLower() ==
                                TournamentName.ToLower()
         && x.Gender == Gender).
            GroupBy(x => x.Winner.FullName,
    (key, values) => new { Player = key, Count = values.Count()  }).
    OrderByDescending(y=>y.Count);
    var MostWins = ans_.FirstOrDefault();
    if (MostWins != null)
       {
         Winningest = MostWins.Player + " has won " +
          MostWins.Count + " times";
        }
    return Winningest;
}
```

Tip: If you are not familiar with LINQ, I recommend reading Jason Robert's LINQ Succinctly title.

MostLosses(TouramentName, Gender)

`MostLosses` is the same code, except looking at the runner up, rather than the winner name during the **`GroupBy`**.

PlayerWins(PlayerName,TournamentName)

Listing 35 shows the code to determine how many times a player has won the tournament.

Listing 35 – Player wins

```
static public string PlayerWins(string PlayerName,string TournamentName)
{
   string ans = PlayerName + " has never won.";
   int NumTimes_ = TennisResults.Where(x => x.Name.ToLower() ==
TournamentName.ToLower()
                  && x.Winner.FullName.ToUpper() ==
PlayerName.ToUpper()).Count();
   if (NumTimes_>0)
    {
      if (NumTimes_ == 1) { ans = PlayerName + " has won once"; }
      if (NumTimes_ == 2) { ans = PlayerName + " has won twice"; }
      if (NumTimes_>2) {
         ans = PlayerName + " has won " + NumTimes_.ToString() + " times";
}
    }
   return ans;
}
```

PlayerLosses(PlayerName,TournamentName)

In Listing 36, determining the number of losses is slightly different, because we need to distinguish between a player who never reached the finals versus a player who reached the finals, but lost.

Listing 36 – Player losses

```
static public string PlayerLosses(string PlayerName, string TournamentName)
{
    string ans = PlayerName + " has never reached the final.";
    int NumWins_ = TennisResults.Where(x => x.Name.ToLower() ==
TournamentName.ToLower()
                       && x.Winner.FullName.ToUpper() ==
PlayerName.ToUpper()).Count();
    int NumLost_ = TennisResults.Where(x => x.Name.ToLower() ==
TournamentName.ToLower()
                       && x.RunnerUp.FullName.ToUpper() ==
PlayerName.ToUpper()).Count();

    // At least player reached the final
    if (NumWins_+NumLost_ > 0)
    {
         if (NumLost_ < 1) { ans = PlayerName + " has never lost in the
finals"; }
         if (NumLost_ == 1 && NumWins_ < 1) {
              ans = PlayerName + " lost all the only time they reached the
final"; }
         if (NumLost_ > 1 && NumWins_ < 1) {
                   ans = PlayerName + " lost all " +
                   (NumLost_).ToString() + " times they played"; }
```

```
          if (NumLost_ > 0 && NumWins_ > 0) { ans = PlayerName + " lost " +
              (NumLost_) + " times in " + (NumLost_ + NumWins_).ToString()+
                " trips to the finals"; }
      }
      return ans;
 }
```

Note: The code to the Tennis Data and all the lookup functions can be downloaded from https://github.com/SyncfusionSuccinctlyE-Books/Natural-Language-Processing-Succinctly.

Summary

The question-answering component of the application needs to understand how to get the answers, be it a LINQ query, SQL call, or web service. The list of tagged words is parsed, and hopefully, tied to one of the functions you wrote. The goal is interpreting a tagged word list and extracting enough information to know which function to call to determine the answer.

Now that we know what data is available, we need to take the user's question and determine which function to call to give the best answers. That is what we will discuss in Chapter 8.

Chapter 8 Answering Questions

We've reached this point with the ability to generate a list of tagged words from a text. Whether you relied on the Cloudmersive (or other API) or used the code presented in the book, that tagged list of word objects give us a good chance to have the computer respond, in a somewhat friendly manner, to the user's text. Our basic method will be one that takes that tagged sentence and tries to provide a response.

```
static public string GetResponse(List<string> Words_,List<string> Tags_)
```

The word list and the tag list are passed as parameters. By searching through the lists, we should be able to determine what the user is asking, and how to answer.

We now have the phrases and a set of functions about our data. In this chapter, we will integrate the pieces and allow you to ask questions and get answers. Figure 12 shows a conversation with the tennis major application.

Figure 12 – Talking tennis

Who won the 2001 French Open?

Gustavo Kuerten won the men's side and Jennifer Capriati won on the women's side.

Who lost?

Àlex Corretja lost to Gustavo Kuerten and Kim Clijsters lost to Jennifer Capriati.

Who won the first Wimbledon?

Rod Laver won in a Good match over Tony Roche and Billie Jean King won in a Good match over Judy Tegart.

Note: Our dataset only goes back 50 years, so "first" refers to the first Wimbledon in our dataset, when the first actual Wimbledon tournament was played in 1877 and won by Spencer Gore.

Who has won the most Wimbledons?

Roger Federer has won 8 Wimbledons**.**

Getting started

Our goal sounds simple—we need to determine three things to answer the question. First, what is being asked. Second, which function call has the answer. Third, if we can get the information the function call needs from the list of word info objects.

What we can answer

If we review the functions from the previous chapter, we know how to answer the following questions. Table 8 lists the functions and the information we need to determine to call the functions.

Table 8 – Functions and parameters

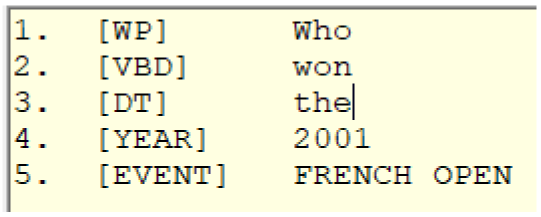
Function	Parameters
`WhoWon`	**`Tournament`**, **`Year`**, **`Gender`**
`WhoLost`	**`Tournament`**, **`Year`**, **`Gender`**
`FinalScore`	**`Tournament`**, **`Year`**, **`Gender`**
`MostWins`	**`Tournament`**, **`Gender`**
`MostLosses`	**`Tournament`**, **`Gender`**
`PlayerWins`	**`Tournament`**, **`PlayerName`**
`PlayerLosses`	**`Tournament`**, **`PlayerName`**

At minimum, we need the tournament name (all functions expect the tournament as the first parameter). The other parameters will vary, depending on the type of question being asked.

First question

Our first question, is "Who won the 2001 French Open?" If we look at our tagged phrases, we get the list of tagged words shown in Figure 13.

Figure 13 – Tagged words



```
1.   [WP]        Who
2.   [VBD]       won
3.   [DT]        the
4.   [YEAR]      2001
5.   [EVENT]     FRENCH OPEN
```

The **`YEAR`** and **`EVENT`** tags let us know which tournament the question is referring to. We have two parameters that we can pass to any of our first three function calls. Since we don't know gender, we will report results from the tournament for both men and women.

The verb is likely to indicate which function we want, the winner or losers of the tournament. Since our verb is the word **`won`**, we should report the winner. Since the question is **`who`**, we know the user is expecting a name.

With this information, we can call the function and generate an answer.

Saving information

One thing we want to do is to create variables to hold the information that the user gives us, that might be reused as parameters. Listing 37 shows the static class variables we declare to remember the parameters.

Listing 37 – Remembered variables

```
private static int TournamentYear;
        private static string Tournament;
        private static string Gender;

        private static string PlayerName;
```

Whenever we determine the parameters from a question text, we update the class variables, so the user doesn't have to repeat themselves.

Initializing the variables

If you can determine them, it could be helpful in the constructor to provide default values for the class variables. For our tennis application, we know there are four tournaments, and they are played in different months. Listing 38 shows us providing tournament and year values, based on the current month. If a user simply asks, "*Who won tennis*?" the system will report results of the appropriate tournament, based on the time on year.

Listing 38 – Default values

```
static TennisMajors() {
            int MM = DateTime.Now.Month;
            if (MM<=2) { Tournament = "AUS"; };
            if (MM>3 && MM <= 5) { Tournament = "FRENCH"; };
            if (MM> 5 && MM <= 7) { Tournament = "WIMBLEDON"; };
            if (MM>7) { Tournament = "USOPEN"; };
            TournamentYear = DateTime.Now.Year;
        }
```

Answering the question

We are now going to create a function that will take the tagged list of words and attempt to generate an answer. Listing 39 shows the function.

Listing 39 – Answer tennis questions

```
static public string GetResponse(List<string> Words_,List<string> Tags_)
{
     string ans_ = "";
```

