Exploring NLP with Python

Building Understanding Through Code

Dr. Karen Mazidi

Copyright © 2021 Karen Mazidi

WWW.KARENMAZIDI.COM

This work is released under Creative Commons License CC BY-NC-ND 4.0

You are free to use this work for your personal educational purposes. You may not redistribute any portion of this work without prior permission of the author.

First Edition, July 2019

Created with a LaTeX template by Mathias Legrand `http://www.LaTeXTemplates.com`

Contents

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Preface to the Book

My interest in Natural Language Processing built over a lifetime. As a child, I loved reading more than anything. On long boring summer days of the electronics-free 1970s of my childhood, I would ride my bike to the little old Carnegie library in Cleburne, Texas. I would spend hours in the cool musty air, dodging the librarian who wanted to confine me to the children's section. When I went to college in 1976, I had only seen women in limited roles: housewife, teacher, nurse, and librarian. I chose librarian. A friend in college convinced me to take a computer science course. He nagged and nagged until I finally took a programming course just to shut him up. This decision was life changing. I loved programming as much as I loved books! I ended up with computer science as my first major, and after graduation I worked several years as a programmer. Fast forward, I am married, and my husband encourages me to get an MS degree in Computer Science. This is when I discovered NLP the first time. It was the late 1980s, and NLP was interesting, but the techniques involved heuristic approaches that stubbornly refused to scale up to real-world utility. Fast forward again, my kids are grown, and I decide to go back to school to get my PhD. I became reacquainted with NLP, which by this time had transformed into something radically different with the introduction of statistical machine learning approaches.

I completed my Computer Science PhD in 2016 with an emphasis on Natural Language Processing, and have been teaching ever since at the University of Texas at Dallas. Our amazing department head, Gopal Gupta, gave me the green light to teach an undergraduate NLP course which eventually led to me putting together this book. I want to thank Dr. Gupta, as well as my wonderful students who continue to impress me as they fly off to internships and jobs both locally and in Silicon Valley.

I also want to thank four amazing professors I had at the University of North Texas where I completed my PhD. Rada Mihalcea and Paul Tarau taught me NLP, and Tarau was the first one to tell me that my course project could be developed into something publishable. That work became the basis of my dissertation. Cornelia Caragia taught me machine learning, and gave me confidence that I could really do some interesting projects. I also want to acknowledge the help of Kathy Swigger, who was my AI professor for my MS degree in the 80s and thankfully didn't retire until I finished my PhD. These professors have given me examples of how good professors treat their students, and I will try my best to carry on their legacy. Any good things in this work can be traced back to these four. Any shortcomings lie with me.

- This book has a companion git hub for code samples: `https://github.com/kjmazidi/NLP`
- You can keep up with future enhancements to the book on my blog: `https://karenmazidi.blogspot.com/`
- Video overviews of content are available here: `https://youtube.com/c/JaniceMazidi`

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Part One: Foundations

Part One

This book will explore traditional and modern techniques for Natural Language Processing. Before diving into the techniques, Part One sets the foundation for later learning. First, Chapter 1 gives an overview of what natural language processing is all about. Chapter 2 is a brief review of Python, since Python is the most commonly used language in NLP. Chapter 3 explores NLTK, the Natural Learning Tool Kit, the first of many NLP tools we will learn to use. NLTK is often used in text preprocessing, a vital first step in NLP projects. Chapter 4 rounds out the foundations part of the book with a glimpse into the field of linguistics and some important terminology that every NLP practitioner should know.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

1. Natural Language Processing

The field of *Natural Language Processing*, NLP, involves creating algorithms that allow computers to process human language. The term *processing* language can mean a wide range of things such as simply listening for and recognizing the phrase *OK Google*, to identifying spam words in emails, to creating a word cloud based on word counts, to classifying product reviews as positive or negative, to any of the multitude of tasks that would fall under the umbrella of NLP. Natural Language Processing itself is a branch of AI, as is Machine Learning. In a complex NLP project, some components may be purely NLP, others purely machine learning, and others in the general category of AI. All three of these fields are developing rapidly, and utilizing techniques from related disciples in novel ways.

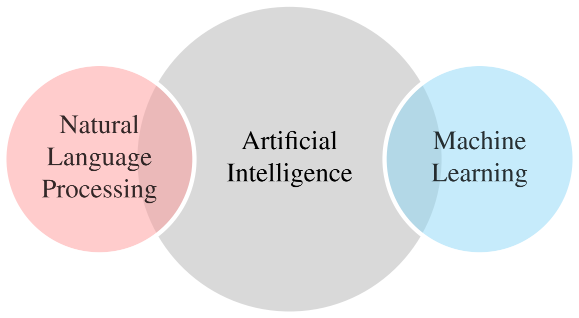


Figure 1.1: Machine Learning and Natural Language Processing

In a human-to-human dialog, two things are going on: *natural language understanding*, meaning that each party understood what the other person said, and *natural language generation*, the formation of spoken responses. This is illustrated in Figure 1.2.

Actually, there is a lot more going on than just verbal processing when two people speak, namely the social rules that govern our turn taking, tone, eye contact, gestures, and so forth. But the focus of

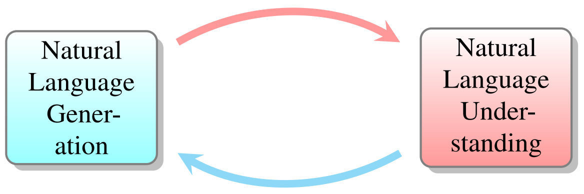


Figure 1.2: Natural Language Processing

this book will be primarily on the words. As you go through the materials, you will naturally become more attuned to human language: what we say, what we mean; and you will begin to think about how a machine could imitate this. You will also become more attentive to the ways that NLP is improving as you interact with it in your daily life through automated assistants, news feeds, web searches, and more. The advances in NLP in the past few years are remarkable and even more impressive accomplishments lie ahead. However, as with any branch of AI there is a lot of hype. Almost every month you can read yet another article written for the general public that claims that NLP has been "solved". Real improvements have been incremental at best, and will continue to be in the foreseeable future.

Before exploring NLP, you may not have thought about language much because it came to you naturally. The more you look at language, the more complex you realize it is. The meaning of our words is not so apparent, due to idiomatic language, sarcasm, hidden motives, and more. How is a machine supposed to learn all of that? We don't yet have answers, but little insights are learned every day.

1.1 A brief and biased overview of NLP

Like any big and rapidly evolving topic, capturing NLP in a nutshell is challenging. The approach taken in this book is to look at natural language, human language, in text form, rather than acoustical form. Automatic Speech Recognition, ASR, is not a solved problem but the progress is remarkable. Automated agents are able to understand human speech with a wide variety of accents and regional distinctions. Speech generation has also progressed to the point that automated agents sound more human and less robotic every year.

The focus of this book is on text. Text is examined in widening categories from words, to sentences, to documents in a corpus. Different things can be learned from text at each level. There are three main approaches to learning from words, sentences, and documents.

1. Rules-based approaches
2. Statistical and probabilistic approaches
3. Deep learning

1.1.1 Rules-based approaches

Rules-based approaches are the oldest techniques in NLP. For example, converting plural forms of words to singular ones can involve a few regular expressions and a list of exceptions. Another rules-based approach involves context-free grammar, which lists production rules for sentences. These production rules could be used either to generate syntactically correct sentences, or to check whether sentences are grammatically correct. A famous rules-based approach from the 1960s was Eliza, which used regular expressions to echo talking points back to the user, mimicking a talk therapist. When Eliza couldn't form an answer, a few canned responses were output. Here's an example:

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

```
User: What do you think of natural language processing?
Eliza: We were talking about you, not me.
```

Rules-based approaches were difficult to scale up because human language is complex, constantly evolving, and simply can't be encapsulated fully in rules. Nevertheless, many text processing problems can be solved with rules-based approaches. When a fast, simple rules-based approach to a problem exists, there is no need to train a huge neural network.

1.1.2 Statistical and probabilistic approaches

Rules-based approaches dominated until the 1980s. Starting in the late 1980s, mathematical approaches to text were developed. Simply counting words and finding the probabilities of words and sequences of words led to useful language models. These models can be part of machine translation systems. When translating 'big sister' from English to another language, a language model can determine that 'big sister' is better translated in the destination language as something that means 'older sister' rather than 'larger sister'. These language models can also be used for predictive text, as when you type a query into a search bar and receive suggestions for the most likely phrase you are typing.

Classic machine learning algorithms fall into this category as well, since they learn by statistical and probabilistic methods. Machine learning approaches became more popular as the data they need to learn from became more widely available. Classic machine learning algorithms such as Naive Bayes, Logistic Regression, SVMs, Decision Trees, and small Neural Networks are used today to solve many NLP problems. These approaches work well when only a moderate to large amount of data is available for training, and may even outperform deep learning algorithms on smaller data sets.

A statistical approach to a more sophisticated Eliza or other chatbot could involve learning prompt-response pairs from a large corpus. This could be done with classic machine learning algorithms or specialized deep learning algorithms.

1.1.3 Deep learning

Deep learning evolved from neural networks when huge amounts of data became available, and processing power increased through GPUs and cloud computing. The algorithms, including recurrent neural networks, convolutional neural networks, LSTMs, and more, are riffs off the basic neural network. New techniques are coming out every day with exciting results. However, not everyone has access to petabytes of data and the hardware to process it, so smaller-scale deep learning is still used in many NLP applications.

In fact many end-to-end NLP projects will involve techniques from rules-based approaches, statistical and probabilistic approaches, and deep learning, so all three approaches need to be understood.

The dream of deep learning is to make more and more human-sounding interactions possible. Achieving this goes beyond just retrieving likely responses to a user's statement, to considering the context of the conversation, to remembering a user's previously stated preferences, and much more.

Like the cutting edge of any AI, deep learning is high on the hype cycle right now. Here's my favorite quote about that (modified to not offend):

> Oh for f****s sake DL people, leave language alone and stop saying you solved it.
>
> - Yoav Goldberg

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

1.1.4 Fake it till you make it

Actually, a chatbot agent doesn't need to understand every word that the user said (or typed). Rather, it can be alert to key words and phrases, and base a response on what is most likely. So you can ask Google or Siri "pizza near me" or "I was wondering if there is any good pizza near me", and options will be presented either way.

1.1.5 Current trends and applications

In any given week you are likely to encounter several real-world NLP applications, such as:

- Sentiment analysis of product/service reviews
- Automated assistants to direct your call or solve your problem
- Machine translation such as Google Translate
- Recommender systems based on key words of products/services you have purchased
- Automated email reply suggestions

Companies have spent millions or hundreds of millions of dollars and untold man-hours building, improving, and maintaining these systems. However, when you get down to the lines of code, much of it involves techniques covered in this book.

1.2 Purpose and organization of the book

This book is written for the average college student or professional who wants to learn NLP. No prior experience is assumed, the book starts from the ground floor. A common complaint heard now among NLP practitioners is that people want to jump to machine learning NLP without learning NLP basics. You will not be an effective NLP practitioner unless you learn some foundational concepts and techniques, so these are covered prior to machine learning topics.

Part One of the book builds the foundational skills and knowledge you will need for NLP. Python is the language to know for NLP. Chapter 2 provides a sufficient background in Python programming for the rest of the book. Chapter 3 introduces NLTK, the Natural Language Tool Kit, that provides many helpful functions for NLP. Chapter 4 covers some linguistics basics that will help in reading and understanding NLP technical literature.

Parts Two through Four explore human language in widening scopes, from words, to sentences, to documents. There are interesting things to be learned from each level of text, and the tools and techniques vary by the scope.

Part Five provides an introduction to machine learning in Python, starting with the extra libraries for machine learning in Python: pandas, NumPy, sklearn, and more. Many machine learning algorithms are explored, from Naive Bayes and Logistic Regression to Neural Networks. Deep Learning and its variations are covered in Part Six.

The book explores NLP through hands-on coding practice to consolidate what you are learning. As you go through the book, you will build your skill set so that you can take on your own projects in the future.

1.3 Beyond the book

As you learn more about NLP, you should start keeping up with research from the top NLP conferences. Most cutting-edge research results are shared through these conferences. A nice NLP calendar is maintained here: `http://cs.rochester.edu/~omidb/nlpcalendar/`

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

2. Python Basics

This chapter covers enough Python to allow you to start coding projects. The first thing to do is install Python by going to: `https://www.python.org/downloads/`. The code in this book is compatible with Python 3.7 or later. If you have Python 2 on your machine, you will want to also install Python 3.[1]

Once Python is installed, you should be able to open a Terminal command line on your system and type `python`. If your system has both Python 2 and Python 3 installed, type `python3` at the console. This opens the Python interactive shell, outputs information about the version of Python installed, and provides the interactive prompt, >>>. Here is a simple interactive demo:

```
$ python3
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 16:52:21)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> name = "Karen"
>>> print("Hello", name)
Hello Karen
>>>
```

Python is an interpreted language and playing around at the console like this is a great way to learn. What did the code above do? The first line saved a string into variable `name`; the second line printed to the console. Python is designed to be clean, readable code, almost like reading pseudocode, so it is easy to figure out what the code does just by reading it.

Python is dynamically typed. Variable `name` was not declared, Python just figured out what type of variable it should be. Later in the code, variable `name` could change to some other type if a different

[1]Mac users should not uninstall Python 2, it may be used by your operating system

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

type of data is assigned to it. Python will not complain. It is the programmer's responsibility to keep up with what types of data variables refer to.