```
      for (int x = 0; x < Tags_.Count; x++)
      {
          if (Tags_[x] == "YEAR")
          {
             TournamentYear = Convert.ToInt16(Words_[x]);
          }
          if (Tags_[x] == "EVENT")
          {
             Tournament = Words_[x];
             if (Tournament.Contains("FRENCH")) { Tournament = "FRENCH"; }
             if (Tournament.Contains("US ")) { Tournament = "USOPEN"; }
             if (Tournament.Contains("AUS ")) { Tournament = "AUS"; }
          }
          if (Tags_[x].StartsWith("VB"))  { LastVerb = Words_[x].ToUpper();
            }
          if (Tags_[x] == "PERSON")   {  PlayerName = Words_[x].ToUpper();
            }
       }

     if (LastVerb == "WON") {
        if (Gender == "B")
           {
              ans_ = WhoWon(Tournament, TournamentYear, "M");
              string ansW = WhoWon(Tournament, TournamentYear, "F");
              if (ansW.Length>0) { ans_ += " and " + ansW; }
           }
           else
           {
             ans_ = WhoWon(Tournament, TournamentYear, Gender);
           }
        }
     if (LastVerb == "LOST") {
        if (Gender == "B")
           {
              ans_ = WhoLost(Tournament, TournamentYear, "M");
              string ansW = WhoLost(Tournament, TournamentYear, "F");
               if (ansW.Length > 0) { ans_ += " and " + ansW; }
            }
         else
            {
               ans_ = WhoLost(Tournament, TournamentYear, Gender);
             }
          }
      if (ans_.Length < 1) { ans_ = "I don't know..."; }
       return ans_;
   }
```

This basic code processes a couple key verbs (**WON** and **LOST**) and calls the appropriate function to return an answer. It makes a loop through the sentence tags, seeing if it could find parameters to pass along to the calls to narrow down which tournament and year the user is asking about.

Don't be boring

When people answer a question, they might phrase it differently each time. We want our “Who Won” routine to be a bit creative. Listing 40 shows the routine determining the answer, but formatting it differently based on a random selection.

Listing 40 – Who won

```
static public string WhoWon(string Tournament, int Year, string Gender)
 {
     string[] PossibleReplies = {
             "{0} was the {1}'s winner",
             "{0} won the {1}'s",
             "{0} won on the {1}'s side",
             "{0} defeated {2} in the {1}'s draw",
             "{0} won in {3} sets over {2} ",
             "It was won by {0} in {3} sets"
             };
    string ans_ = "";
    Tournament Results_ = GetResults(Tournament, Year, Gender);
    if (Results_ != null)
       {
         string GenderText = "men";
         if (Gender=="F") { GenderText = "women"; }
         int reply = rnd.Next(1, PossibleReplies.Length) - 1;
         // Some tennis vocabulary
         string SetText = Results_.SetsPlayed.ToString();
         if (Results_.SetsPlayed == 3 && Gender=="M")
            { SetText = "straight"; }
         if (Results_.SetsPlayed == 2 && Gender == "F")
           { SetText = "straight"; }

         ans_ = string.Format(PossibleReplies[reply],
                Results_.Winner.FullName,
                GenderText,
                Results_.RunnerUp.FullName,
               SetText);
       }
  return ans_;

}
```

While it is possible to simply extract the answer and return the person's name, the application will appear more friendly and easier to use if it seems more human (in this case, by giving different ways of providing the answer).

Depending on your application, you can really enhance the application by understanding your data. In our case, the score of the match is stored as a string. With a bit of string manipulation, we can make a guess as to how close (or one-sided) the match was.

Speaking the user's language

Knowing your user and application vernacular is helpful and can make your application seem more "human-like." For example, in tennis, if a person wins all the sets without losing a set, it is said to be a straight set victory. If a player wins all the games, they've "bageled" the other player.

Games won

The score string from the data, looks like the following.

6–2, 6–2, 6–2

Listing 41 shows how to determine the games won and the games lost based on the score string.

Listing 41 – Games won

```
static public int GamesWon(string Scores)
{
    int TotalWon = 0;
    string[] Sets_ = Scores.Split(',');
    foreach(string CurSet in Sets_)
      {
           string[] WinLoss = CurSet.Split(',');
           if (WinLoss.Count()==2)
             {
               TotalWon += Convert.ToInt16(WinLoss[0]);
             }
       }
       return TotalWon;
}
static public int GamesLost(string Scores)
{
     int TotalLost = 0;
     string[] Sets_ = Scores.Split(',');
     foreach (string CurSet in Sets_)
       {
         string[] WinLoss = CurSet.Split(',');
         if (WinLoss.Count() == 2)
         {
             // Deal with tiebreakers
```

```
              int x = WinLoss[1].IndexOf("(");
              if (x>0) { WinLoss[1] = WinLoss[1].Substring(0, x - 1); }
                        TotalLost += Convert.ToInt16(WinLoss[1]);
           }
        }
        return TotalLost;
}
```

Games lost is very similar, but deals with the tiebreaker string if it appears. By looking at a match and applying some tennis logic, we can determine if the match was close or not. We might want to adjust our replies even further. Let's determine if the match was one-sided, close, or a good match.

We will add to our stock replies, the following text response.

```
           "{0} won in a {4} match over {2} "
```

For simplicity sake, we use the following formula.

If not straight sets (three for men, or two for women), it means that the winner lost at least one set. So, we can assume that was a close match. If the match was decided in straight sets, and the winner won more than 2-3 times as many games as the loser, we will call that a one-sided match. Listing 42 shows the function to make a rating guess for the match.

Listing 42 – Match rating

```
static public string MatchRate(bool StraightSets,int GamesWon,int
GamesLost)
 {
    string ans_ = "good";
    if(!StraightSets)
      {  ans_ = "close";
         if (GamesWon-GamesLost < 4) { ans_ = "very close"; }
      }
    else
     {
       if (GamesWon > GamesLost*2.5)
         {
           ans_ = "one-sided";
         }
     }
   return ans_;
}
```

With this code added to our functions, the application gets a bit opinionated (Nadal won 6-2,6-3,6-1), as shown in Figure 14.

Figure 14 – Opinionated tennis application

Who won the 2017 French Open?

Rafael Nadal won in a one-sided match over Stan Wawrinka, and Jeļena Ostapenko won in a very close match over Simona Halep.

Of course, the computer knows nothing about the actual matches, just what the data it sees tells it. Be sure to consider your audience as your application gets more creative in its responses.

Second question

The second question is very simple: "*Who lost*?" Since the user has not given us a year or the tournament name, we will rely on the previous answers. From the previous question, we know it was the 2001 French Open, so the system finds the verb **`LOST`**, and has enough information to determine which function to call.

Remembering previous replies will make the system much more friendly to the user. If I enter "*Who is the* Human Resource manager*?*" and the system replies "**Julie**," you can assume my follow-up question of, "What is her email*?*" refers to Julie and can be answered. If your system identifies a person, the pronouns **he** and **she** should be replaced with the person's name in the system's memory.

Third question

The third question is: "Who won the first Wimbledon?" The **`WHO`** question and **`EVENT`** tell us we are looking for a person, but we don't know the year. However, the keyword **`FIRST`** tells us to get the very first year we have data for. So, we scan our word list, looking for first, earliest, etc., keyword-searching the word array.

One of the drawbacks is having to possibly know all the synonyms a person might use. There is a web service available called WordsAPI that allows you to find synonyms for a given word. An example of the JSON response is shown in Listing 43.

Listing 43 – Words API synonym API call

```
{ "word": "first",
   "synonyms":
   [  "1st",
       "inaugural",
        "maiden",
        "kickoff",
        "start",
        "foremost",
   ]
}
```

By using the API, you can anticipate expected words, such as *first* or *latest*, and have the API prebuild possible synonyms. You can also store the list locally in a dictionary. If you are using your own code, you will probably want to keep a synonym dictionary of words likely to be used by your audience.

Final question

The last question is: “Who has won the most Wimbledons?” In this example, we are counting on the tags to identify the event (**`Wimbledon`**) and verb (**`WON`**). We are also expecting the keyword **`most`** (as an adverb or adjective). By detecting the verb (**`WON`** or **`LOST`**) and the modifier **`most`**, we can determine which method in the dataset to call. We can change the question a bit, and still get a reasonable reply, as shown in Figure 15.

Figure 15 - Who lost

```
WHO HAS LOST THE US OPEN THE MOST?
Ivan Lendl has lost 5 times
```

Summary

By parsing the tagged sentence to extract missing data and relying on the verb to guess which function to call, we can generally do a pretty good job of matching input sentences to functions that provide the answer. Again, the more you know about your application, the better you will be able to anticipate the types of questions you might find.

I would suggest, at least initially, keeping a log of the questions asked and answers provided by your application. You will likely keep tweaking your code, based on what the users are asking. As you get a collection of common questions, and tweak the code to answer them, your system will appear smarter every time.

It is possible to simply return the exact answer a person wants, but the system will appear more useful if you use random responses, or even humorous answer to provide the information. We are designing a system to interact with people, so we don't need to be quite as rigid as the protocols needed when we talk between computer systems. People will like the variety and light nature of the responses generated.

Have fun generating responses, but know your audience. If you are designing a system for the military, they might not appreciate a lighter, varying response. (And they carry guns.)

Chapter 9 Cloudmersive

As we mentioned earlier, Cloudmersive offers several web services to save developers from tasks that could be tedious or difficult to do. For example, you can validate that an input from the user looks like an email address. This is a simple Regular Expression. However, there isn't an easy way to confirm that joe@bogus.com is not an actual address.

The Cloudmersive API provides additional web services, to see if that is an actual email address. When I ran the */validate/email/address/full* web service, I received this reply.

```
{

  "ValidAddress": false,

  "MailServerUsedForValidation": "psg.com"

}
```

Be sure to explore the Cloudmersive web service library; the email example is just the tip of the iceberg.

Natural Language services

The website has examples of how to call the API from various languages. In this chapter, we will illustrate how to use the services from C#. Note that you will need to obtain an API key (free for up to 50,000 calls daily) from Cloudmersive. The static class shown in Listing 44 is the basic framework to call these web services.

Listing 44 – Cloudmersive API

```
static public class CloudmersiveNLP
    {
        private static string APIKey = " your key here ";

        static public void SetAPIKey(string myKey)
        {
            if (!string.IsNullOrEmpty(myKey))
            {
                APIKey = myKey;
            }
        }

    }
```

Sentence parsing

Listing 45 shows the code to ask the API to break a paragraph into sentences. The API returns a delimited string; this code breaks that string apart and returns a list of strings (sentences).

Listing 45 - Sentence parsing

```
static public List<string> ExtractSentences(string Paragraph)
{
     Configuration.Default.AddApiKey("Apikey", APIKey);
     var apiInstance = new SentencesApi();
     try
       {
          string result = apiInstance.SentencesPost(Paragraph);
          string[] Sentences_ = result.Replace("\\r", "").
                   Split(new String[] { "\\n" },
                   StringSplitOptions.RemoveEmptyEntries);
          for (int x = 0; x < Sentences_.Length; x++)   {
              Sentences_[x] = Sentences_[x].Replace("\"", "");
          }
          return Sentences_.ToList<string>();
       }
       catch (Exception)
       {
          return null;
       }
}
```

Language detection

Listing 46 shows the code to ask the API to determine the language of a text string.

Listing 46 – Language detection

```
static public LanguageDetectionResponse DetectLanguage(string text)
{
    Configuration.Default.AddApiKey("Apikey", APIKey);
     var apiInstance = new LanguageDetectionApi();   // Which API to call
     try
       {
         LanguageDetectionResponse result =
                 apiInstance.LanguageDetectionPost(text);
          return result;
        }
     catch (Exception)
        {
           return null;
        }
 }
```

Extracting entities

This API attempts to extract entities (people, locations, etc.) from the input sentence. It is shown in Listing 47.

Listing 47 – Extract entities

```
static public string ExtractEntities(string text)
 {
     Configuration.Default.AddApiKey("Apikey", APIKey);
     var apiInstance = new ExtractEntitiesStringApi();
     try
       {
          string result = apiInstance.ExtractEntitiesStringPost(text);
          return result;
       }
       catch (Exception)
       {
          return null;
       }
 }
```

Extracting Words

Listing 48 extracts a list of words from a sentence.

Listing 48 – Get words

```
static public List<string> GetWords(string text)
{
     Configuration.Default.AddApiKey("Apikey", APIKey);
     var apiInstance = new WordsApi();
     try
        {
          string result = apiInstance.WordsGetWordsString(text);
          List<string> Words_ = result.Replace("\"", "").
               Split(',').ToList<string>();
          return Words_;
         }
     catch (Exception)
        {
           return null;
        }
}
```

Tagging words

Cloudmersive uses the Penn Treebank tags, and Listing 49 shows the code that returns a string with the words tagged with the appropriate codes. This web service provides the same type of information as the code we developed in Chapter 5.

Listing 49 – Tagged words

```
static public string TaggedPartofSpeechJSON(string text)
{
     Configuration.Default.AddApiKey("Apikey", APIKey);
     var apiInstance = new PosTaggerStringApi();
     try
        {
            // Part-of-speech tag a string
            var result = apiInstance.PosTaggerStringPost(text);
            return result.ToString();
         }
     catch (Exception)
         {
            return null;
         }
 }
```

Summary

While the web services don't offer Sentiment Analysis or Text Summarization, they do provide a solid set of services you can use to create a tagged list of words for your application to use. Since they are calling web services via a **`POST`**, you need to pass your credentials and check for an exception (typically if the server is unavailable, network is down, etc.).

The source code for the Cloudmersive API calls is included on the book's source code site.

Chapter 10 Google Cloud NLP API

Google provides many API web services, including a set of API calls for processing text. In this chapter, we will discuss how to get set up to use the service, and the service calls available.

You can find the API Documentation here.

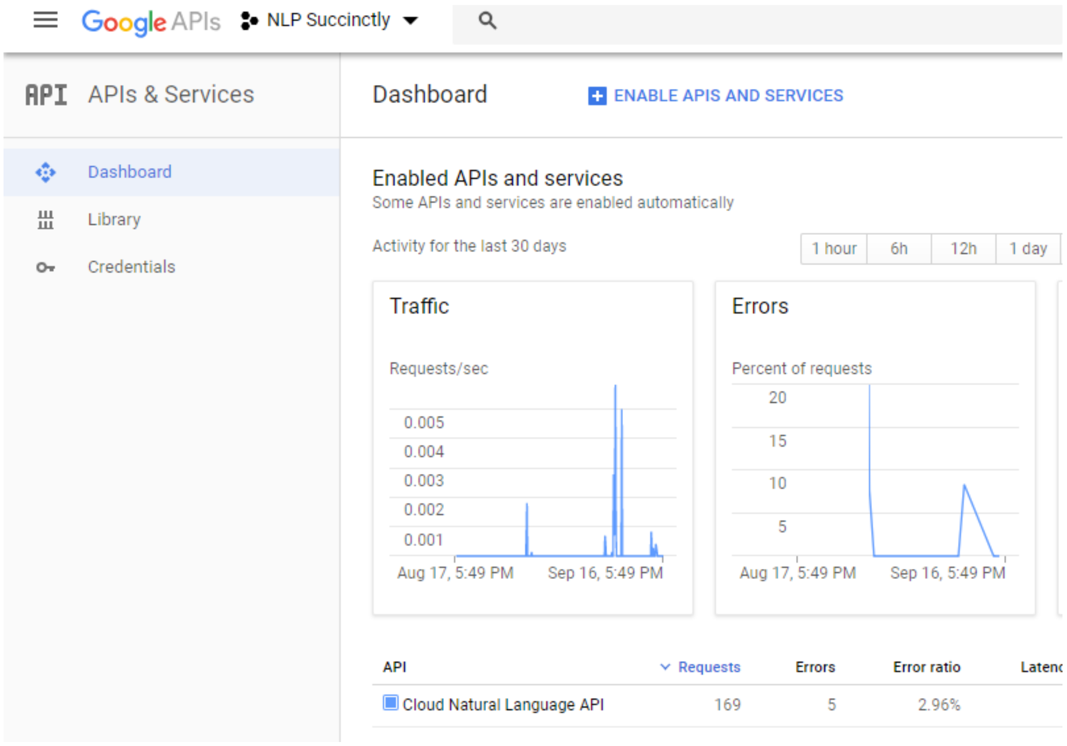
Getting an API key

To get an API key, you must first have a Google account. Visit this site and sign in with your user name and password.

Developers dashboard

The developer's dashboard (Figure 16) provides access to all of Google APIs, which you can explore by clicking on the library icon.

Figure 16 – Google developers dashboard

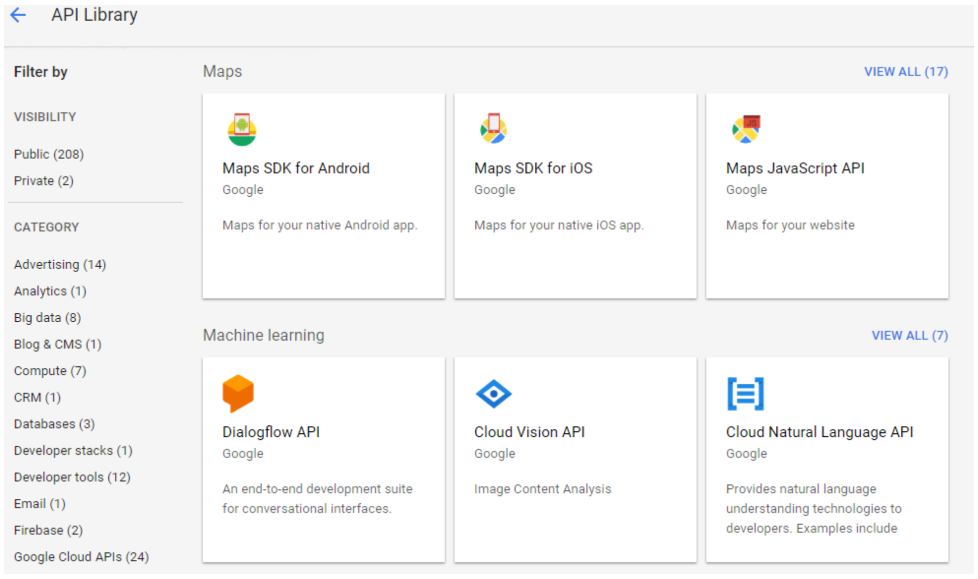


API	Requests	Errors	Error ratio	Latenc
Cloud Natural Language API	169	5	2.96%	

API Library

You will need to create credentials by first finding the Cloud Natural Language API and clicking on it. Figure 17 shows the API library that Google offers.

Figure 17 – Google API library


API Library

Filter by

VISIBILITY

Public (208)
Private (2)

CATEGORY

Advertising (14)
Analytics (1)
Big data (8)
Blog & CMS (1)
Compute (7)
CRM (1)
Databases (3)
Developer stacks (1)
Developer tools (12)
Email (1)
Firebase (2)
Google Cloud APIs (24)

Maps — VIEW ALL (17)

Maps SDK for Android
Google
Maps for your native Android app.

Maps SDK for iOS
Google
Maps for your native iOS app.

Maps JavaScript API
Google
Maps for your website

Machine learning — VIEW ALL (7)

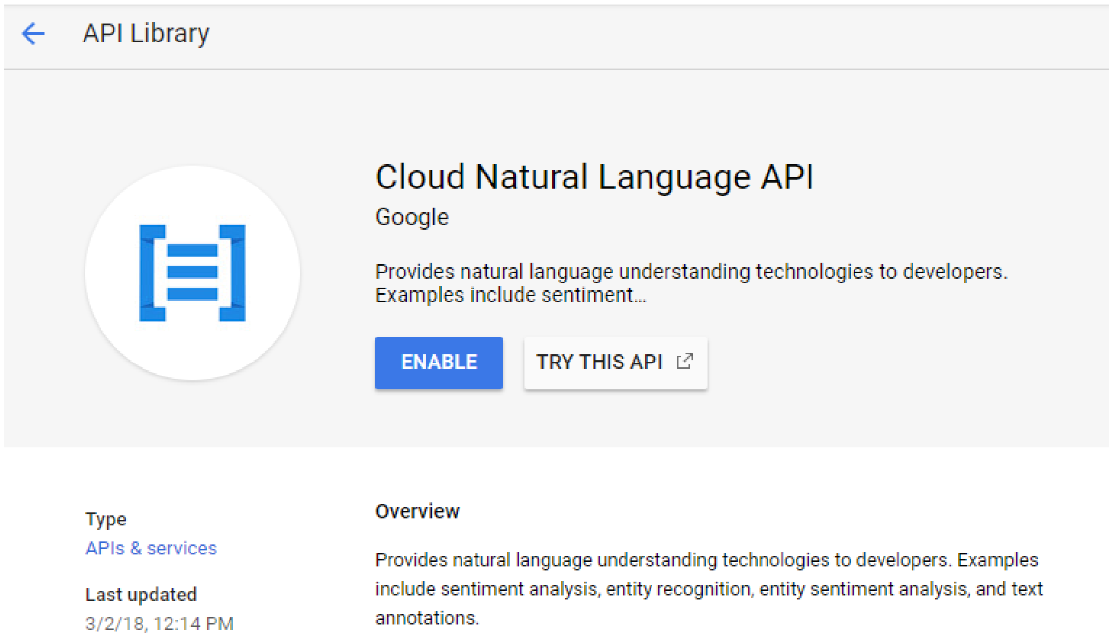
Dialogflow API
Google
An end-to-end development suite for conversational interfaces.

Cloud Vision API
Google
Image Content Analysis

Cloud Natural Language API
Google
Provides natural language understanding technologies to developers. Examples include

Select the API by clicking on the box, shown in Figure 18. Click **Enable** to enable the API.

Figure 18 – Enable your API


API Library

Cloud Natural Language API
Google

Provides natural language understanding technologies to developers. Examples include sentiment...

ENABLE   TRY THIS API

Type
APIs & services

Last updated
3/2/18, 12:14 PM

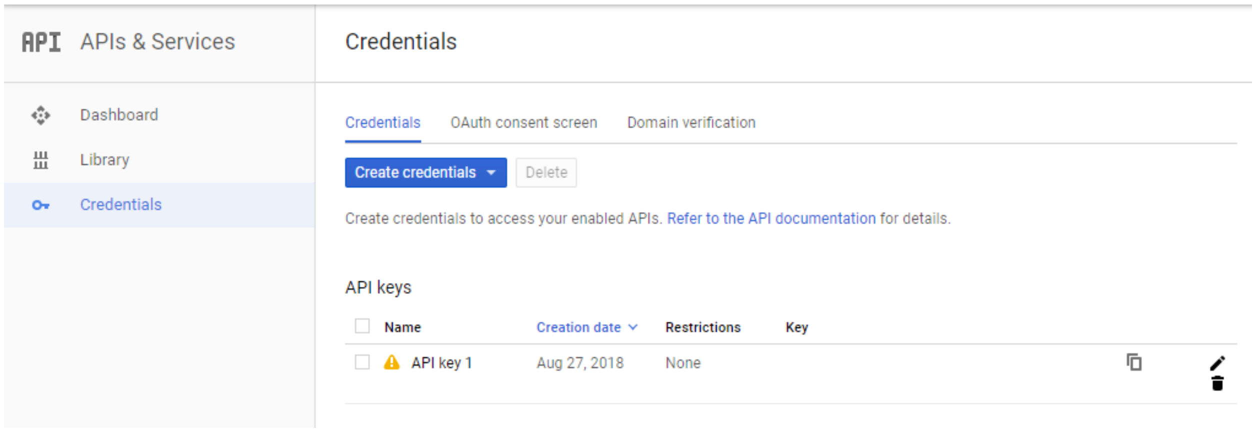
Overview

Provides natural language understanding technologies to developers. Examples include sentiment analysis, entity recognition, entity sentiment analysis, and text annotations.

Creating credentials

Once you've picked the API, you'll need to create credentials for using the API. Once you do, you'll be given an API, which you'll need in your code. (I've hidden mine here.)

Figure 19 – Google credentials



Create credentials to access your enabled APIs. Refer to the API documentation for details.

API keys

Name	Creation date	Restrictions	Key
API key 1	Aug 27, 2018	None	

That's it. Save that API key, and you are ready to start interfacing with Natural Language API.

Creating a Google NLP class

We are going to write a static class that will have private methods handling the interaction with the Google APIs and public methods to expose the APIs you want to use within your application.

Basic class

Listing 50 is the basic class to call the Google APIs. Be sure to set the **`GOOGLE_KEY`** variable to your API obtained previously.

Listing 50 – Base Google class