There are several ways to write Python code:

- Type code into the console; this is useful for testing out ideas and learning how Python works
- Type code into a simple text file, ending in .py; such Python scripts can then be run from the console
- Type code into an IDE; this approach is best for larger projects
- Type code into a Jupyter notebook; this approach is nice for sharing code with others

The short demo above showed how to run Python code at the terminal. To save code in a Python script, just use any simple text editor that saves plain text (ASCII) files, and make sure the file ends in `.py`. Then you can run the code from the console:

```
python myscript.py
```

or, if your system has both Python 2 and Python 3 installed:

```
python3 myscript.py
```

For larger projects, most people prefer to code in an IDE. There are many great (and free!) IDEs for Python, such as PyCharm.

2.1 Jupyter Lab

Somewhere between the interactive shell and the stand-alone program run in an IDE is the Jupyter notebook. Project Jupyter is an open-source project that supports interactive programming. This is a common platform for sharing code, commentary, and results for developers, data scientists, and more. Installing and using Jupyter notebooks is not required to go through the material in this book. Coding in an IDE is fine. Jupyter notebooks are available on the github site for the book and you can read those online without having to install Jupyter. However, being able to share code in a notebook is a good skill to have. JupyterLab is easy to install. Go to: `https://jupyterlab.readthedocs.io/en/stable/` for installation instructions. Installation of Jupyter can be completed with the usual pip install. Alternatively, you could install Anaconda which installs Python, Juypter and other packages. Be aware that Anaconda will take up a lot of space on your computer, so consider that before installing it. Throughout the book, please keep in mind that the `pip` command should be `pip3` if you have both Python 2 and 3 on your system.

Jupyter notebooks intersperse text cells and code cells. Any output from code cells displays immediately below the cell. Text cells are for commentary; they can be plain text or can use Markdown for formatting. There is built-in help for Markdown in the Jupyter notebook.

For the remainder of the chapter we will use code that can be found in my `Python for AI` GitHub repo.[2] You can follow along in your own notebook, or open up a Python interactive terminal at your computer and type along. If you just need a quick refresher on Python, see the GitHub repo for this book.[3]

[2] `https://github.com/kjmazidi/Python_for_AI/tree/master/1-Python_Basics`
[3] `https://github.com/kjmazidi/NLP/tree/master/Part_1-Foundations/Chapter_02_intro_python`

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

2.2 Getting Started with Python 3

The goal of this chapter is to learn enough Python to complete projects. There are links at the end of the chapter for more in-depth Python resources. Here are a few things to keep in mind about Python:

- Python is an interpreted language
- Source code is compiled into bytecode to be executed by the operating system
- There are no type declarations as in other languages, like *int i*;
- Types are checked dynamically as the program runs
- Python uses indents to signify code blocks, not { }
- The end of a line is the end of a statement, no need for ;
- Python is case sensitive
- Comments start with #

This book focuses on Python 3. There were some significant changes between Python 2 and Python 3 that break backward compatability. The main thing you will notice when looking at code is that with Python 3, parenthesis are required around the print statement:

```
print "hello"   # Python 2
print ("hello") # Python 3
```

Another thing to keep in mind when looking around at Python code in the wild are two flame wars: spaces v. tabs, and camelCase v. underscores. Underscores are generally preferred for variable and function names, but the main thing is to be consistent within your own code and follow conventions of your peers when writing code with others. Regarding tabs versus spaces, PyCharm converts tabs to spaces automatically, but when writing code at the console, spaces need to be used. The authority on Python style is the PEP8 style guide: `https://www.python.org/dev/peps/pep-0008/`.

The next few sections teach the basics of Python programming:

- Python native data types and how to use them
- Python control structures to create loops, functions, if statements
- Python built-in data structures: lists, dictionaries, and more

2.3 Variables and Data Types

The native data types in Python include:

- int - non-limited length
- float - same as C double
- complex
- boolean
- string - 'single' or "double" quotes

Python also has some useful built-in constants: `True, False`, and `None`.

Python variables are essentially pointers to memory locations. The type of a variable is determined by its contents. Type along at the console:

```
>>> v = 5
>>> type(v)
<class 'int'>
>>> v = 'a'
>>> type(v)
<class 'str'>
```

In the code above, when 5 is assigned to v, the type of v is int. Later when 'a' is assigned to v, the type of the variable changes to string. What happened under the hood is that after the first assignment, v pointed to the integer 5, and after the second assignment, v was modified to point to the string 'a'.

Operators can have different meanings in different contexts, as shown below. The shortcut operator += means addition with integers but concatenation with strings. By the way, Python does not have the ++ or −− shortcut operators as many other languages do. Don't be afraid of experimenting with code and causing errors. Getting to know Python's error messages is an important part of learning.

```
v = 5
v += 1
print(v)   # v is now 6
v = 'a'
v += 'b'
print(v)   # v is now 'ab'
v += 1    #  will cause an error, should be v += str(1)
```

The print() function is used to output to the console, and the input() function is used to get input from the user. Here is an example of console input and output:

```
name = input("What's your name? ")
print('Hello ', name, '!')  # notice that ',' adds a space
print('Hello '+ name + '!') # '+' (concatenate) does not add a space
```

The input() function reads whatever the user types as a string. To get numeric data from a user, the string input can be converted to numbers with the built-in int() and float() functions:

```
radius = input("Enter radius: ")
radius = float(radius)
area = radius * radius * 3.14
print("area = ", area)
```

2.4 Python Scripts

A Python script is just a plain text file of Python code. The file name should end in .py. All statements should start at the leftmost column. Each line of code will be executed sequentially. The following code example shows a commonly used outline for Python scripts with a main function, and shows how to process system arguments.

Code 2.4.1 — A Python Script. Example with Arguments

```
import sys

def main():
    print("Hello " + sys.argv[1] + "!")

if __name__ == '__main__':  # uses double underscores (dunders)
    main()
```

In the code above, any system libraries needed are imported first. The `sys` library is used to get command-line arguments. After the import, a `main` function is defined. The main function just prints Hello to the user. Python will import sys then read in the main function definition. If we want it to actually run main, it needs to be called. The last two lines call the main function. This is a typical way of starting a program. Later we will discuss what a single underscore or double underscore signify, for now just copy this code. You can run this script as follows, assuming it is saved it as hello.py:

```
$python hello.py Karen
$Hello Karen!
```

2.5 Control Structures

The following example shows the syntax for an `if` statement, an `if-else`, and an `if-elif-else`.

Code 2.5.1 — Python IF Statements. Note the importance of the indents.

```
grade = 66

# simple if
if grade >= 60:
    print("passed")

# if-else
if grade >= 60:
    print('passed')
else:
    print('failed')

# if-elif-else
if grade >= 90:
    print('excellent')
elif grade >= 60:
    print('passed')
else:
    print('failed')
```

Notice that code blocks are defined simply by indentation. This can be tricky at first. A common error is to place statements at the wrong indentation level. Notice also:

- No parenthesis are needed around conditions
- A colon is required after the condition
- A colon is required after the else

The if-elif-else shows how to have multiple conditions checked in one statement. There is no case or switch statement in Python, since the if-elif-else structure can be used to implement the logic.

Python has built in constants True and False. Logical False can also be 0, and empty object, or the built-in constant None. Everything else is True. Let's see how that works:

```
flag = True
if flag:
    print('flag is true')

string1 = 'abc'
if string1:
    print(string1)

string1 = ''
if not string1:
    print('Empty string)
```

The first print statement checks the value of the variable flag, which is True, and prints the statement. It is not considered Pythonic to do this: `if flag == True`, because it is unnecessarily complex. Notice the second if statement above also results in True because the variable string1 is not null. The third example above prints only if the string is empty.

Code 2.5.2 — Python while Statement. Condition must change within the body of the loop in order to avoid infinite loops.

```
i = 3
while i > 0:
    print(i)
    i -= 1


3
2
1
```

Again, the while condition does not need parentheses around it but does need the colon at the end. The body of the while loop is indented. Here is the same logic using a for loop:

```
for i in range(3, 0, -1):
  print(i)
```

The for loop above uses the built-in `range(start, stop, step)` function that creates a sequence of values to iterate over. This is one way to use a traditional for loop. Typically we use the for loop to iterate over a sequential object:

Code 2.5.3 — Python for loop. Iterating over a list.

```
for item in ['a', 2.3, 'hello']:
    print(item)


a
2.3
hello
```

The code `['a', 2.3, 'hello']` creates a sequential structure that the for loop iterates over. This structure is a list, which is discussed later. With each iteration, the next element of the list is copied to `item`, and then printed.

The code example above iterated over each element in the list, but what if you also want to know their position in the list? The built-in function enumerate does that.

Code 2.5.4 — For Loop. Using the enumerate() construct.

```
mylist = ['apple', 'banana', 'orange']
for i, item in enumerate(mylist):
    print(i, item)

0 apple
1 banana
2 orange
```

The enumerate() function returns two things, the index and the element. In the for loop above, the variable i will contain the index, starting at 0, and the variable item will contain the element. Python functions, either built-in or user-defined, can return more than one object.

The code example below shows how to define and call a function. A function is defined with reserved word `def`. A function definition needs a name, optional arguments in parenthesis, followed by the colon. Statements within the function are indented. All the statements in the function body, including the return, are at the same level of indentation, except when you need further indentations as in the for and if statements.

Code 2.5.5 — Functions. Function definition and sample call.

```
def find_first(names):
    first = names[0]
    for name in names[1:]:
        if name < first:
            first = name
    return first
names = ['Jane', 'Zelda', 'Bud']
print('The first name alphabetically is ', find_first(names))

The first name alphabetically is Bud
```

The function above iterates through a list of names to find the one that is first alphabetically. There are several things that should concern you about the code: (1) the function expects variable names to be a list but the type was not checked, (2) the code assumes that **names** has at least one element, and (3) the function was not documented. Below is an improved version, with triple-quoted docstrings.

The documentation in the function below with the triple quotes is called a *docstring*. A docstring can be used for a string literal that spans multiple lines, but it is primarily used for function documentation. The docstring below might be overkill for such a simple function, but it demonstrates good practices. The first line of the docstring should be a simple description of what the function does, starting with a verb. Function documentation should explain the input arguments as well as what the function returns, and finally give a sample function call.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

```
def find_first(names):
    """
    Finds the first item alphabetically in a list.
    Args:
        names:  a list of items to be compared
    Returns: string
        the first item, alphabetically
    Example:
        >>>find_first(['george','anne'])
        >>>'anne'
    """
    if not type(names) == list:
        return 'Error: "names" is not a list'
    if not names:
        return 'Error: "names" is an empty list'
    first = names[0]
    for name in names[1:]:
        if name < first:
            first = name
    return first
```

Most of the coding examples in the book and online in the GitHub will not demonstrate either the type checking that is essential for reliable code, or this full documentation style for functions. This is done intentionally for the reader, to condense the number of lines to read. However, in actual projects, this kind of documentation is standard.

2.6 Strings

Python has a built-in `str` class with many built-in functions for text processing. Strings can be enclosed in single or double quotes, as well as triple quotes for multiple lines. Python doesn't have a char type, a character is just a string of length 1. You can access the individual characters in a string with [n], where n ranges from 0 to the length of the string minus 1. The code below iterates over a string one character at a time, using i to index into the string. The output is shown below the code.

```
string1 = 'hello'
for i in range(len(string1)):
    print(i, string1[i])
0 h
1 e
2 l
3 l
4 o
```

Python also supports negative indices. That is, [-1] is the last element, [-2] is the next to last, etc.

```
print(string1[-1])
o
print(string1[-2])
l
```

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

2.6.1 Slicing and Operators

Substrings can be extracted from strings in various ways with slicing:

- s[i:j] extracts from s[i] to s[j-1]
- s[i:] extracts from s[i] to the end of the string
- s[:j] extracts from the beginning to s[j-1]
- s[:] makes a copy of the entire string

Try the following at the console:

```
string_utd = 'The University of Texas at Dallas'
string_utd[18:23]
string_utd[4:]
string_utd[9:] + ", " + string[18:23]
```

There are special operators for strings: + for concatenation, ∗ for repetition, *in* for iteration or conditional, % for formatting. Try the following at the console:

```
print('a' + 'b')
print('a' * 3)
print('a' in string_utd)
# older form of formatting
print("Format an int: %d, a float: %2.2f, a string: %s" % (5, 5.6, 'hi'))
# newer formatting style
print("Format an int: {}, a float: {:2.2f}, a string: {}".format(5, 5.6, 'hi'))
# f-string (newest) formatting style
print(f"Format an int: {5}, a float: {5.6:2.2f}, a string: {'hi'}")
```

2.6.2 String Methods

There are dozens of built-in string methods. You can find a comprehensive list in the Python documentation. Here are some that are commonly used:

- upper() and lower() to change case
- isalpha(), isdigit(), isspace() to check type
- startswith() and endswith() to match start/end substrings
- strip() to remove whitespace from start and end
- split() to split a string into a list of substrings
- join() to join a list of strings into one string
- find() - return index or -1
- count() - count unique occurrences
- len() - return length of string

Some of these methods return strings, some return boolean values and some return integer indices. The return type makes sense when you practice using the functions. Because they are methods, they can be applied to strings as in `mystring.method()`. The methods in the following code sample, lower(), upper(), and strip(), apply to a string, and return a modified string.

Code 2.6.1 — Selected String Methods. These methods return strings.

```
# upper() and lower()
string_pp = 'Pied Piper'
print(string_pp.lower(), string_pp.upper())
pied piper PIED PIPER

# strip whitespace from both ends
spacey = " hello "
not_spacey = spacey.strip()
len(not_spacey)
5
```

The methods in the following code sample apply to a string and return boolean values True or False.

Code 2.6.2 — Selected String Methods. These methods check string content and return Boolean values.

```
# check characters
string4 = 'number = 3'
print(string4[0].isalpha())  # True
print(string4[-1].isdigit())  # True
print(string4[6].isspace()) # True

# check start and end of string
string_hello = 'hello world'
print(string_hello.startswith('hello'))  # True
print(string_hello.endswith('world'))  # True
```

The following methods operate on a string and return integers. The find(text) function returns the index where "text" starts in the string. The count(text) function returns the number of times that "text" appears in the string. There is an index() function that works like find() but raises an exception if the text is not found in the string; the find() function will return -1.

Code 2.6.3 — Selected String Methods. These methods return integers.

```
string_utd = 'The University of Texas at Dallas'
i = string_utd.find('Dallas')
string_utd[i:]  # Dallas

count_a = string_utd.count('a')
print(count_a)  # 4
```

The next two methods convert from a string to a list or vice versa. The split() function will split a string into a list based on white space by default. Within the parenthesis you can also split on your chosen delimiters. The join() function takes a list and joins the elements into a string with the join character (or none) specified. Try the following.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Code 2.6.4 — Selected String Methods. Methods that work on lists and strings.

```
# split a string into words
long_string = 'this is a lot of text in a string'
tokens = long_string.split()
for token in tokens:  # prints each word on a line
    print(token)

# join a list into a string
print(''.join(tokens))  # smashes words together
print(' '.join(tokens)) # separates words with a space
print('*'.join(tokens)) # separates words with *
```

2.7 Lists, Tuples, Dicts, and Sets

Python has several built-in data structures that organize data. These structures are called **containers** and have a fairly uniform set of methods for operating on them. We explore the most important types of Python containers in this section: lists, tuples, dicts, and sets.

2.7.1 Lists

A Python list is an ordered sequence of objects. These objects can be of any type, and can be other lists. Lists are mutable, meaning that we can change them in place. This is in contrast to strings. When we change a string in place, Python actually creates a new string. Follow along at the console as we show several ways to create lists from data and/or other lists. Echo each list back to the console as you go so that you understand the syntax.

```
lista = [1, 2, 3] # create a list using [ ]
listb = list(range(4, 7)) # create a list using list()
listab = lista + listb # "+" means concatenate for lists
listab += [8, 9] # "+=" is the concat shortcut
listab += [7] # we need [ ] around the 7
print('length = ', len(listab))
```

By the way, Python will let you overwrite functions. Look at the following code sequence, but don't do this! Once the code overwrote 'list', the original function list() could not be called. If you accidentally do something like this at the console, just get out of the console and go back in to reset everything to the original state.

```
list = [1, 2, 3] # don't do this!
new_list = list(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' object is not callable
```

We can modify lists by adding and removing items as shown in the next code example. Again, follow along at the console and echo back `fruits` after each command.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Code 2.7.1 — Lists. Ways to add and remove list items.

```
fruits = ['apple', 'banana', 'orange']

fruits.insert(0, 'peach') # insert at the start of the list
fruits.append('pear') # insert at the end of the list

print(' '.join(fruits))

if 'apple' in fruits:
    fruits.remove('apple')

fr = fruits.pop()   # pops last item
fr = fruits.pop(0)  # pops first item

i = fruits.index('banana')  # throws Value Error if not in list

del fruits[i] # throws Index Error if out of range
del fruits[:] # clears out a list
```

Copying a list is straightforward: `list2 = list1`. However, you may observe some strange behavior when you copy a list if that list contains lists. Try the following at the console:

```
# copy example 1
list1 = [1, 2, [3, 4], 5]
list2 = list1
list2[2][0] = 9
list1  # the sublist in list 1 was changed when list 2 was changed
```

In the example above, we performed a shallow copy. Python copied each element and a pointer to the list within the list. The way around this is to do a deep copy:

```
# copy example 2
from copy import deepcopy
list1 = [1, 2, [3, 4], 5]
list2 = deepcopy(list1)
list2[2][0] = 9
list1 # list1 remains unchanged
```

In Python, try to avoid for loops because they are slow. Instead of looping over elements in a for loop, try to use a *list comprehension*. A list comprehension iterates over elements in an object, and creates a list of these objects. The basic syntax below just iterates over each element in the old list and makes a new list:

```
list_new = [x for x in list_old]
```

Of course it is inefficient to copy a list like this. The real power of list comprehensions comes in adding conditions and modifications, as shown in the example below:

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Code 2.7.2 — List Comprehensions.. Use list comprehensions instead of for loops.

```
friends = ['Jim', 'Hamed', 'Charlotte']

friends2 = [f.lower() for f in friends]
print(friends2)
['jim', 'hamed', 'charlotte']

friends3 = [f for f in friends if not f.startswith('C')]
print(friends3)
['Jim', 'Hamed']
```

2.7.2 Tuples

A tuple is a sequence of objects, just like a list. The difference is that tuples are immutable and use parenthesis instead of square brackets to easily distinguish them. Why do we need tuples? There are situations that arise where you want something like a list but immutable. For example, a key for a dictionary can be a tuple, but not a list, because the tuple is immutable. You can convert between tuples and lists with the list() and tuple() functions. The first line of code below created tuple t1 by specifying a comma separated list on the right side of the assignment operator. The second line of code created a tuple with the tuple() function. Unpacking a tuple involves copying each element to a variable. This can be a useful way to return multiple values from a function.

Code 2.7.3 — Tuples. Create and Unpack Tuples.

```
# create a tuple in 2 different ways
t1 = 'a', 3, 5.6 # create a tuple
t2 = tuple(range(1:4))

# tuple unpacking
astring, adigit, afloat = t
print("%d %s %f" % (adigit, astring, afloat))
```

2.7.3 Sets

A set is an unordered container object that does not allow duplicates. Sets are immutable in that you cannot change a set element in place but sets have other methods such as union(), intersection(), and difference().

Code 2.7.4 — Sets. Set Creation.

```
# create a set
set1 = {'apple', 'banana', 'orange'}
# create a set from a list
list1 = [1, 2, 3, 4, 3, 2, 1]
set1 = set(list1)
print(set1)
{1, 2, 3, 4}
```

2.7.4 Dicts

Python dictionaries contain unordered key:value pairs. Order is not maintained in dicts so if you want them in a particular order for processing, you must sort them as shown in the following code example. The sorted() function will return a sorted `list` of key:value pairs. Key types can be anything that is immutable, like strings or tuples. Dictionary values can be any Python type.

Code 2.7.5 — Dictionaries. Creating and Using Dicts

```
# create dict in two ways
fruits = {'a':'apple', 'b':'banana', 'p':'pear'}  # one way

# another way:
fruits = {}   # create an empty dict
fruits['a'] = 'apple'
fruits['b'] = 'banana'
fruits['p'] = 'pear'

print(fruits['p'])
fruits['p'] = 'peach'  # change 'p' value
print(fruits['p'])

if 'c' in fruits:      # check for 'c' to avoid key error
    print(fruits['c'])
else:
    print("There is no fruit c")
```

In the code above, we used `in` to check if an item was in the dictionary in order to avoid key errors. Another way is to use get() as shown below:

```
# use .get() to avoid key error
print("fruit c = ", fruits.get('c'))
```

We can delete dict items using their key values as in:

```
del fruits['p']
```

Code 2.7.6 — Dictionaries. Iterating over Dicts

```
# iterate over dict
for k, v in sorted(fruit.items()):
    print(k, v)
```

Dictionaries aren't sorted, by definition. The sorted() function above actually returns a list of tuples, where the first item is the key and the second is the value. Each tuple is unpacked into k for key and v for value. The tuples in the list are iterated over, not the dictionary.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Often in text processing we will create a vocabulary dictionary with counts from text. Here is a simple example:

Code 2.7.7 — Dict. Create vocabulary counts.

```
text = 'jack be nimble jack be quick'
tokens = text.split()
vocab = {}
for token in tokens:
    if token in vocab:
        vocab[token] += 1
    else:
        vocab[token] = 1
print(vocab)
```

2.7.5 Operations on Container Objects

We can iterate over container objects (and strings) with the `for-in` loop, and test for membership with `in` as well. We demonstrate these methods with a tuple below, but they work on many containers.

Code 2.7.8 — Container Methods. Loops and Membership Check

```
friends = ('Jim', 'Hamed', 'Charlotte')

for friend in friends:
    print('Hello', friend)

if 'Charlotte' in friends:
    print('Hello Charlotte')
```

There are functions for sorting containers as well. Use sorted() when you want a new list returned and sort() or reverse() to modify the list in place. The slicing techniques we discussed in strings above work the same on lists. Keep in mind that if a container is unordered, like sets and dicts, what sorted() returns is a sorted list.

Code 2.7.9 — Container Methods. Sorting.

```
friends = ['Jim', 'Hamed', 'Charlotte']
for friend in sorted(friends):
    print(friend)

print('after sorted():',' '.join(friends))
friends.sort()
print('after sort():', ' '.join(friends))
friends.reverse()
print('after reverse():', ' '.join(friends))
# slicing
print('after reverse():', ' '.join(friends[1:]))
```

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

2.8 Type Checking

A notebook in the GitHub folder `Xtra_Python_Material` named `Type Checking` shows how to check types with an external Python library named mypy. The mypy library can be installed with pip/pip3. First, we annotate types. This annotation doesn't do anything within Python, but when combined with mypy, type checking is possible.

The type of function parameters can be specified with this syntax: `identifier [: expression]` In the code below, the function parameter is annotated with ':list' to let the reader of this code know that variable names needs to be a list, and will also specify the type for the mypy checker.

Another thing we can do is to annotate what the function should return with the arrow -> syntax: `-> str`. In the code below, the function returns a string, which is made clear by the arrow annotation.

Code 2.8.1 — Type Checking. with mypy

```
In [6]:
# let the syntax help us with types
def find_first(names: list) -> str:
    #if not type(names) == list:
    #    return 'Error: "names" is not a list'
    if not names:
        return 'Error: "names" is an empty list'
    first = names[0]
    for name in names[1:]:
        if name < first:
            first = name
    return first

names = ['Jane', 'Zelda', 'Bud']
print('first name is ', find_first(names))
```

With annotations in code, mypy can be run at the console as follows:

```
$mypy myprog.py
```

If you get no output from mypy, that's great. If the above code is edited to include an error, such as: `find_first(1)`, then mypy will output the following error message.

```
$ mypy names.py
names.py:15: error: Argument 1 to "find_first" has incompatible type "int";
     expected "List[Any]"
Found 1 error in 1 file (checked 1 source file)
```

Notice the error above said that the input type should be List[Any]. If we want to be more specific, we can modify the annotation to be `List[str]`, if we also import List:

```
from typing import List
def find_first(names: List[str]) -> str:
    . . .
```

2.9 A word about underscores

As mentioned above, PEP8 suggests naming variables and function with words separated by underscores. In Python code, you will see underscores used in other ways as well.

A trailing underscore in a variable name is often used to prevent a clash with Python reserved words, as shown in the example below, although it is probably better to think up a more meaningful name.

```
list_ = [1, 2, 3]
```

A leading underscore, like `_phone` is an indication to readers of your code that the variable is intended for internal use. This does not strictly enforce privacy, but does prevent objects whose names start with an underscore from being imported with the import statement.

Double leading underscores (called a *dunder*) for class attributes invokes name mangling. This is not encapsulation, but yet another weak privacy enforcement. Dunders on leading and trailing ends of a name are so-called *magic* objects or attributes that exist in namespaces. Examples are the `__init__` method. The PEP guidelines recommend that you do not make up your own magic names but follow the conventions in Python.

One other thing to note about underscores and naming is that if a C/C++ module has a Python module that provides a higher-level interface, the C/C++ module will have a leading underscore, example: `_socket`.

An underscore by itself can be a variable name, and is often used for throw-away variables we don't care about:

```
some_tuple = ('Karen', 'Mazidi')
_, last = some_tuple   # don't care about first item in tuple

print(last)
```

The notebook `Underscore Syntax` in Folder `Xtra Python Material` in the GitHub has sample code to accompany this section.

2.10 More Python

There is much more to Python than has been covered so far. The best way to learn more is to start coding. Python is a versatile language that you can use for either traditional, object-oriented, or functional programming. You can refer to the Python notebooks on the Python for AI GitHub repo for how to create and use classes, read from and write to files, handle exceptions, and use regular expressions. For now, we have covered enough to get started with the rest of the material.

Most of the code that you find for demos in NLP and machine learning is procedural, in Jupyter notebooks, and not object-oriented. This is because no one wants the overhead. As Joe Armstrong, creator of the Erlang programming language observed:

> The problem with object-oriented languages is that they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

On the other hand, most of the Python NLP and ML libraries you encounter are written in an object-oriented paradigm.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

2.11 Summary

Python is the most important language in NLP. Python is an interpreted language, which makes it easy to play with at the console to explore how things work. Python is dynamically typed, placing more burden on the programmer to remember the types of variables.