```
static public class GoogleNLP
{
  static private string BASE_URL =
        "https://language.googleapis.com/v1/documents:";
  static private string GOOGLE_KEY = " your API key ";

  static private dynamic BuildAPICall(string Target,string msg)
  {
      dynamic results = null;
      string result;
      HttpWebRequest NLPrequest = (HttpWebRequest)WebRequest.
                Create(BASE_URL + Target+"?key=" + GOOGLE_KEY);
      NLPrequest.Method = "POST";
      NLPrequest.ContentType = "application/json";
      using (var streamWriter = new
             StreamWriter(NLPrequest.GetRequestStream()))
      {
          string json = "{\"document\": {\"type\":\"PLAIN_TEXT\"," +
                        "\"content\":\"" + msg + "\"} }";
          streamWriter.Write(json);
          streamWriter.Flush();
          streamWriter.Close();
      }
      var NLPresponse = (HttpWebResponse)NLPrequest.GetResponse();
      if (NLPresponse.StatusCode == HttpStatusCode.OK)
      {
           using (var streamReader = new
                  StreamReader(NLPresponse.GetResponseStream()))
           {
                result = streamReader.ReadToEnd();
           }
           results = JsonConvert.DeserializeObject<dynamic>(result);
       }
       return results;
 }
```

Adding public methods

With this class built, the public methods to expose the methods are all simply wrapper calls. Listing 51 shows the public methods to call the Google API calls directly.

Listing 51. – Expose Google API calls

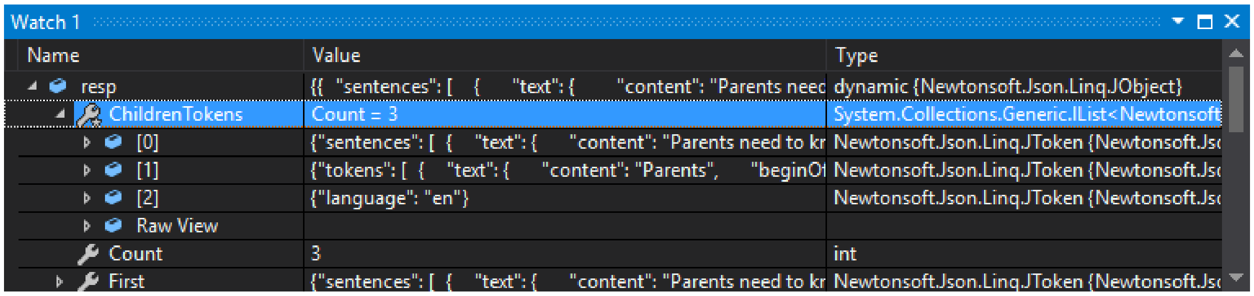
```
static public dynamic analyzeEntities(string msg)
        {
            dynamic ans_ = BuildAPICall("analyzeEntities", msg);
            if (ans_ == null) return null;
            return ans_;
        }

static public dynamic analyzeSentiment(string msg)
        {
            dynamic ans_ = BuildAPICall("analyzeSentiment", msg);
            if (ans_ == null) return null;
            return ans_;
        }

static public dynamic analyzeSyntax(string msg)
        {
            dynamic ans_ = BuildAPICall("analyzeSyntax", msg);
            if (ans_ == null) return null;
            return ans_;
        }

static public dynamic classifyText(string msg)
        {
            dynamic ans_ = BuildAPICall("classifyText", msg);
            if (ans_ == null) return null;
            return ans_;
        }

static public dynamic analayzeEntitySentiment(string msg)
        {
            dynamic ans_ = BuildAPICall("analyzeEntitySentiment", msg);
            if (ans_ == null) return null;
            return ans_;
        }
```

These calls return dynamic objects, which you can further parse to pull out information. The Google documentation provides details on the response data returned via the API.

Viewing the response data

The data coming back from the direct API calls will be a dynamic JSON object. You can explore the object in the debugger to see what properties are being returned. Figure 20 shows the debugger snapshot from the **`AnalyzeSyntax`** call.

Figure 20 – Response from API



Watch 1

Name	Value	Type
resp	{{ "sentences": [{ "text": { "content": "Parents need	dynamic {Newtonsoft.Json.Linq.JObject}
ChildrenTokens	Count = 3	System.Collections.Generic.IList<Newtonsoft
[0]	{"sentences": [{ "text": { "content": "Parents need to kr	Newtonsoft.Json.Linq.JToken {Newtonsoft.Js
[1]	{"tokens": [{ "text": { "content": "Parents", "beginO	Newtonsoft.Json.Linq.JToken {Newtonsoft.Js
[2]	{"language": "en"}	Newtonsoft.Json.Linq.JToken {Newtonsoft.Js
Raw View		
Count	3	int
First	{"sentences": [{ "text": { "content": "Parents need to kr	Newtonsoft.Json.Linq.JToken {Newtonsoft.Js

You can access any of the properties of the object by name. For example, using the syntax **`resp.language.Value`** will return the string **`en`**. The syntax **`resp.sentences.Count`** will return the number of sentences in the response, and **`resp.sentences[x]`** will let you access the individual items in the **`sentences`** collection.

Wrapping the calls

In Chapter 2, we introduced a function to return a list of sentences from a paragraph. Listing 52 shows a wrapper function around the Google **`analyzeSyntax`** call that returns a list of strings.

Listing 52 – Extract sentences