As we go through the book we will learn how to use many Python libraries that extend Python's functionality, and enable more memory efficient and faster data structures.

2.11.1 Quick Reference

Reference 2.11.1 Slicing Strings

```
>>> s1 = 'abcdef'
>>> s1[:5]  # up to but not including index 5
'abcde'
>>> s1[5:] # from index 5 to the end
'f'
>>> s1[-3:] # from the -3 index to the end
'def'
```

Reference 2.11.2 List Comprehension

```
>>> numbers = [1, 2, 3]
>>> numbers_squared = [n**2 for n in numbers if n > 1]
>>> [4, 9]
```

Reference 2.11.3 Tuple Unpacking

```
>>> t = (1, 2, 3)
>>> n1, n2, n3 = t
```

Reference 2.11.4 Create a Set

```
>>> s = set('elephant'}
>>> s
{'l', 'p', 'h', 'a', 'n', 'e', 't'}
```

Reference 2.11.5 Dictionary Comprehension

```
>>> d = {i:i**2 for i in range(1,4)}
>>> d
{1: 1, 2: 4, 3: 9}
>>>
```

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Reference 2.11.6 Sort Dictionary by Keys

```
>>> for k, v in sorted(d.items()):
...     print(k, v)
...
1 1
2 4
3 9
```

Reference 2.11.7 Sort Dictionary by Values

```
>>> for k, v in sorted(d.items(), key=lambda item: item[1]):
...     print(k, v)
...
1 1
2 4
3 9
```

2.11.2 Practice

The following exercises will reinforce your understanding of Python.

Exercise 2.1 Practice writing functions and calling them:

1. Write a function to return the average of a list of numbers. Test your function.
2. Write a function to check if number k evenly divides n. Return a boolean value. Print a message in the calling code. By the way, the operator % is modulus and works as in other programming languages.

■

Exercise 2.2 Write the following string manipulation functions and test them on various input strings:

1. Given a string, return the first and last characters joined into a new string; if the string is less than 2 characters, return the string
2. Given a string, return the number of vowels
3. Given a string, return a string containing all found vowels, ex: 'ieaieuai....' for this sentence.
4. Given a string, return a string containing 'aeiou' if all vowels found, 'ai' if only vowels a and i were found, etc. The returned string of vowels should be in alphabetical order.

■

Exercise 2.3 Practice with lists:

1. Make a list from the individual words in a sentence.
2. Print only tokens that start with a given letter or group of letters.
3. Print only tokens that end with a given letter or group of letters.
4. Use a list comprehension to create a list of only words greater than 2 characters long.

■

Exercise 2.4 Practice with Tuples and Sets:

1. Create a tuple from the letters of a word using the split() function. Print the tuple in a loop.
2. Create a set from the tuple. Print the set.

■

Exercise 2.5 Try the following:

1. Create a long text string.
2. Create a set of unique tokens in the string.
3. Create a dictionary of vocabulary counts by iterating over the set of words.
4. Print the dictionary counts in alphabetical order.

■

2.11.3 Going Further

This has been a whirlwind tour of Python. As you develop your Python skills you will need more detailed resources. Here are a few recommended resources:

- Dive into Python – `http://www.diveintopython.net/index.html`
- The Hitchhiker's Guide to Python – `https://docs.python-guide.org/`
- The Python Documentation – `https://docs.python.org/3/index.html`
- PEP 8 Style Guide – `https://www.python.org/dev/peps/pep-0008/`

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

3. Intro to NLTK

Throughout this book you will learn how to use several NLP tools and libraries. The first one discussed is NLTK – the Natural Language Toolkit. The website for NLTK (`https://www.nltk.org/`) provides instructions for installing NLTK, as well as documentation. You should follow the links to install NLTK. There is also an associated book: `http://www.nltk.org/book/` by Bird, Klein, and Loper. The authors of the book are also the main developers of NLTK, as well as NLP researchers and educators. The book is a great reference to add to your library.

After you have installed NLTK, you can import it as shown in the following console interaction, and then download some of the data:

```
$python
>>> import nltk
>>> nltk.download()
```

The last line above will pop up a box that lets you select what to download. If you have a lot of room on your computer, you could download all of the data. However, you can just select the "book" data for now if you are concerned about space. You can always rerun `nltk.download()` at any time if you need more data. NLTK is open source, so you can dig through the code at the API link (`https://www.nltk.org/api/nltk.html`) and click on any of the source links to see the code. Additionally, the code is an excellent example of Python in an object-oriented paradigm.

NLTK has a lot of useful functionality, but to get started, the next sections look at two features: tokenization, and obtaining frequencies.

3.1 Tokenization

The process of *tokenization* involves dividing text into smaller units. Usually the word *tokenize* refers to dividing text into individual *tokens*, which include words, numbers, and punctuation, basically

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

everything but whitespace. The term *sentence tokenization* or, more precisely *sentence segmentation*, means to divide text into individual sentences. You can do both with NLTK.

Recall that text can be split with the Python split() function:

```
text = "I am Sam. Sam I am. I do not like green eggs and ham."
tokens = text.split()
print(tokens)
['I', 'am', 'Sam.', 'Sam', 'I', 'am.', 'I', 'do', 'not', 'like',
    'green', 'eggs', 'and', 'ham.']
```

Looking at the output above, you can see that punctuation attached to text. That's generally not desired. Let's see how NLTK tokenizes it in a better way.

Code 3.1.1 — NLTK. Tokenizing Words.

```
from nltk import word_tokenize
tokens = word_tokenize(text)
print(tokens)
['I', 'am', 'Sam', '.', 'Sam', 'I', 'am', '.', 'I', 'do', 'not', 'like',
    'green', 'eggs', 'and', 'ham', '.']
```

Now, the punctuation marks are separate tokens. The tokens are returned as strings within a list. Notice that you need to import the word tokenize module first.

The sent_tokenize module divides text into sentences:

Code 3.1.2 — NLTK. Tokenizing Sentences.

```
from nltk import sent_tokenize
sentences = sent_tokenize(text)
print(sentences)
['I am Sam.', 'Sam I am.', 'I do not like green eggs and ham.']
```

NLTK separated the text into individual sentences, as strings within a list.

3.2 Preprocessing Text

The first step in most NLP projects is text preprocessing, sometimes called *text normalization*. This could involve any or all of the following:

- Convert all the text to lower case
- Removing punctuation
- Removing numbers or replacing them with a NUM token
- Stemming words - removing affixes
- Lemmatizing words - converting to the root form
- Removing stop words - reducing text to important words only

Why is normalization important? Normalization changes related words into some standard form for the purpose of counting the occurrences. For example, you may want *Education, education, educational, educationally* to all be counted as *education*.

Converting all text to lower case is often a first step, and this can be done with Python's lower() function. Regular expressions can be used to get rid of punctuation and digits. See the GitHub online

(Fundamentals Folder of Chapter 2) for regular expression examples. For the remaining preprocessing steps, NLTK is often used, as shown in the following code example. For most projects, you will choose either stemming or lemmatization. The output below shows the difference.

Code 3.2.1 — NLTK. Text Preprocessing

```
import nltk
from nltk import word_tokenize
from nltk.stem.porter import *
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords
import string
import re

raw_text = """ I teach at the University of Texas at Dallas. I
     started teaching there in 2016. As of 2018, UTD has been designated
     as a national research university!"""

# remove punctuation and numbers with a regular expression
text = re.sub(r'[.?!,:;()\-\n\d]',' ', raw_text.lower())
print(text)
' i teach at the university of texas at dallas  i
   started teaching there in       as of       utd has been designated
      as a national research university '

# tokenizing extracts words, not white space
tokens = nltk.word_tokenize(text)

# stemming removes affixes from words.
stemmer = PorterStemmer()
stemmed = [stemmer.stem(t) for t in tokens]
print(stemmed)
['i', 'teach', 'at', 'the', 'univers', 'of', 'texa', 'at', 'dalla', 'i',
'start', 'teach', 'there', 'in', 'as', 'of', 'utd', 'ha', 'been',
'design', 'as', 'a', 'nation', 'research', 'univers']

# lemmatization finds the root words
wnl = WordNetLemmatizer()
lemmatized = [wnl.lemmatize(t) for t in tokens]
print(lemmatized)
['i', 'teach', 'at', 'the', 'university', 'of', 'texas', 'at', 'dallas',
   'i',  'started',  'teaching', 'there', 'in', 'a', 'of', 'utd', 'ha',
    'been', 'designated', 'a', 'a', 'national', 'research', 'university']
```

Code 3.2.2 — NLTK. removing stopwords

```
from nltk.corpus import stopwords

# removing stopwords
stop_words = set(stopwords.words('english'))
tokens = [t for t in tokens if not t in stop_words]
print(tokens)
['teach', 'university', 'texas', 'dallas', 'started', 'teaching', 'utd',
'designated', 'national', 'research', 'university']
```

You should type the code in the above Code block into the console to get comfortable with the syntax. A Python script `preprocessing.py` is available in the GitHub that contains the preprocessing code in a function.

There are several stemmers available in NLTK. The Porter stemmer is a well-known stemmer that has been around for decades. In looking at the results of the NLTK implementation, you can see some problems with stemmers. Namely, they are rather blunt instruments. Here are the major problems:

- 'university' was stemmed to 'univers' which does not have the same meaning
- 'Texas', 'Dallas' and 'as' had the s removed as if these were plurals, which they are not

Now look at the output of the lemmatizer. Not much has changed there. The NLTK lemmatization code seems to be much less aggressive than the stemming function.

Finally, we see the results of stop words being removed. In this case, what remained are the nouns, verbs other than 'be', and adjectives. Prepositions, determiners, and other words with little content were removed.

3.3 NLTK in Other Languages

We will be dealing with English text but many NLP tools support other languages. NLTK stemmers support Danish, Dutch, English, Finnish, French, German, Hungarian, Italian, Norwegian, Portugese, Romanian, Russian, Spanish, and Swedish. Other NLP tools discussed later can handle more international languages such as Arabic and Chinese.

Code 3.3.1 — NLTK. Other Languages.

```
import nltk.data
tokenizer = nltk.data.load('tokenizers/punkt/spanish.pickle')
>>> tokenizer.tokenize('Hola mi amor. Como estas?')
['Hola mi amor.', 'Como estas?']
```

3.4 Summary

This chapter gave a preview of ways to use both Python and NLTK for text processing. The main objective of the chapter was to get NLTK installed on your computer, and give you practice in using a few of its features.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

3.4.1 Quick Reference

Reference 3.4.1 Tokenize Text

```
# returns a list of tokens including punctuation
from nltk import word_tokenize
tokens = word_tokenize(text)
```

Reference 3.4.2 Tokenize Sentences

```
# returns a list of sentences
from nltk import sent_tokenize
sentences = sent_tokenize(text)
```

Reference 3.4.3 Remove Punctuation and Numbers

```
# returns a modified string; also lower cased
import re
text = re.sub(r'[.?!,:;()\-\n\d]',' ', raw_text.lower())
```

Reference 3.4.4 Stem Tokens

```
# returns a list of stemmed tokens (includes punctuation)
from nltk.stem.porter import *
stemmer = PorterStemmer()
stemmed = [stemmer.stem(t) for t in tokens]
```

Reference 3.4.5 Lemmatize Text

```
# returns a list of lemmas (includes punctuation)
from nltk.stem import WordNetLemmatizer()
wnl = WordNetLemmatizer()
lemmas = [wnl.lemmatize(t) for t in tokens]
```

Reference 3.4.6 Remove Stopwords

```
# returns a list of tokens that are not stopwords
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))
tokens = [t for t in tokens if not t in stop_words]
```

Reference 3.4.7 Timing Code

```
import timeit

start_time = timeit.default_timer()
# do something
stop_time = timeit.default_timer()
print('Time:', stop_time - start_time)
```

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

3.4.2 Practice

The following exercises will introduce you to features of NLTK.

Exercise 3.1 Download raw text and perform some preprocessing.
1. Download any book from Project Gutenberg. See NLTK Book Chapter 3 for an example.
2. Perform word tokenizing.
3. Find the longest word in the text.
4. Perform sentence tokenizing.
5. Find the longest sentence in the text.

■

Exercise 3.2 Practice regular expressions. See NLTK Book Chapter 3 for examples.
1. Create a regular expression to match a phone number in form: 999-999-9999
2. Create a regular expression to match a phone number as 9 continuous digits
3. Create a regular expression to match a phone number in form: 999.999.9999
4. Create a regular expression to match a phone number in form: (1) 999-999-9999

■

3.4.3 Going Further

Explore the NLTK open source code in GitHub to see examples of best coding practices in an object-oriented paradigm. For example, see this code: `https://github.com/nltk/nltk/blob/develop/nltk/chunk/named_entity.py`

4. Linguistics 101

Linguistics and NLP are closely bound together. In fact, NLP is sometimes called Computational Linguistics. Linguistics is the study of human language, and is a fascinating field of study. Many universities offer advanced degrees in Linguistics. The goal of this chapter is more modest: To familiarize the reader with terminology and concepts that are frequently used in NLP. Language is complex. Therefore, it is not surprising that linguists study language from so many aspects, from the letters that make up written words, the sounds that make up spoken words, sentence structure, meaning, culture, and more. To further linguistics literacy, the next few paragraphs explore terminology that every NLP practitioner should understand.