```
static public List<string> GoogleExtractSentences(string Paragraph)
 {
     List<string> Sentences_ = new List<string>();
     var ans_ = analyzeSyntax(Paragraph);
     if (ans_ !=null)
        {
           for(int x=0;x<ans_.sentences.Count;x++)
           {
               Sentences_.Add(ans_.sentences[x].text.content.Value);
            }
         }
         return Sentences_;

 }
```

In Chapter 5, we introduced the concept of tagging words, using the Penn Treebank tag set. (See Appendix A.) Google uses a different set of tags, so we need to do some conversion work to map the Google tags to the Penn Treebank set.

Google tags

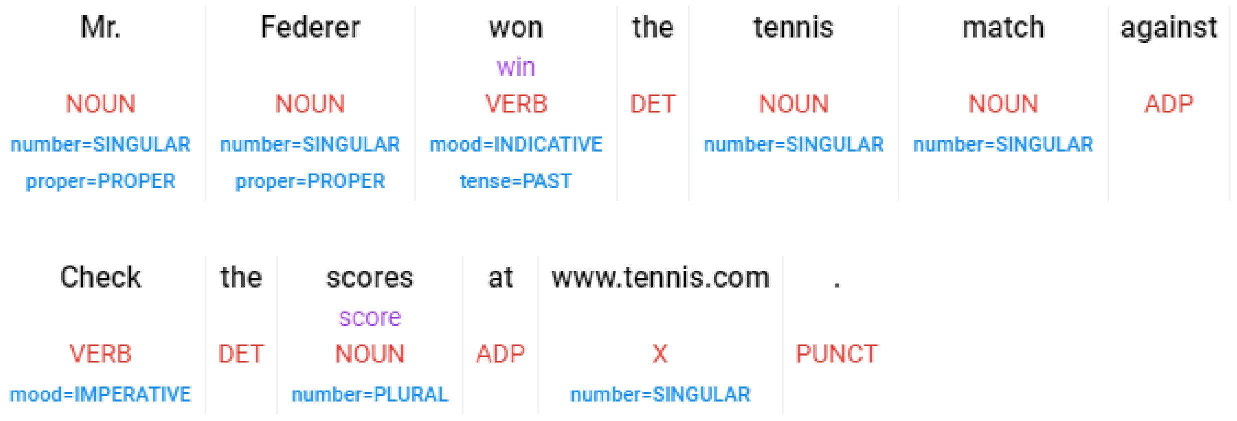
Google's API returns a token for each word in the parsed word list. These tags are based on the Universal tag list (see Appendix B). Each token has a part of speech component, which has 12 properties. We need to map these properties to the appropriate Penn Treebank tag. Listing 53 shows the code to get a list of tagged words from the Google API (by converting the Google **`partOfSpeech`** information to a Penn Tree Bank tag).

Listing 53 – Google tagged words

```
static public List<string> GoogleTaggedWords(string Sentence,bool
includeLemma=false)
{
   List<string> tags = new List<string>();
   var ans_ = analyzeSyntax(Sentence);
   if (ans_ != null)       {
         for (int x = 0; x < ans_.tokens.Count;x++)          {
            string Tag =
GoogleTokenToPennTreebank(ans_.tokens[x].partOfSpeech,
                           ans_.tokens[x].text.content.Value.ToLower());
            if (includeLemma)     {
                   Tag += ":" + ans_.tokens[x].lemma.Value;
               }
           tags.Add(Tag);
         }
      }
      return tags;
}
```

We've included an option to return the lemma. A lemma is the base form of a word, so that wins, won, winning, etc. would all have a lemma of `win`. Adding a lemma value simplifies the task of determine the concept the verb or noun represents. Google's API includes a lemma, so the method can append the lemma to the returned tag. Figure 21 shows a sample syntax analyze call from the API, including lemmas.

Figure 21 – Sample Google API syntax analysis



Mr.	Federer	won	the	tennis	match	against
		win				
NOUN	NOUN	VERB	DET	NOUN	NOUN	ADP
number=SINGULAR	number=SINGULAR	mood=INDICATIVE		number=SINGULAR	number=SINGULAR	
proper=PROPER	proper=PROPER	tense=PAST				

Check	the	scores	at	www.tennis.com	.
		score			
VERB	DET	NOUN	ADP	X	PUNCT
mood=IMPERATIVE		number=PLURAL		number=SINGULAR	

Listing 54 shows the code to convert the Universal Part of Speech tag to a Treebank code, relying on the Universal tag and some of the other fields available with the 12 properties returned for each word.

Listing 54 – Mapping part of speech tags

```
static private Dictionary<string, string> TagMapping = new
Dictionary<string, string>
    {
        {"ADJ","JJ" },
        {"ADP","IN" },
        {"ADV","RB" },
        {"CONJ","CC" },
        {"DET","DT" },
        {"NOUN","NN" },
        {"NUM","CD" },
        {"PRON","PRP" },
        {"PUNCT","SYM" },
        {"VERB","VB" },
        {"X","FW" },
    };


static private string GoogleTokenToPennTreebank(dynamic Token)
{
    string ans = "";
    string GoogleTag = Token.tag.Value.ToUpper();
    if (TagMapping.ContainsKey(GoogleTag))      {
        ans = TagMapping[GoogleTag];
    }
    // Tweak certain tags
    if (ans=="NN")
      {
        string number = Token.number.Value.ToUpper();
        string proper = Token.proper.Value.ToUpper();

        if (number == "PLURAL" && proper == "PROPER" )
              { ans = "NNPS"; }    // Plural proper noun
        if (number == "SINGULAR" && proper == "PROPER")
              { ans = "NNP"; }     // Singular proper noun
        if (ans=="NN" && number=="PLURAL")
              { ans = "NNS"; }               // Plural common noun
      };
    if (ans=="VB")
      {
        string tense = Token.tense.Value.ToUpper();
        string person = Token.person.Value.ToUpper();
        if (tense == "PAST") { ans = "VBD"; }    // Past tense verb
        if (tense == "PRESENT") { ans = "VBG"; }    // Present tense verb
      };
  return ans;

 }
```

The code takes the **`Token`** object and maps it to the appropriate Treebank tag. However, there is additional information in the Google token, to let us map further, determining the type of noun and the case of verb.

Summary

The Google Cloud API provides a good set of NLP API calls, and can be helpful to build your tagged list of words. The API also offers categorization, named entity recognition, and sentiment analysis, all handy features when processing natural language text.

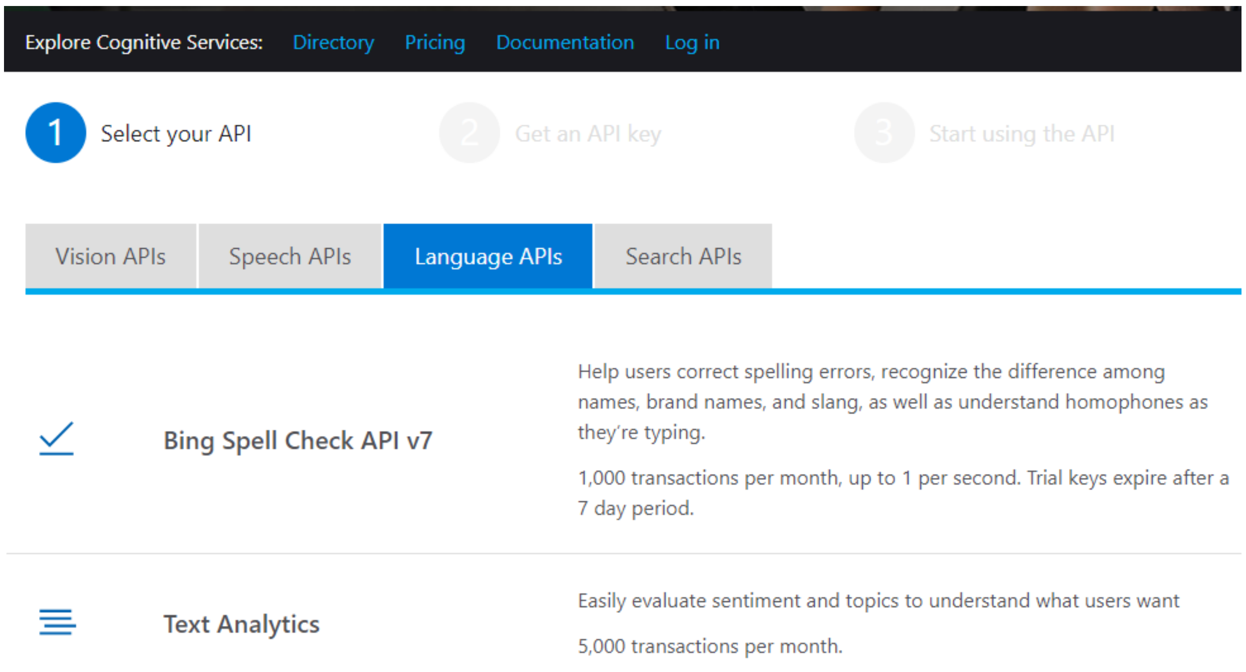
Chapter 11 Microsoft Cognitive Services

Microsoft Cognitive Services are a variety of machine learning algorithms developed by Microsoft to solve Artificial Intelligence (AI) problems. There are number of services available; we will be exploring the text analytics services in this chapter.

Getting started

To get started with Microsoft Cognitive services, you will first need to visit this site and sign up for an API key.

Figure 22 - Cognitive Services



Bing Spell Check API v7

Help users correct spelling errors, recognize the difference among names, brand names, and slang, as well as understand homophones as they're typing.

1,000 transactions per month, up to 1 per second. Trial keys expire after a 7 day period.

Text Analytics

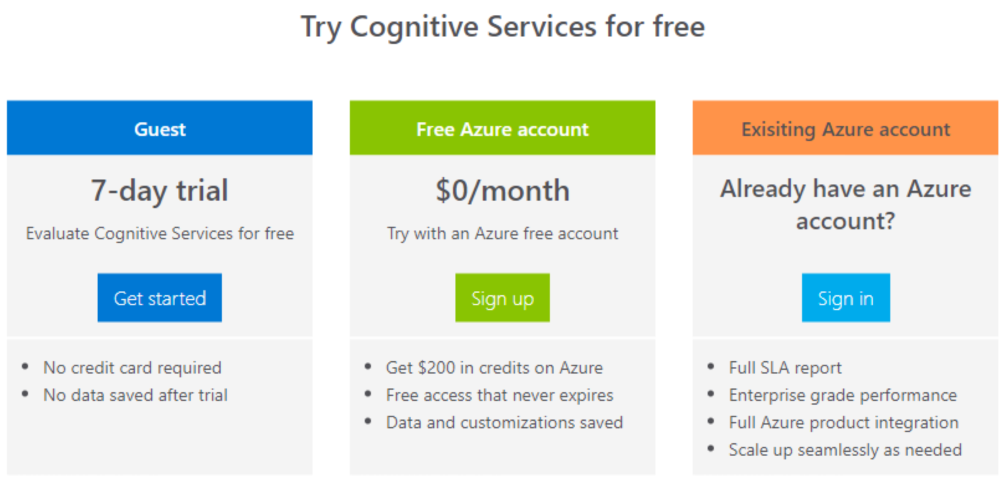
Easily evaluate sentiment and topics to understand what users want

5,000 transactions per month.

Select the Text Analytics option

If you have an Azure account already, you can add cognitive services to that account; otherwise, to first take a look, select the **Guest 7-day trial** option shown in Figure 23.

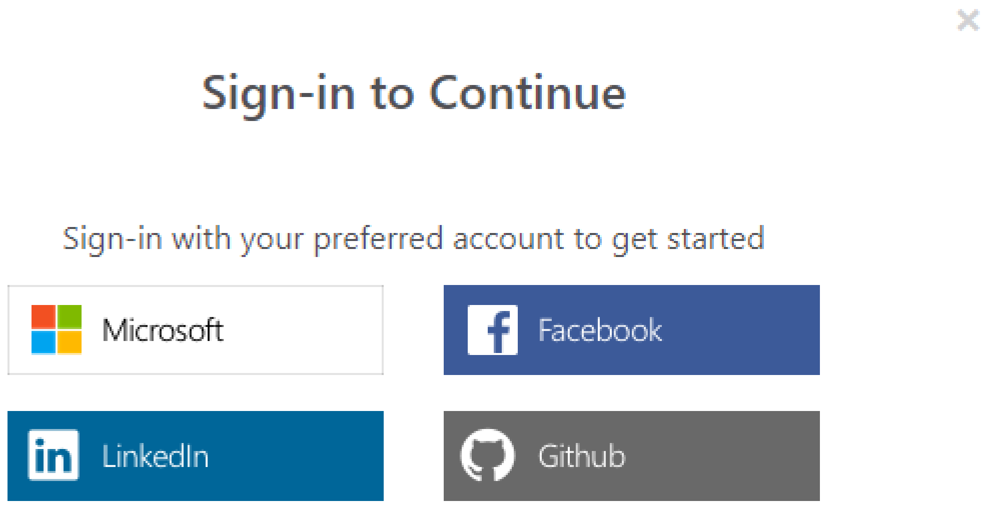
Figure 23 – Cognitive Services sign-up



Try Cognitive Services for free

Guest	Free Azure account	Exisiting Azure account
7-day trial	$0/month	Already have an Azure account?
Evaluate Cognitive Services for free	Try with an Azure free account	
Get started	Sign up	Sign in
• No credit card required • No data saved after trial	• Get $200 in credits on Azure • Free access that never expires • Data and customizations saved	• Full SLA report • Enterprise grade performance • Full Azure product integration • Scale up seamlessly as needed

Figure 24 – Sign in



Sign-in to Continue

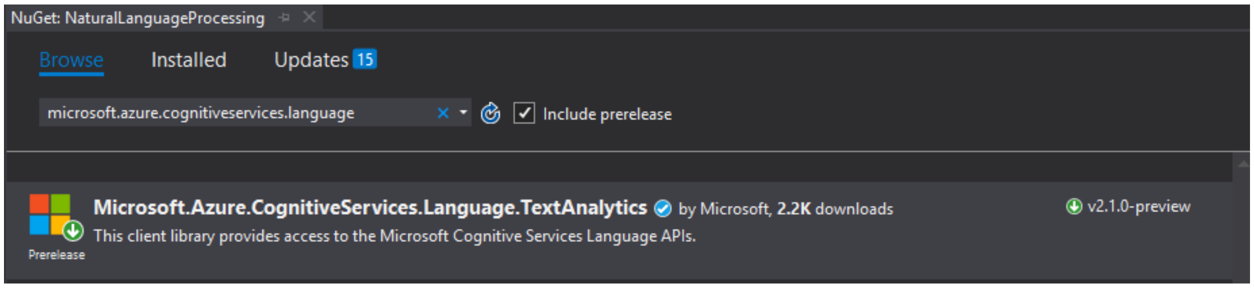
Sign-in with your preferred account to get started

Once you've signed in, you will be provided an endpoint and an API key. Save these two, because you'll need them to retrieve data via the API.

Installing Cognitive Services

The Microsoft documentation for installing Cognitive Services can be found here. This page describes how to install the API for many different programming languages. Figure 25 shows installing the NuGet SDK package into your C# solution or project.

Figure 25 – NuGet package


NuGet: NaturalLanguageProcessing

microsoft.azure.cognitiveservices.language

Microsoft.Azure.CognitiveServices.Language.TextAnalytics by Microsoft, 2.2K downloads v2.1.0-preview

This client library provides access to the Microsoft Cognitive Services Language APIs.

Communicating with the web service

Once you have your credentials (Endpoint and API Key), you can call the services and return the data for use in your application. We are going to call the web services directly for simplicity purposes (very similar to our approach to calling the Google API). Listing 55 shows the basic setup of the `static` class to use the Cognitive Services web API.

Listing 55 – Web service call to Cognitive Services

```
static public class MicrosoftNLP
{

    static private string API_KEY = "";
    static private string ENDPOINT = ""
    static private dynamic BuildAPICall(string Target, string msg)
   {
       dynamic results = null;
      string result;
        HttpWebRequest NLPrequest = (HttpWebRequest)WebRequest.
               Create(ENDPOINT + "/"+Target );
        NLPrequest.Method = "POST";
        NLPrequest.ContentType = "application/json";
        NLPrequest.Headers.Add("Ocp-Apim-Subscription-Key", API_KEY);

        using (var streamWriter = new
StreamWriter(NLPrequest.GetRequestStream()))
        {
            string json = "{\"documents\": [ {\"id\":\"1\"," +
                           "\"text\":\"" + msg + "\"}] }";
            streamWriter.Write(json);
            streamWriter.Flush();
            streamWriter.Close();
        }
        var NLPresponse = (HttpWebResponse)NLPrequest.GetResponse();
        if (NLPresponse.StatusCode == HttpStatusCode.OK)
        {
         using (var streamReader = new
             StreamReader(NLPresponse.GetResponseStream()))
```

```
           {
               result = streamReader.ReadToEnd();
             }
           results = JsonConvert.DeserializeObject<dynamic>(result);
          }
         return results;

   }
```

This is the basic code to reach the endpoint and pass the data along to Cognitive Services. Be sure to set the **`ENDPOINT`** and **`API_KEY`** to the ones you obtained from the registration. This method returns the result object as a JSON object; you will need to have your method calls extract whatever in information your application needs.

Listing 56 is a number of wrapper calls to extract the JSON information for use in your application.

Listing 56 – Wrapper calls to Cognitive Services