4.1 Levels of language

Commonly used linguistic terms are important things to know so that you can read NLP papers and understand lectures at NLP conferences or online. The first two terms concern the sounds of human language. **Phonetics** is concerned with the different sounds that human language contains and the articulation skills that people use to produce them. **Phonology** is more specific to a given language. For example, English has about 45 **phonemes** (unique sounds) that are combined into syllables within words. Phonemes include the different sounds letters can make as well as consonant blends (like the *bl* in blends), and dipthongs (the combination of two vowel sounds). The letter c has one phoneme in the word *cat* and another in the word *ace* and others when combined with other letters as in *chip, clip, crypt*, and so forth. The study of phonetics and phonology is very important for speech recognition tasks. In this book, however, the focus is more on written language. The term **orthography** refers to the written equivalent of phonemes, punctuation, etc.

A **homonym** is a word that sounds the same or is spelled the same as another word, but has a different meaning and different word origin. An example of a homonym is *pen*. One meaning of pen is the writing instrument; this meaning comes from the Latin word ‘penne’, for feather. Another meaning

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

of pen is an enclosure for livestock; this meaning comes from an Old English word ‘penn’ of unknown origin. A **homophone** is word with the same pronunciation but different meaning as another word, such as ‘not’ and ‘knot’. A **homograph** is a word with the same spelling but not the same meaning and not necessarily the same pronunciation. An example is ‘bow’ which rhymes with ‘toe’ and ‘bow’ which rhymes with ‘cow’. The term **polysemy** refers to the fact that a word may have many meanings. In contrast to homonyms, polysemous words often have a similar origin. An example is ‘wood’ which can mean the product cut from trees, or an area with a lot of trees.

Words are shape shifters. Think of the words run, running, ran. The word *run* itself can have different meanings. For example, it may be a verb, as in *I like to run*, or a noun as in *I participated in the Fun Run*. The term **morphology** refers to the different shapes and meanings that a word can take on, while the term **morphemes** refers to the specific components that carry meaning. For example, *running* contains the atomic morpheme *run* as well as the affix *ing*. There are rules in every language that specify how we can correctly combine morphemes.

Every language has a **lexicon**, a set of words that are agreed to be legitimate words in that language. Information about words that may be included in a lexicon includes parts of speech, meaning, and pronunciation. The entries in a lexicon are **lemmas**, the base form of the word. The set of words in a given group of documents, a *corpus*, is often called the document *vocabulary*.

Words are combined into sentences using the rules specified by a language’s **syntax**. The term *grammar* is also used for these rules. Classic grammars are used for humans to speak and write their language correctly. The term *formal grammars* refer to specifications of a language that can be used computationally to check or generate language. Syntax supports meaning, or **semantics** but not always unambiguously. Take this classic example, read two ways. The first reading says that man would be lost without woman. The second says the opposite.

```
Woman, without her, man would be lost.
Woman without her man would be lost.
```

Meaning is not only conveyed in words but also in tone of voice and context. This is the domain of **pragmatics**. If a wife asks her husband, *Could you give me a hand?* she’s not asking for a literal hand, she’s not asking for applause, and she’s not asking if he could (but chooses not to) help her. Pragmatics also includes things like sarcasm. If a mother looks at her son’s room and says *Beautiful*, it could mean that he did a beautiful job organizing his room, or it could mean the opposite when spoken sarcastically.

Linguistics examines much more than just the structure of language. Linguists study how languages are formed, how children learn them, how languages vary among peoples, and much more. In a fascinating TED Talk[1], Lera Boroditsky talks about how the particular language we speak influences our thought. Note that not all linguists believe that your native language influences your thought process.

4.2 Word categories

The parts of speech, POS, are fairly stable across languages. The main parts of speech that are recognized in English are the noun, verb, adjective, adverb, pronoun, preposition, conjunction and article (also called determiner). Some lists also include the interjection. Linguists divide the parts of speech into two categories: closed and open. Open categories are open to additions. We see this with nouns, verbs, adjectives and adverbs, as well as pronouns. For example, *google* has become a new

[1]https://www.youtube.com/watch?v=RKK7wGAYP6k

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

proper noun as well as a new verb. The closed categories don't change, or change very slowly over long periods of time. Closed categories include preposition, conjunction, article and pronouns.

4.2.1 Nouns

Nouns refer to people, places, things, and concepts. These things referred to are called *entities* in NLP. Nouns can be classified in various ways: concrete v. abstract, common v. proper, animate v. inanimate.

Nouns have inflection, but this varies by language. Examples of inflection:

- Number: singular or plural; ex: child, children
- Gender: feminine, masculine, neutral; ex: Spanish chico, chica
- Case: nominative (subject), genitive (possessive), accusative (direct object), dative (indirect object)

English has regular plurals (dog/dogs) and irregular plurals (child/children). English generally doesn't specify gender but languages like Spanish do: hijo for son and hija for daughter. If you've had a Latin class in your background, you are familiar with case. English has lost most of its cases. You still see case in genitive case (possessive pronouns) and the nominative and genitive cases for personal pronouns. English pronouns still carry case but nouns, not so much.

4.2.2 Verbs

Another open class of words is the verb. Verbs inflect according to the person, and whether the subject is plural. Verbs also inflect according to tense (present, past, etc.), aspect (progressive, perfect), voice (active or passive), and modality (should, will, etc.). Ordinary verbs are the common verbs that describe action or state of being. These are often augmented with auxiliaries (do, have, be) and modals (can, should, must, etc.). The verb 'be', sometimes called a *copula* (or linking verb), links the subject with further information about the subject, as in 'She is clever.'

4.2.3 Pronouns

There are many categories of pronouns. The personal pronouns: I, you, he, she, they, etc., vary by number and gender. Closely related to these are the possessive pronouns: my, mine, their, theirs, etc. The reflexive pronouns refer to: myself, yourself, ourselves, etc. The demonstrative pronouns are: this and that. Interrogative pronouns are: who, what, where, etc. And finally there are indefinite pronouns: one, someone, something, etc.

Although pronouns are in the closed category and have not changed for centuries, in modern times we are seeing a push for gender-neutral pronouns in English. In recent years, the pronoun 'they' is often used to specify a person who may be male, female, or other.

4.2.4 Adjectives and adverbs

Adjectives modify nouns and are often derived from nouns or verbs. Adjectives can be ordinary (large, red), possessive (driver's license) or proper (Republican candidate). They are inflectional for comparative and superlative senses in English, as in: fast, faster, fastest. Numerals are special types of adjectives and are sometimes used as nouns. Examples are: first, many, two hundred, and so forth.

Adverbs modify verbs and are often derived from adjectives, ex: happy, happily. Adverbs can make the verb more specific as to time, place, manner and degree. Additionally, there are wh-adverbs for questions such as: why, when, where, how, etc.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

4.2.5 Other POS

The nouns and pronouns, verbs, adjectives and adverbs discussed above carry most of the semantic weight of sentences. But there are other parts of speech that hold the sentence together syntactically and clarify the meaning of sentences. Prepositions occur before noun phrases and give important information about it, as in ‘on’ in ‘on the table’. Particles are words that accompany verbs to give a more specific or sometimes entirely different meaning, as in ‘take up’ a hobby. A verb plus a particle is called a *phrasal verb* because the verb’s meaning is not completely specified with the verb alone.

Determiners such as ‘the’ and ‘a’ help determine the specificity of a noun. For example, ‘the book’ refers to one and only one book while ‘a book’ is more general. Determiners also include words like ‘these’ and ‘those’.

Conjunctions combine words together to form more complex structures. Coordinating conjunctions are often used to combine words together as in ‘peas <u>and</u> carrots’, as well as to combine independent clauses in a single sentence: ‘I like peas <u>but</u> he likes carrots.’ Subordinating conjunctions make one clause dependent on the main clause: ‘He doesn’t like peas <u>because</u> they are green.’

And finally there are a few words that are in a class by themselves. The existential ‘there’ as in ‘There is an answer’ serves the function of a subject. Negation words like ‘no’ and ‘not’ serve to negate the meaning, as in ‘There is no answer.’

4.3 Classifying languages

Commonly, language are classified according to their origin, such as Indo-European, Romance languages, and so forth. Linguists also like to classify languages based on morphological features. There are 3 main groups: analytical languages, inflective languages, and agglutinative languages.

Analytical languages like English use a lot of function words to convey precise meaning: *I would have liked the party*. Other analytical languages include French, Italian, Japanese and Chinese. Inflective languages uses affixes (prefix, suffic, infix). Examples of these languages include many Slavic languages and Arabic. Agglutinative languages connect together unchanging morphemes to convey meaning. Turkish and Hungarian are examples. In English we combine words to convey meaning, in Turkish, you can convey meaning by gluing morphemes together, which results in delightfully long words. According to the translator’s notes in the afterward of Orhan Pamuk’s novel *The Black Book*, "Apparently, they were inside their houses" is expressed as a single word in Turkish.

4.4 Language Origins

No other creatures on Earth have the language capabilities demonstrated by humans. That is where agreement among linguists ends. There are competing theories about language origins. Next, we briefly compare the instinct school of thought, largely associated with the linguist Noam Chomsky, and the culture school of thought.

Chomsky argues that languages is too complex, and mastered by children so quickly that it cannot be a learned skill like riding a bicycle; there must be a genetic program for learning language like birds know how to build a nest. Steven Pinker likens it to upright posture, every baby learns to crawl, then sit up, then stand up. It is not culturally transmitted.

One of the proponents of the cultural school of thought is field linguist Daniel Everett. Everett argues that language is not a recent evolutionary mutation, but that communication and language became incrementally more complex, beginning as far back as with Homo Erectus, and was made universal among humanoids as Homo Erectus migrated across the globe. Everett views increasing

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

social complexity to be the driving force for language evolution. As we evolved, the need to organize and coordinate human actions for survival increased.

The point of agreement is that something within man evolved to enable language complexity. The disagreement is if humans in isolation would spontaneously develop language without inspiration from another culture. We've never found a group of humans without language, so there is no definitive evidence either way.

Researchers are learning more about how humans process language in the lab. Using magnetoencephalography, researchers at NYU were able to identify distinct cortical activity that concurrently tracked auditory input (stripped of acoustic cues) at different hierarchical levels: words, phrases, sentences. In other words, a hierarchy of neural processing underlies grammar-based internal construction of language. This exciting research may in the future be able to tease out what information these hierarchical processes actually encode, although at present we cannot say whether these processes were innate abilities or learned through experience.

4.5 Language complexity

Linguists also study many features of language that prove challenging for NLP applications. For example, the term **zeugma** is a literary or rhetorical device in which a word, most typically a verb, extends to two distinct phrases of a sentence. In the following quote from Francis Bacon, 'make' is not repeated because it is implied to extend through the subsequent phrases:

> Histories make men wise; poets, witty; the mathematics, subtle; natural philosophy, deep; moral, grave; logic and rhetoric, able to contend.

In the quote above, zeugma made the quote more crisp and concise. Sometimes zeugma is used either for humorous effect, or to make people think. Here is a great example from Star Trek:

> You are free to execute your laws, and your citizens, as you see fit.

Above, 'execute' is the link between laws and citizens, but with startling implications. A zeugma can make humans stop and think, or laugh, but proves challenging for NLP applications to 'understand'.

Other literary devices that prove challenging to NLP applications are metaphors, for example: *Love is not a bed of roses*, and similes: *The well is dry as a bone*. In short, any type of figurative or idiomatic speech is problematic for NLP.

4.6 Summary and Exploring Further

The goal of this chapter is to introduce vocabulary that is commonly used in NLP research. These terms will become second nature as you read and learn more about NLP. Here are some good resources for extending your learning:

- A talk by Steven Pinker about language and the brain `https://www.youtube.com/watch?v=Q-B_ONJIEcE`
- High-level linguistics topics from the University of Edinburgh `https://www.youtube.com/user/edinburghLangScience/`
- Daniel L. Everett, *How Language Began* `https://www.youtube.com/watch?v=1hVijQZLEeM`
- Steven Pinker, *On Chomsky* `https://www.youtube.com/watch?v=vg4BtHuLtCY`

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

II

Part Two: Words

Part Two

Part Two looks at ways we can learn about a text by just looking at its words. Hearing or reading a word evokes a meaning in your mind, but how? All we know is that a set of neurons will become associated with a word with repeated exposure to the word. Children begin to learn words before they are one; between years one and two they learn words at the rate of one every two hours; by the time they graduate high school they know about 60K words.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

5. Words and Counting

Words are the first bits of language we learn as children. In some magical moment we have long forgotten, it dawned on us that objects and feelings had names, and that we could communicate our wants and needs through words: cookie, thirsty. The ability of children to learn words is perhaps something we take for granted until we try to teach words to computers. A child may only need a few examples to learn the word ‘elephant’ and associate it with the animal’s shape. A machine learning algorithm may need hundreds of thousands of examples.

The linguistics chapter discussed morphology - the shape shifting nature of words. Words can take on affixes but something of the core remains. Take the word *anti-inflammatory*. The root is *inflame* which takes its origin in Latin words for in and flame. Another morph of the word is *inflamation*, the state of being *inflamed* (yet another morph). The morph *inflamatory*, then means something that inflames, and adding the prefix *anti-* means something that does the opposite. You understand the shades of meaning indicated by these different shapes because somewhere along the line you learned the rules of English word formation. You know that changing a word’s shape can change its category, its part of speech. The verb *inflame* is modified to a noun as *inflammation* and an adjective as *inflamed*.

This chapter explores what we can learn about text from the words themselves. First, simply counting word frequencies can yield important information about text.

5.1 Word Counts and Text Analysis

This chapter performs some text analysis with text samples from NLTK. If you have not installed the NLTK book data (see Chapter 3), you should do that so that you can follow along with the code. After executing the code below, you will see a list of text files.

```
from nltk.book import *
```

To see the list again at any time in the Python session, type `texts()` at the console.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

5.1.1 Counting words

The NLTK texts have already been tokenized. The first line of code below displays the first 9 tokens of text4, the Inaugral Address Corpus. The second line of code uses Python's count() function to count the occurrences of 'citizen' in text4. The NLTK texts are a Text object, with some nice built-in functions, discussed in the next section. This section shows what you can do with Python.

```
text4[:9]
['Fellow', '-', 'Citizens', 'of', 'the', 'Senate', 'and', 'of', 'the']
text4.count('citizen')
56
```

5.1.2 Text Analysis with len(), set(), and sorted()

Looking at the first few tokens above, you can see that the text is not in lowercase and that there is punctuation. The text needs some preprocessing. The code chunk below is from notebook 5.1_words1.py in the GitHub. The first thing it does it use a list comprehension to lowercase all the tokens in text4. Then it uses Python's len(), set(), and sorted() functions to explore the text. Running this code shows that text4 has 149,797 tokens and that 9216 of them are unique. Printing the first few unique tokens shows that they are all punctuation.

Code 5.1.1 — Python. Word Counts with Python Functions.

```
# lowercase the text
tokens4 = [t.lower() for t in text4]

print("\nThe number of tokens in text4: ", len(tokens4))
The number of tokens in text4:  149797

set4 = set(tokens4)  # find unique tokens
print("\nThe number of unique tokens in text4:", len(set4))
The number of unique tokens in text4: 9216

print("\nThe first 5 unique tokens in text4:", sorted(set4)[:5])
The first 5 unique tokens in text4: ['!', '"', '";', '"?', '$']
```

5.1.3 Text Analysis with Preprocessing

The next code chunk continues the Python script above. The first thing the code below does is to use another list comprehension to get rid of punctuation, numbers, and stop words. Of course, it would have been more efficient to do all the preprocessing in one list comprehension, but they are separated here to show the effects of each type of preprocessing.

After getting rid of punctuation and stop words, the text now has 64,336 important words, 8973 of which are unique. By removing non-alpha tokens and stop words, the number of important tokens is less than half of the number of tokens in the original text. The number of unique words decreased by about 3%.

There is no standard sequence of preprocessing steps for NLP projects. Every project is unique and may require different approaches. In this example, stop words are removed because the idea is to learn about the content words used in the text. For other NLP applications, you may not want to

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

remove stop words. Stop words are largely function words, the words in English that glue a sentence together syntactically. The way that an author uses these functions words can give insight into who the author is, an NLP task called *authorship attribution*. Function words can also reveal something about the psychological state of the author, as demonstrated in research by Dr. Pennebaker at UT Austin.[1]

Finally, the code chunk below calculates lexical diversity by dividing the number of unique tokens by the total number of tokens. Lexical diversity is a measure of the variety of vocabulary used. Different linguists have proposed more complex methods of computing lexical diversity than the simple formula used here, but without universal agreement.

Code 5.1.2 — **Python.** Word Counts after Preprocessing.

```
# get rid of punctuation and stopwords
tokens4 = [t for t in tokens4 if t.isalpha() and
           t not in stopwords.words('english')]
print("\nThe number of important words in text4:", len(tokens4))
The number of important words in text4: 64336
print("\nThe number of unique important words in text4:",
      len(set(tokens4)))
The number of unique important words in text4: 8973

print("\nThe first 10  important words in text4:", tokens4[:10])
The first 10  important words in text4: ['fellow', 'citizens', 'senate',
    'house',  'representatives', 'among', 'vicissitudes', 'incident',
    'life', 'event']

# lexical diversity
print("\nLexical diversity: %.2f" % (len(set4) / len(tokens4)))
Lexical diversity: 0.14
```

5.1.4 Text Analysis with Lemmas

The problem with the counts above is that it didn't take into consideration related words. For example, 'democracy' and 'democratic' would count as separate words, even though they have the same root. The code chunk below first lemmatizes the tokens.

The next part of the code below uses a dictionary comprehension to make a dictionary of lemma:count entries for unique lemmas. Python dictionaries are unordered. The code sorts key:values into a list, sorting by value, in reverse order, highest counts to lowest.

The 5 least common words are interesting. Some, like 'afresh' and 'thence' sound a bit archaic. The words 'underneath' and 'imperiled' sound like negative words whereas most inaugural speeches are optimistic. It is perhaps not surprising that the word 'cathedral' is only used once since America was founded on separation of church and state, among other first principles.

Most of the 5 most common words in the inaugural texts are no surprise: government, people, nation, and state. But what is that 'u' doing there? A first guess is that the lemmatizer shortened 'us' to 'u', which is not what you would expect. We will dig into that more in the next subsection.

[1] `https://www.researchgate.net/publication/237378690_The_Psychological_Functions_of_Function_Words`

The code to sort by value needs a bit of explanation. A helpful discussion of sorting can be found in the Python docs: `https://docs.python.org/3/howto/sorting.html`. The first thing to note is that sort() sorts in place whereas sorted() returns a new sorted object. The mylist.sort() function is only defined for lists, whereas sorted() works for any iterable object. A `key` parameter can be specified that gives the sort or sorted function more information about how you want the items sorted. In the next Code chunk box, the key is: `key=lambda x:  x[1]`. This makes the sort work on the values not the key, but how? Let's look at a really simple dictionary to see:

```
>>> mydict = {'a':5, 'b':7}
>>> sorted(mydict.items(), key = lambda x: x[1], reverse=True)
[('b', 7), ('a', 5)]
```

The dictionary created above has two items, its keys are letters and the values are integers. The sorted() function returns a list of tuples, not a dict! That is key to understanding how this works. Remember that dicts have no order, but you can extract a sorted list of tuples where the first item in the tuple is the key and the second is the value. That's why x[1] sorts on values. Note that this is just one way to sort the dictionary.

Code 5.1.3 — Python. Dictionary Counts of Lemmas.

```
# get the lemmas
wnl = WordNetLemmatizer()
lemmas = [wnl.lemmatize(t) for t in tokens4]
# make unique
lemmas_unique = list(set(lemmas))  # ?
print("\nThe number of unique lemmas in text4: ", len(lemmas_unique))
The number of unique lemmas in text4:  7935

# make a dictionary of counts
counts = {t:lemmas.count(t) for t in lemmas_unique}
print('citizen', counts['citizen'])

# print most/least common words
# dicts are unordered so we sort it and put it in a list of tuples
sorted_counts = sorted(counts.items(), key=lambda x: x[1], reverse=True)
print("5 most common words:")
for i in range(5):
    print(sorted_counts[i])

5 most common words:
('government', 651)
('people', 623)
('nation', 515)
('u', 478)
('state', 442)
```

5.1.5 NLP is not Perfect but is Perfectable

The code above discovered ‘u’ in the 5 most common lemmas. This reveals a few harsh truths about NLP work:

- NLP results are not perfect because language is messy.
- NLP results are not perfect because the available tools are not perfect.
- NLP is perfectable, meaning that results can be incrementally improved with hard work, patience, and persistence.

Let’s look at how we might debug what has happened. Debugging at the console is nice because you can examine the results after each step. We know that ‘u’ is not a word, so the lemmatizer must have shortened some word to ‘u’. Likely candidates are short words that start with u. In the code below, these are identified by a list comprehension, and made unique with set(). There are two candidate problem words: ‘up’ and ‘us’. The code below ran the lemmatizer on ‘us’ and sure enough, it returned ‘u’.

```
>>> x = set([t for t in text4 if t.startswith('u') and len(t) < 3])
>>> x
{'up', 'us'}
>>> wnl.lemmatize('us')
'u'
>>>
```

So now that the source of the error has been found, what should be done? There are a few options to consider:

- Use a customized list of stop words, and include ‘us’
- Remove words of length 1 from the set of unique lemmas
- Try different lemmatizers to see if better results can be achieved

Looking at the top 50 words instead of just the top 5 shows some words that may be candidate stop words, like: must, may, every, one. What to include in the stop words will be a critical decision, which will also be domain and problem dependent. To make a custom stop word list, you can start with NLTK stopwords and add your own:

```
>>> from nltk.corpus import stopwords
>>> stop_words = stopwords.words('english')
>>> stop_words += ['may', 'must', 'every', 'one']  # add more stop words
```

The above code is just an example, and is not meant to imply that these words should be added to the list of stop words.

5.2 NLTK Text Object and Methods

NLTK has a concordance function for Text objects, which lists every occurrence of a word with its surrounding context. Try the following two lines at the console. The code below gives us insight into the two novels, *Moby Dick*, published in 1851, and *Sense and Sensibility*, published in 1811. The word ‘handsome’ only occurs 3 times in text1, and was used exclusively to describe objects as well-made, or quantities as substantial. These conform to the various dictionary definitions of the word. In text2, the word is used to convey these meanings as well, but is used much more frequently to refer to attractive men and women.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

```
>>> text1.concordance("handsome")
Displaying 3 of 3 matches:
ain had provided the chapel with a handsome pair of red worsted man - ropes
tics in all things ), is much more handsome and becoming to the boat , than
rtain cook of the court obtained a handsome reward for inventing an admirab
>>> text2.concordance("handsome")
Displaying 25 of 27 matches:
d pounds : it would be liberal and handsome ! It would be enough to make th
set of breakfast china is twice as handsome as what belongs to this house
. . .
```

The concordance() function cannot be used on raw text; it only works on an nltk Text object. However you can convert text to a Text object and then run concordance.

```
>>> from nltk.text import Text
>>> textList = Text(tokens)
>>> textList.concordance('handsome')
```

There are many more such interesting functions in NLTK that we will explore as we go. The take-away is that when you consider writing a function for text analysis, you should first explore libraries such as NLTK to see if there is already a function that meets your needs.

5.2.1 Word Frequencies

Counting word frequencies is an important aspect of many NLP projects. For example the frequencies of certain words may provide evidence that an email is spam. We can use some Python functions for counting, as well as some NLTK functions, as we see in the next Code block. We are looking at the book of Genesis which has a lot of 'begat' words. How many? Try these at the console.

Code 5.2.1 — Python and NLTK. Word Frequencies.

```
# explore frequencies with Python
text3.count('begat')  # 67
text3.count('begat') / len(text3) * 100  # almost 0.15%

# explore frequencies with NLTK
fdist_text3 = FreqDist(text3)
fdist_text3.most_common(10)

# remove stop words
from nltk.corpus import stopwords
text_lower = [t.lower() for t in text3]
text3_content = [t for t in text3 if not t in stopwords.words('english')
   if t.lower() not in stopwords.words('english')]
fdist_text3 = FreqDist(text_lower)
fdist_text3.most_common(10)
```

After removing stopwords, we see that common words include 'thee' and 'thou'. This tells us that we should augment our stop words with common words in older forms of English for this text.

The code above used the Text method FreqDist() to create a frequency distribution. The frequency distribution can be plotted with the code below, as shown in Figures 5.1 and 5.2.

```
fdist1.plot(50, cumulative=True)  # 50 most common words

fdist1.plot(50, cumulative=False)  # 50 most common words
```

Figure 5.2 shows the frequency distribution without cumulative frequencies, which shows that the frequency of a word diminishes with its place in the sorted list of common words. The cumulative frequency in Figure 5.1 shows how the number of words is a logarithmic curve as new words are put into the distribution.

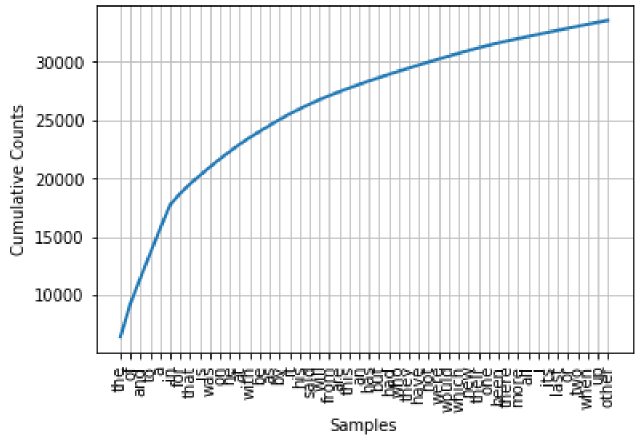


Figure 5.1: Frequency Distribution

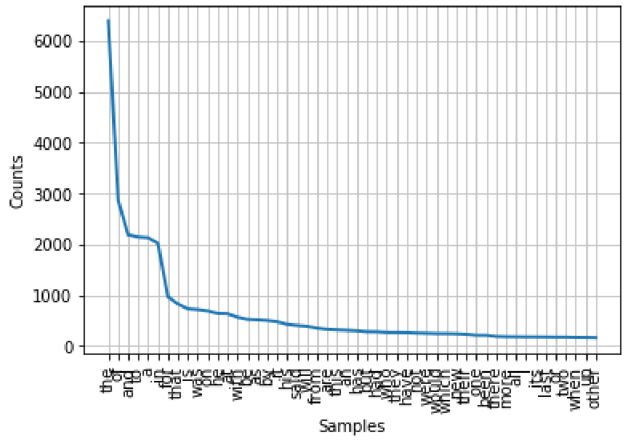


Figure 5.2: Cumulative Frequency Distribution

5.2.2 Zipf's Law

The frequency and cumulative frequency distributions of words in a corpus, plotted above, show the logarithmic nature of word frequencies. This phenomenon has been observed across diverse corpora and was first proposed by linguist George Zipf in the 1930s. Zipf's law observes that in a sufficiently large corpus, the frequency of any word is inversely proportional to its rank in a frequency table. That is, the nth most common word will have a frequency proportional to 1/n. Another way of saying this is that there is some constant, k, that quantifies the proportionality. Zipf's law is empirical in that it has been observed but not explained theoretically.

$$f \times r = k \tag{5.1}$$

The following code explores Zipf's law using the Brown Corpus, which is included in the NLTK corpora. The Brown Corpus of American English was the first million-word electronic corpus. It was compiled in 1961 by W. N. Francis and H. Kucera at Brown University in Providence, RI. The corpus has 500 texts of about 2,000 words each, sampled from 15 different text categories.