```
static public string DetectLanguage(string text)
        {
            var ans = "";
            var results = BuildAPICall("languages", text);
            try
            {
               ans =
results.documents[0].detectedLanguages.First.name.Value;
            }
            catch
            {
               ans = "UNKNOWN";
            }
            return ans.ToString();
        }

        static public double AnalyzeSentiment(string msg)
        {
            double ans = 0;
            var results = BuildAPICall("sentiment", msg);
            try
            {
                ans = Convert.ToDouble(results.documents[0].score);
            }
            catch
            {
                ans = -1;
            }
            return ans;
        }
       static public List<string> classifyText(string text)
        {
            List<string> ans = new List<string>();
            var results = BuildAPICall("keyPhrases", text);
            try
            {
                foreach (var curPhrase in results.documents[0].keyPhrases)
                {
                    ans.Add(curPhrase.Value);
                }
            }
            catch
            {
            }
            return ans;
        }
        static public List<string> analyzeEntities(string text)
        {
            List<string> ans = new List<string>();
            var results = BuildAPICall("entities", text);
            try
            {
                foreach (var curEntity in results.documents[0].entities)
                {
                    ans.Add(curEntity.name.Value);
                }
            }
            catch
            {
            }
            return ans;

        }
```

These calls make the API call and extract the returned data, either as a string, a list, or other object, depending on your application needs.

Summary

Cognitive Services is a helpful API from Microsoft that handles some of the trickier issues in Natural Language processing. You can explore the text analytics offered via these APIs to enhance your application.

Chapter 12 Other NLP uses

In the previous chapters, we explored some of the basic functions and code to create a simple question-answering NLP application. The goal of the book was to get the reader comfortable with understanding how to parse sentences to create a tagged word list, and then use that tagged word list to answer English-language questions.

Answering questions is just the tip of the iceberg. Knowing the words and a reasonable set of tags helps the programs attempt to discern meaning. Here are some other applications of Natural Language Processing, which can be accessed through the various APIs we talked about in earlier chapters.

Language recognition

As the world continues to get smaller, most application cannot simply assume the user is entering English text. For example, if any application allows user to send tech support requests via email or text message, you might get some text as follows:

- Mi equipo email mantiene colgados
- Il mio computer email mantiene appeso

Since you would like to send the support to a person who speaks the language, you would first need to discern the language. Most of the APIs handle this task for you, by returning the language it thinks the text is in. Many European languages were derived from Latin, so discerning the actual language is a tricky task. Leave this one to the big guys. 😊

The Microsoft Cognitive services API can detect over 100 different languages. For example, say we give the API the request shown in Listing 57.

Listing 57 – Request for language identification

```
"documents": [

    {"id":"1","text": "Hello world"     },

    {"id":"2","text": "Bonjour tout le monde"     },

    {"id":"3","text": "La carretera estaba atascada.

                        Había mucho tráfico el día de ayer."    } ]
```

The API will return the results shown in Listing 58.

Listing 58 – Results of language detection