The code block below shows the code that created the distribution and cumulative distribution plots in Figures 5.1 and 5.2.

Code 5.2.2 — Python and NLTK. Exploring Zipf's Law.

```
import nltk
from nltk.corpus import brown

# extract just the news category
news_text = brown.words(categories='news')
# limit words to alpha words and lowercase them
news_text = [w.lower() for w in news_text if w.isalpha()]

# create a frequency distribution and plot
fdist_brown = nltk.FreqDist(news_text)
fdist_brown.plot(50)
fdist_brown.plot(50, cumulative=True)
```

Fagan and Gencay observed in *An Introduction to Textual Econometrics* that about 135 of the most frequent words equal half of the total number of words. Using the Brown Corpus again in the code below shows that the top 135 represent 49% of the total words.

```
most_common = fdist_brown.most_common(135)
total = 0
for w, f in most_common:
    total += f
print('The top 135 words = ', total/len(news_text),
       'percent of the number of words.')
# output:
The top 135 words =  0.4904262703142577 percent of the number of words.
```

The constant k for the news category of the Brown corpus is 0.11, a fairly typical value for a diverse and large corpus.

NLTK also has a hapaxes() function which finds rare words. The term *hapax legomenon* is used in NLP to refer to words that occur only once in a given context. Most often these words are nouns.

```
haps = fdist_brown.hapaxes()
haps[:5]
['presentments',
 'durwood',
 'pye',
 'handful',
 'outmoded']
```

Zipf's law also predicts that about half of the words in a corpus will be rare words. In the Brown Corpus, the percentage of rare words for the news category was 48.6%.

5.2.3 Heap's law

The following code chunk processes a tokenized text sequentially, adding 1 to the dictionary count if the word has already been encountered, or adding a new dictionary item for words that have not yet been encountered:

```
vocab = {}
for token in tokens:
    if token in vocab:
        vocab[token] += 1
    else:
        vocab[token] = 1
```

This can be done with simpler syntax, using a dictionary comprehension and the count() function:

```
# make a dictionary of word counts
vocab = {t:count(t) for t in tokens}
```

As the first few words are encountered, most of them will be new. As the text is processed, fewer and fewer new words will be found. Heap's law quantifies this phenomenon. The formula below indicates that vocabulary growth is a function of some constant, k, multiplied by n, the total number of words, to the power of another constant, b.

$$v = k \times n^b \tag{5.2}$$

The implication is that as more documents are added to a corpus, fewer new words will be found. The constant k varies by corpus, typically between 30 and 100. Word normalization and spelling correction would lead to a smaller k value. The power b is around 0.5 most of the time. Typically, the number of unique words is a function of the square root of the vocabulary size. This is visualized in Figure 5.3.

Common sense would tell us that the curve should not continue as a square root curve, that it should level off at some point because there are a finite number of words in a language. Surprisingly, this does

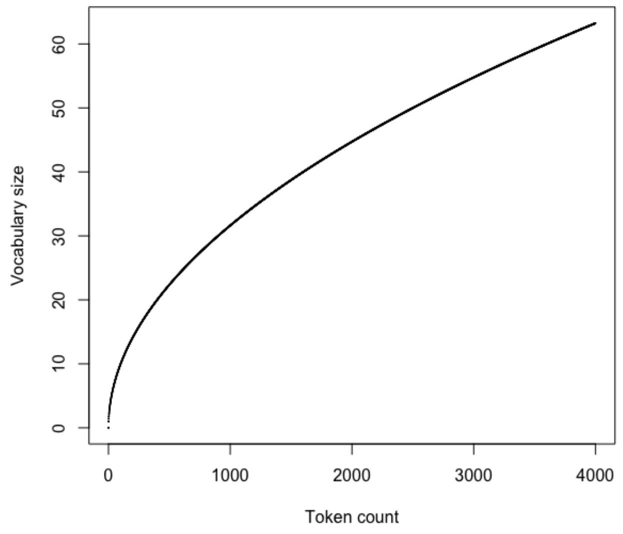


Figure 5.3: Heap's Law

not happen. This is because new words include words not found in the lexicon, such as proper nouns, urls, and so forth.

An example from *Introduction to Information Retrieval* by Manning, Raghavan and Shutze[2] shows a Heap's law prediction over a 1 million word Reuters corpus.

$$44 \times 1,000,010^{0.49} = 38,3232 \tag{5.3}$$

The actual number of unique tokens is 38,365. Heap's law does seem to capture the growth of vocabulary as more text is encountered.

The importance of Heap's law is found in applications that index terms, such as search functions. Heap's law tells us that our index will be forever growing by a predictable amount.

Heap's law is sometimes called Herdan's law in honor of Gustav Herdan who is more accurately credited with the discovery than Harold Stanley Heaps.

5.3 Summary

This chapter looked at the building blocks of language: words. A first step in text analysis is preprocessing, which often includes: lower-casing text, removing punctuation and numbers, removing stop words, and normalizing text by stemming or lemmatization.

Word frequencies can give insight into what text is about. NTLK has functions for extracting and plotting word frequencies. Word frequencies form a logarithmic curve, an observation first noted by George Zipf in the 1930s. Another interesting observation about words is Heap's law, which notes that

[2]https://nlp.stanford.edu/IR-book/html/htmledition/heaps-law-estimating-the-number-of-terms-1.html

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

as more text is explored, fewer new words will be found, and that the rate of new words is proportional to the square root of the number of tokens in the text.

5.3.1 Quick Reference

Reference 5.3.1 Tokenizing

```
from nltk import word_tokenize

tokens = word_tokenize(text)   # tokenize text
tokens = [t.lower() for t in tokens if t.isalpha()]  # get rid of non-text
```

Reference 5.3.2 Removing stop words

```
from nltk.corpus import stopwords
stopwords = stopwords.words('english')

tokens = [t for t in tokens if t not in stopwords]
```

Reference 5.3.3 Normalizing Text

```
from nltk.stem.porter import *
from nltk.stem import WordNetLemmatizer

# stem tokens
stemmer = PorterStemmer()
stemmed = [stemmer.stem(t) for t  in tokens]

# lemmatize tokens
wnl = WordNetLemmatizer()
lemmatized = [wnl.lemmatize(t) for t in tokens]
```

Reference 5.3.4 Find most common words

```
# make a dictionary of word counts
counts = {t:count(t) for t in tokens}

# get a sorted list of tuples: (token, count)
sorted_counts = sorted(counts.items() key=lambda x: x[1], reverse=True)
```

5.3.2 Practice

Exercise 5.1 Use the built-in corpora for this exercise. See NLTK Book Chapter 1 for examples.

1. Load the NLTK corpora
2. Create a frequency distribution on a corpus
3. Create a frequency distribution plot on a corpus

■

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Exercise 5.2 Repeat Exercise 5.1 with a different corpus and compare the results. What does this tell you about words and corpora? ■

5.3.3 Going Further

To explore more, check out the following resources:

- An Introduction to Textual Econometrics by Fagan and Gencay `https://www.taylorfrancis.com/books/e/9780429141898/chapters/10.1201/b10440-9An`
- Video about Zipf's Law `https://www.youtube.com/watch?v=fCn8zs912OE`
- Article in Gizmodo about Zipf's law and the size of cities `https://io9.gizmodo.com/the-mysterious-law-that-governs-the-size-of-your-city-1479244159`

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

6. POS Tagging

In NLP applications, it often helps to know the part of speech (POS) of a word. Chapter 4 discussed the parts of speech identified by traditional grammars: noun, pronoun, verb, adjective, adverb, preposition, conjunction, determiner, particle. However, computational approaches use dozens of categories, as shown in Table 6.1. Determining parts of speech is important in NLP tasks because it narrows the role that a word can play in a sentence. For example, the subject of a sentence cannot be an adverb. This chapter first looks at using the POS tagger in NLTK, then discuss techniques for POS tagging.

6.1 POS Tagging with NLTK

NLTK has a POS tagger that takes as input a list of tokens, and returns a list of tuples. In each tuple, the word is the first item, and the part of speech is the second item.

Code 6.1.1 — NLTK. POS Tags.

```
import nltk
from nltk import word_tokenize
text = "This is a sentence. This is another one."
tokens = word_tokenize(text)
tags = nltk.pos_tag(tokens)
print(tags)
[('This', 'DT'), ('is', 'VBZ'), ('a', 'DT'), ('sentence', 'NN'),
('.', '.'), ('This', 'DT'), ('is', 'VBZ'), ('another', 'DT'),
 ('one', 'NN'), ('.', '.')]
```

Any tag that starts with 'V' is a verb, there are tags for different verb tenses. Any tag that starts with 'N' is a noun. There are different tags for plural, singular, and proper nouns. Adverbs are 'RB' and

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

adjectives are ‘JJ’. Determiners are ‘DT’. There are many different tagsets used in NLP, with varying numbers of tags. The Brown tag set has 87 tags and the Penn TreeBank tag set has 36 categories for words as well as categories for punctuation. A list of the Penn TreeBank tags is provided at the end of the chapter. This is the tag set used by NLTK.

6.2 Determining Tags

POS tagging has been around in NLP for a long time, using various approaches. Early approaches were rule-based. Later, Hidden Markov Models (HMMs) were found to improve accuracy. Eventually, machine learning approaches were developed that improved accuracy even more. All approaches input a list of words and output the list with each word given the most appropriate tag.

6.2.1 Rule-based Tagging

A rule-based tagger starts with assigning a list of possible tags for each word, based on a lexicon or dictionary. Then a series of rules are applied to find the most likely tags. Some rules use the morphology of words. For example, ‘loveliest’ ends in ‘est’ with the stem ‘loveli’ and therefore it must be a superlative adverb. Rules specify the constraints inherent in English grammar that help eliminate certain tags. Such taggers end up with thousands of rules. Rule-based systems are useful, but known to be brittle. Human language is too complex and varied to be represented by a set of rules. However, these approaches gave reasonable results before more statistical methods were applied.

6.2.2 HMM and the Viterbi Algorithm

A hidden Markov model approach learns transition probabilities. This can be done by starting with bigrams, a sliding window of size 2 over a sentence, plus start and end tags. Below we see a sample sentence, followed by the most likely tags.

```
<s> John is a handsome man. </s>
<s> NNP  VBZ DT JJ NN . </s>
```

The Viterbi algorithm uses dynamic programming to find maximum transitions between states. Dynamic programming is a technique that reduces run time by storing solutions to sub-problems so that those intermediate solutions don’t have to be recomputed. The algorithm has utility in many applications. Applied to parsing, the Viterbi algorithm finds the most likely parse of a sentence. Each input (see the list below) will be described in terms of its purpose, but also how it could be implemented in a Python program. A training file is first read in with word/POS tags for thousands of sentences.

Parsing with the Viterbi algorithm needs the following inputs:

- Observations - a tokenized sentence; the series of tokens is what has been observed
- States - dozens or more parts of speech tags found in the training data
- Starting probability - start tag ‘S’ with a small negative probability
- Transition probabilities - a dictionary of probabilities of transitioning from one POS to another, learned from counts in the training data
- Emission probabilities - a dictionary of probabilities of POS for every word in the training data

Let’s say we have 50 POS states, and a sentence of length 10, which is 10 transition states. Exploring every possible transition would require calculating 50^{10} probabilities. The insight of the Viterbi algorithm is that for each transition t to $t+1$, there is one transition (learned from the data) that is most probable. This reduces the number of calculations to $10 * 50^2$. That’s still a lot, but at least

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

it's not exponential. There are a few variations of the algorithm, some iterative and some recursive. Andrew Viterbi has a nice overview in a blog post. [1]

The original application of his algorithm was preventing errors in digital communication. Others later clarified the approach using either a trellis model, or a Markov model. The algorithm is used in mobile phones and enables an increase in the number of devices on a given frequency band. The algorithm has found use cases in genome sequencing, search engines, and more. The algorithm can be applied to any problem where underlying states and transition probabilities can be determined.

The central formula is:

$$M_j(k) = Max_i(M_i(k-1) + m_{ij}(k)) \tag{6.1}$$

In this formula, $M_j(k)$ is the sum of metrics that reach state j at time k. In terms of the parsing problem, the sum of probabilities (log likelihoods) for POS j at this token in the sequence. And m_{ij} is the metric for the transition of state i to state j, or for transitioning from one POS to the other. This probability will be set to a large negative value if it does not exist. The Max() portion of the equation selects the most likely path leading from time $k-1$ to time k from all possible paths that have been identified.