```
"documents": [
  { "id": "1",  "detectedLanguages":
   [{ "name": "English", "iso6391Name": "en", "score": 1.0 }] },
  { "id": "2", "detectedLanguages":
   [{ "name": "French", "iso6391Name": "fr",  "score": 1.0 }] },
  { "id": "3", "detectedLanguages":
   [{ "name": "Spanish", "iso6391Name": "es", "score": 1.0 }]
]
```

The API returns the language, the language code, and a score indicating the confidence in its language determination.

Sentiment analysis

Sentiment analysis attempts to discover the attitude a piece of text is conveying. It is a positive text or negative? Such knowledge would be very helpful for a business trying to keep in touch with their customers. Social media and online reviews provide a great source of customer feedback, and companies would do well to listen to it.

Unfortunately, it is not a simple task. While some sentences are clear, there are many times where the meaning is not. For example, is the following sentence positive or negative?

I really enjoyed the movie, but the ending was unsettling.

Even particular words can be better interpreted with knowledge of the domain. For example, if I see the word “thin”, and I am talking about cell phones, this is probably a positive comment. However, if I am referring to my hotel room, thin sheets or thin walls would generally be negative comments.

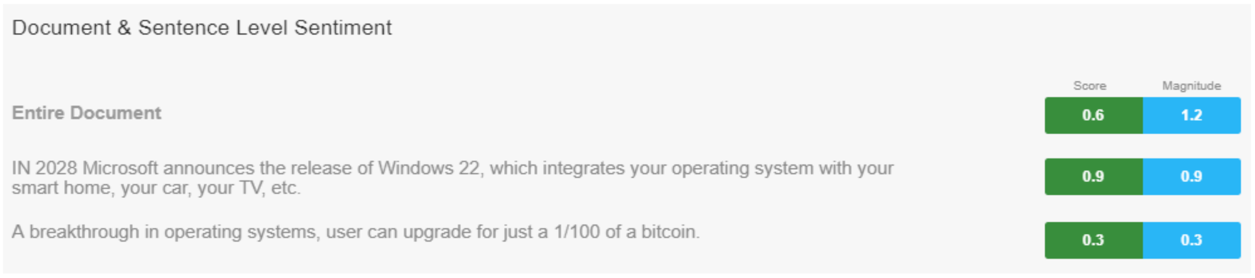
Both Microsoft Cognitive Services and Google Cloud NLP offer sentiment analysis API calls.

Google sentiment analysis

You can demo Google's sentiment analysis here.

Figure 26 shows a sample press release and Google's analysis of it.

Figure 26 – Google sentiment analysis


Document & Sentence Level Sentiment

	Score	Magnitude
Entire Document	0.6	1.2
IN 2028 Microsoft announces the release of Windows 22, which integrates your operating system with your smart home, your car, your TV, etc.	0.9	0.9
A breakthrough in operating systems, user can upgrade for just a 1/100 of a bitcoin.	0.3	0.3

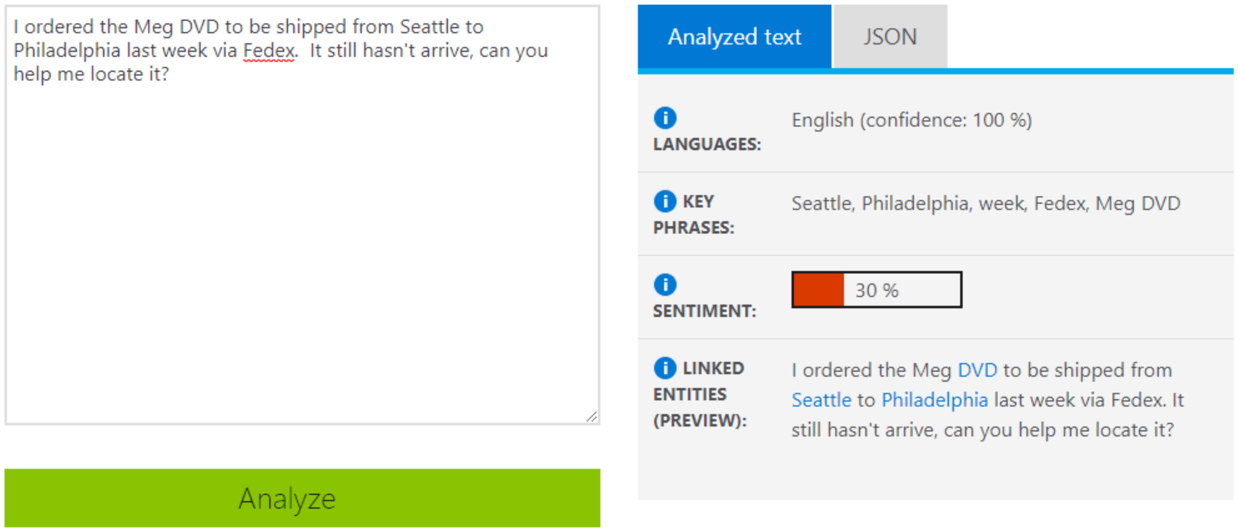
The score ranges from -1 to 1; the lower the number, in general, the more negative the text is perceived. In this case, a score of 0.6 indicates a pretty positive press release. The magnitude, which ranges from 0 to infinity, indicates how strong the emotion is. Based on Google's analysis, I can't wait for Windows 22.

Categorization

Imagine a corporation that sells many different products and services. They have an NLP system to process received emails. Knowing the general category that the email is discussing could help them direct the message to the appropriate department for processing. The larger the domain space (that is, number products and services), the more difficult it becomes to extract categories from text messages.

Microsoft Cognitive Services provide a means to read through a text and extract the various objects the document is talking about. Figure 27 shows an example of a customer complaint email and the information that the Cognitive Services API gleaned from it.

Figure 27 – Cognitive Services at work


I ordered the Meg DVD to be shipped from Seattle to Philadelphia last week via Fedex. It still hasn't arrive, can you help me locate it?

LANGUAGES:	English (confidence: 100 %)
KEY PHRASES:	Seattle, Philadelphia, week, Fedex, Meg DVD
SENTIMENT:	30 %
LINKED ENTITIES (PREVIEW):	I ordered the Meg DVD to be shipped from Seattle to Philadelphia last week via Fedex. It still hasn't arrive, can you help me locate it?

You can use to the code from Chapter 11 to gather this information into JSON objects so your code can decide what actions to take.

Summary

Using the APIs discussed in the next few chapters will assist you in adding NLP to your application. The problems mentioned in this chapter are handled via the APIs, so while it can be intellectually fun and challenging to think about, I would suggest letting the big guys deal with the nuances if your goal is simply to enhance your application by allowing English questions.

Chapter 13 Summary

Natural Language Processing is an ambitious field. The idea of being able to converse with a computer program—once the realm of science fiction—is getting closer all the time. In this book, we touched upon how to use Natural Language Processing in your code. The code in the early part of the book illustrates the principal components, but the API services from the “big guys” offer a quick way to implement those concepts.

Once you have parsed and tagged list of words, your application still needs to decide what to do with them, but ideally, allowing your users to communicate with your systems in English would be a nice accomplishment.

The list of words and tags (and any other information) is the starting point to most text analytics usages. The code in this book and the various API calls all provide methods to build that list. If your domain is limited and you know the data and response very well, you can use the limited code here to process and interpret the text sentences. If you are trying to create a more general application, I suggest using the APIs and leaving the hard work to someone else.

Keep in the mind that the more you know about your domain and usage, the less ambiguous the user questions will be, so spend a lot of time getting to know your data and users. Now, I must go to play tennis with Roger. My scheduling app set it up for me—I just hope the application knows I meant my friend Roger.

Appendix A Penn Treebank tags

The Penn Treebank project provides a list of 36 tags used to classify words. Many NLP web services use these tags. The complete list is shown in Table 10. The official site can be found here.

Table 9 – Penn Treebank tag list

Tag	Meaning	Examples
CC	Coordinating conjunction	and, or
CD	Cardinal number	1,2, one, million
DT	Determiner	the, an
EX	Existential there	there
FW	Foreign word	mucho tráfico
IN	Preposition	in, to, during
JJ	Adjective	good, amazing, fast
JJR	Adjective-comparative	better, faster, braver
JJS	Adjective-superlative	best, fastest, bravest
LS	List item marker	1), a) II.
MD	Modal	could, would, should
NN	Noun-singular	man, woman, car, airplane
NNS	Noun-plural	men, women, children, cars, airplanes
NNP	Proper noun-singular	Washington, James
NNPS	Proper noun-plural	Phillies, Eagles
PDT	Predeterminer	both
POS	Possessive ending	's
PRP	Personal pronoun	her, him, them, us
PRP$	Possessive pronoun	her, his, mine, ours, my, your
RB	Adverb	swiftly, slowly, magically
RBR	Adverb-comparative	grander, louder, lower
RBS	Adverb-superlative	best, loudest, quickest
RP	Particle	up
SYM	Symbol	. % & @ (
TO	to	to
UH	Interjection	golly, heck, amen, shucks
VB	Verb-base form	play, win, eat
VBD	Verb-past tense	played, won, ate
VBG	Verb-present	playing, winning, eating
VBN	Verb-past participle	been, was
VBP	Verb-non-3rd person singular present	take, love
VBZ	Verb-3rd person singular present	takes, likes
WDT	Wh-determiner	that, whatever,
WP	Wh-pronoun	which, who, whom
WP$	Possessive wh-pronoun	whose
WRB	Wh-adverb	where, when

Appendix B Universal POS tags

The Universal Part of Speech tag set (used by Google), is smaller than the Penn Treebank tags. Table 11 shows the list of Universal POS tags (see http://universaldependencies.org/u/pos/ for more information.

Table 10 – Universal POS tags

Tag	Meaning	Examples
ADJ	Adjective	big, old, first, red
ADP	Adposition (prepositions)	in, to, during
ADV	Adverb	very, exactly, socially
AUX	Auxiliary	has, will, was
CCONJ	Coordinating conjunction	and, or, but
DET	Determiner	a, an, the, this
INTJ	Interjection	psst, golly, bravo, heck
NOUN	Noun	man, boy, tree, car, girl
NUM	Numeral	1, three, thousand, 3.14
PART	Particle	's, not
PRON	Pronoun	I, you, we, himself, them, somebody
PROPN	Proper noun	Mary, Washington,
PUNCT	Punctuation	Period(.), Comma(,), Semi-colon (;)
SCONJ	Subordinating conjunction	if, like, then, that
SYM	Symbol	$, %, @ , +
VERB	Verb	run, eat, walking, sit
X	Other	Any word that doesn't fit into the other categories

Appendix C About the code

The code for the book consists of a single solution with three projects. The primary project is the Natural Language Processing class library, which contains the code samples discussed in this book. You can also find them at the Syncfusion GitHub repository.

Natural Language Processing

This is a class library containing classes to perform Natural Language methods. The primary three classes are:

- **`NLP`**: Core library to parse sentences, and words.
- **`Tagger`**: Takes the word list and generates tags for each word
- **`Entities`**: Processes the tagged word list, assigning entities where needed.

You will likely need to review the dictionary of words, regular expressions, etc. in **`tagger`** and **`Entities`** to customize them for your application needs.

There are also three wrapper classes that interact with the various APIs we've discussed throughout the book. You will need to obtain keys and endpoints, and then update these API classes with your credentials. These classes are:

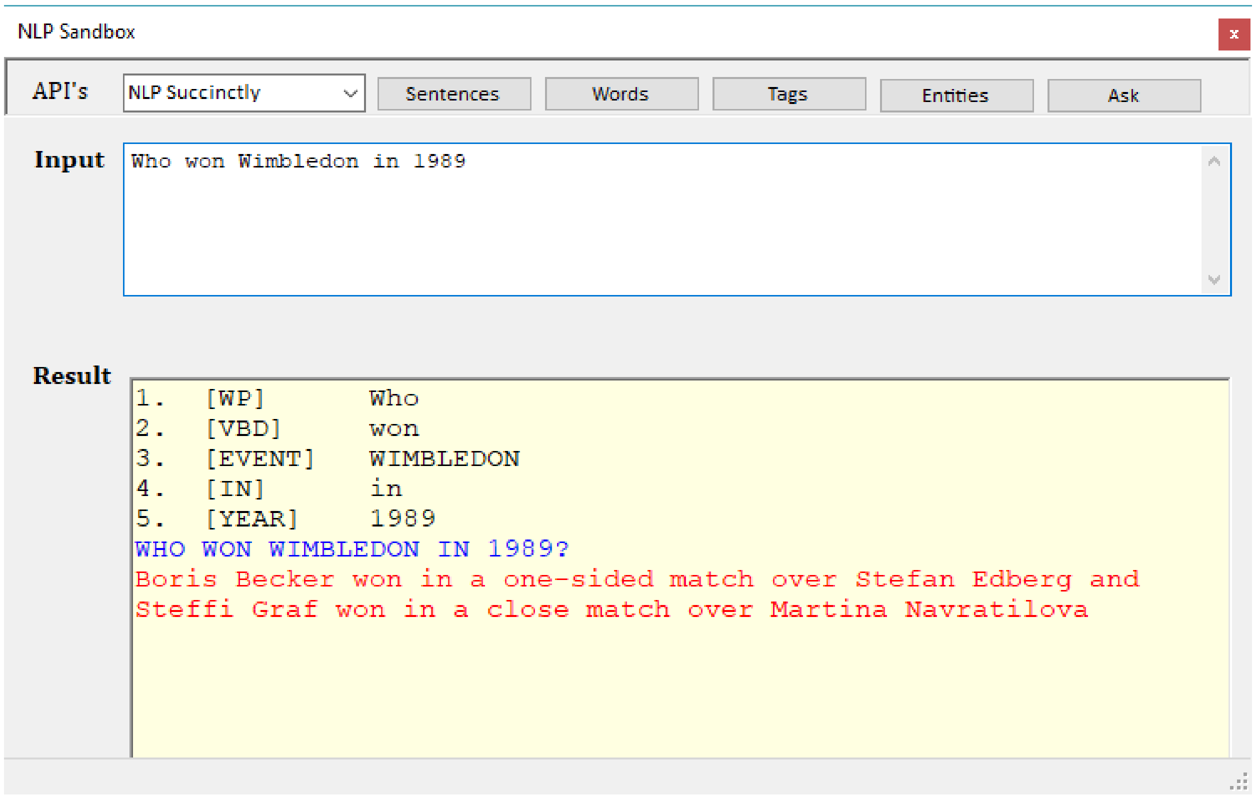
- **`CloudmersiveNLP`**
- **`GoogleNLP`**
- **`MicrsoftNLP`**

All of the source files are available at https://github.com/SyncfusionSuccinctlyE-Books/Natural-Language-Processing-Succinctly.

Playground

This project is simply a Windows Forms application that allows you to enter a sentence and see the results of the various NLP operations. It simply looks for an input text, and then performs the requested API call. The API dropdown list allows you to specify which API (or the book's internal code) to perform the request action (see Figure 28).

Figure 28 - Playground



```
Who won Wimbledon in 1989
```

```
1.   [WP]        Who
2.   [VBD]       won
3.   [EVENT]     WIMBLEDON
4.   [IN]        in
5.   [YEAR]      1989
WHO WON WIMBLEDON IN 1989?
Boris Becker won in a one-sided match over Stefan Edberg and
Steffi Graf won in a close match over Martina Navratilova
```

Tennis Data

The Tennis Data project is the code to load the data, and using LINQ queries, to create functions that answer the questions the user might ask. You will likely write your own version, using SQL, Entity Framework, etc.