The state metrics are computed at each step from state to state. Further, the history of the path so far must be stored as well. Running the sentence 'The price was depressed.' as a test sentence through the algorithm, shows how the most likely POS is chosen while iterating over the tokens.

```
sentence:  The/DT price/NN was/VBD depressed/VBN ./.
iterations:
['DT']
['DT', 'NN']
['DT', 'NN', 'VBD']
['DT', 'NN', 'VBD', 'VBN']
['DT', 'NN', 'VBD', 'VBN', '.']
```

6.2.3 Statistical Approaches

In 1992 Eric Brill introduced a tagger based on rules (aka transformations) that were learned from annotated data rather than human expertise. Once the tagger is trained on this annotated data, new data can then be input for tagging. The Brill tagger can be seen as a hybrid between rule-based and statistical approaches. The advantage it offers overly purely statistical approaches is that it can transfer to a different genre of text or even different language, whereas a statistial approach would need entirely different annotated texts.

The Brill tagger initially assigns each tag to its most likely tag, according to a tagged corpus, without regard for any syntax or context. An initial procedure tags words that are not in the training corpus and are capitalized as nouns. Then it tags unknown words according to the last three letters, so that an unknown word like 'xyzbrous' would be classified as an adjective because of the 'ous' ending. Using this approach alone achieves about 92% accuracy. The next stage of the Brill tagger applies patches to improve the accuracy. The patches follow three forms:

1. If a word is tagged as **a** AND it is in context **C**, then change tag to **b**
2. If a word is tagged as **a** AND it has lexical property **P**, then change tag to **b**
3. IF a word is tagged as **a** AND a nearby word has lexical property **P**, then change the tag to **b**

[1] `https://magazine.viterbi.usc.edu/spring-2017/intro/the-viterbi-algorithm-demystified-a-brief-intuitive-appro`

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

As described in his 1992 ACL paper[2], the first rule learned by the system was `TO IN NEXT-TAG AT`, which says that if a word is tagged as TO and followed by a word tagged as AT, then change the TO tag to IN (preposition). This means that 'to' is tagged as TO (infinitive to) initially, but changed to IN when it is acting as a preposition. The system learned this rule and the others from the data itself. For each rule that the system wants to add, the reduction in error is computed if that rule were added. Only patches that significantly reduce errors are added. The Brill tagger achieved an accuracy of about 95% which was competitive with statistical approaches used at that time.

6.2.4 Machine Learning Approaches

Matthew Honnibal is the creator of the spaCy API. An overview of how to use spaCy is given in the GitHub. Honnibal has a blog post [3] where he gives an overview of how to create a pos tagger using the averaged perceptron model and a greedy approach. The post is worth reading, but here I'll just sketch out some main points.

A perceptron is a feed-forward neural network using a step function for the activation. The traditional perceptron sums the input features and learned weights plus the bias weight and outputs a binary value indicating if the sum surpassed a certain threshold. The average perceptron variation keeps a running average of the learned weights.

In the POS tagger described in the post, each token will have a set of features, including prefix, suffix, surrounding tags, and surrounding words. During training iterations, predictions are made for the tag, and weights are incremented if the predictions are correct, and decremented if they are not. At each iteration, average weights are used instead of final weights, so that later examples in the training don't skew the weights. Honnibal's tagger, written in about 200 lines of Python, achieved near 97% accuracy, which is competitive with state of the art taggers.

6.3 TextBlob

TextBlob is an NLP API that is compatible with NLTK and in part is built on top of it. TextBlob sits somewhere between the education-grade code of NLTK and the industrial-grade code of spaCy. TextBlob does a lot of useful NLP tasks quickly and accurately. To install TextBlob, use pip/pip3, then install corpora as follows:

```
$pip3 install  textblob
$python3 -m textblob.download_corpora
```

6.3.1 The Blob

In order to use TextBlob methods, first convert a string to a TextBlob object as follows, assuming that variable 'text' refers to a raw text string:

Code 6.3.1 — Create a TextBlob object. From a raw string

```
from textblob import TextBlob
# convert raw text to a TextBlob object
blob = TextBlob(text)
```

[2] http://www.aclweb.org/anthology/A92-1021
[3] `https://explosion.ai/blog/part-of-speech-pos-tagger-in-python`

6.3.2 TextBlob Methods

Once a TextBlob object is created, the methods in the API can be applied. The following code chunk shows how to get POS tags. Notice that the tags are already in the TextBlob object, there is no need to run tokenized text through another method.

Code 6.3.2 — POS tagging. on a TextBlob object

```
# get pos tags
tagged = [x for x in blob.tags]
tagged[:4]  # print just the first 4 tags

[('TextBlob', 'NNP'),
 ('is', 'VBZ'),
 ('a', 'DT'),
 ('Python', 'NNP')]]
```

Notice how straightforward the syntax is. In NLTK, text must be tokenized before POS tagging. In TextBlob, the POS tagging is applied directly to the TextBlob object because it has already been tokenized in the process of creating the TextBlob object as we can see:

Code 6.3.3 — Tokenizing. Sentence and Word Tokenizing

```
blob.sentences[0]

Sentence("TextBlob is a Python (2 and 3) library for processing
textual data.")


t_words = [w for w in blob.words if w.lower().startswith('t')]
t_words

['TextBlob', 'textual', 'tasks', 'tagging', 'translation']
```

Code 6.3.4 — Noun Phrases. Extracted from TextBlob object

```
# extract noun phrases
blob.noun_phrases

WordList(['textblob', 'python', 'processing textual data', 'api',
     'common natural language processing',  'nlp',
     'noun phrase extraction', 'sentiment analysis'])
```

In summary, TextBlob is an alternative to NLTK, with simpler syntax for common NLP tasks, and may be faster for some functions. The TextBlob notebook in the GitHub has many more examples.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

6.4 Summary

An ACL Wiki page `https://aclweb.org/aclwiki/POS_Tagging_(State_of_the_art)` compares state of the art tagging systems. The Stanford tagger for example achieved an accuracy of 97.2%. We will explore the Stanford Parser in Chapter 10.

This chapter showed how to do POS tagging with NLTK and TextBlob. Notebooks in the GitHub show how to do POS tagging with NLTK, TextBlob, as well as spaCy.

6.4.1 Quick Reference

Reference 6.4.1 POS Tagging with NLTK

```
import nltk

tokens = word_tokenize(text)
tags = nltk.pos_tag(tokens)
```

Reference 6.4.2 POS Tagging with TextBlob

```
from textblob import TextBlob

# convert raw text to a TextBlob object
blob = TextBlob(text)

# get POS
tagged = [x for x in blob.tags]
```

Reference 6.4.3 POS Tagging with spaCy

```
import spacy

# load a model
nlp = spacy.load('en_core_web_sm')

# create a spacy object
doc = nlp(text)

# extract pos
# the tag is more detailed than pos
for token in doc:
    print(token, token.pos_, token.tag_)
```

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

6.4.2 Practice

Exercise 6.1 Compare POS taggers. Create a sample text of at least 3 sentences.

1. Preprocess the text with whichever methods you decide
2. Output POS tags using NLTK
3. Output POS tags using TextBlob
4. Output POS tags using spaCy
5. Compare the three approaches to POS tagging in terms of ease of use, and accuracy of results

■

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Tag	**Meaning**	**Example**
CC	coordinating conjunction	but
CD	cardinal number	two
DT	determiner	the
EX	existential	there
FW	foreign word	ciao
IN	preposition	on
JJ	adjective	big
JJR	comparative adjective	bigger
JJS	superlative adjective	biggest
LS	list marker	A.
MD	modal	may
NN	noun	car
NNS	plural noun	cars
NNP	proper noun	Mary
NNPS	plural proper noun	Marys
PDT	predeterminer	*both* Marys
POS	possessive	Mary*'s*
PRP	personal pronoun	she
PRP$	possessive pronoun	hers
RB	adverb	badly
RBR	comparative adverb	worse
RBS	superlative adverb	worst
RP	particle	give *up*
SYM	symbol	$
TO	infinitive to	*to* be
UH	interjection	ugh
VB	lexical verb	run
VBD	past tense verb	ran
VBG	gerund or present participle	running
VBN	past particple	ran
VBP	singular present, not 3rd person	run
VBZ	singular present, 3rd person	runs
WDT	wh-determiner	which
WP	wh-pronoun	who
WP$	possessive wh-pronoun	whose
WRB	wh-adverb	when

Table 6.1: Part of Speech Tags

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

7. Relationships between Words

7.1 WordNet

WordNet is a lexical database of nouns, verbs, adjectives, and adverbs that provides short definitions called glosses, and use examples. WordNet groups words into synonym sets called *synsets*. WordNet started as a project at Princeton University, organized by George Miller. The original goal of the project was to support theories of human semantic memory that suggested that people organize concepts mentally in some kind of hierarchy.

NLTK provides a WordNet interface. The interactive console below shows that first wordnet is imported. The second line retrieves all synsets related to the word ‘exercise’. Notice that exercise can be a noun or a verb, and that each noun or verb meaning has a unique identifier, such as ‘exercise.v.04’ for the 4th meaning. Each synset identifier is of this form: `word.pos.num`. Recall that lemmas are the dictionary form of a word, so lemmas are just entries in the WordNet lexicon.

```
# find the synsets of 'exercise'

>>> from nltk.corpus import wordnet as wn
>>> wn.synsets('exercise')
[Synset('exercise.n.01'), Synset('use.n.01'), Synset('exercise.n.03'),
Synset('exercise.n.04'), Synset('exercise.n.05'), Synset('exert.v.01'),
Synset('practice.v.01'), Synset('exercise.v.03'), Synset('exercise.v.04'),
Synset('drill.v.03')]
```

There are several methods that can be applied to a synset, including:

- `definition()` - retrieves the gloss
- `examples()` - gives usage cases
- `lemmas()` - returns a list of WordNet entries that are synonyms

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

```
# definition() method
>>> wn.synset('exercise.n.01').definition()
'the activity of exerting your muscles in various ways to keep fit'
>>> wn.synset('exercise.v.01').definition()
'put to use'

# examples() method
>>> wn.synset('exercise.v.01').examples()
["exert one's power or influence"]

# lemmas() method
>>> wn.synset('exercise.v.01').lemmas()
[Lemma('exert.v.01.exert'), Lemma('exert.v.01.exercise')]
```

7.1.1 Morphy

We can get root forms of words with morphy. Morphy is a rule-based system that can work with different parts of speech as shown below.

```
# get the adjective lemma of 'friendlier'
>>> wn.morphy('friendlier', wn.ADJ)
'friendly'

# get the verb lemma of 'exercised'
>>> wn.morphy('exercised', wn.VERB)
'exercise'

# get the noun lemma of 'exercises'
>>> wn.morphy('exercises', wn.NOUN)
'exercise'
>>>
```

7.1.2 Antonyms

WordNet provides antonyms for lemmas in a synset, if available. The following console interaction requests the first synset for ‘friendly’, specifying an adjective. Since ‘friendly’ is always an adjective, that argument wasn’t really necessary but is included as an example. In the NLTK implementation, we can’t get antonyms from a synset, we have to get it from the lemma. This is why we extract the first lemma and then apply the `antonym()` method.

```
>>> friendly = wn.synsets('friendly', pos=wn.ADJ)[0]
>>> friendly
Synset('friendly.a.01')
>>> friendly.lemmas()
[Lemma('friendly.a.01.friendly')]
>>> friendly.lemmas()[0].antonyms()
[Lemma('unfriendly.a.02.unfriendly')]
```

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

7.2 Synset Relations

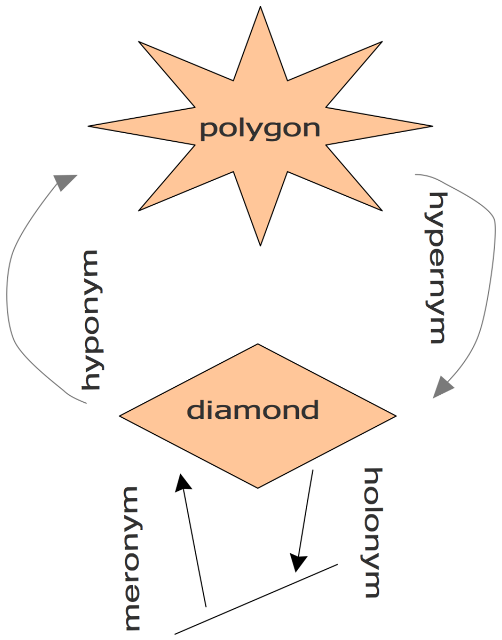


Figure 7.1: Noun Synset Relations

WordNet synsets are connected to other synsets via semantic relations that are hierarchical. These hierarchical relations include:

- hypernym (higher) – canine is a hypernym of dog
- hyponym (lower) – a dog is a hyponym of canine
- meronym (part of) – wheel is a meronym of car
- holonym (whole) – car is a holonym of wheel
- troponym – (more specific action) – whisper is a troponym of talk

Nouns are the most highly connected synsets with several semantic connections in addition to synonyms. In Figure 7.1 we see that diamond and polygon are in a hyponym-hypernym relation. Since *hypo* means under and *hyper* means over, this describes their position in the hierarchy. A diamond is a hyponym of polygon. The term polygon is more general than diamond. An edge and diamond are in a meronym-holonym relation. The edge is a part of the diamond.

Verbs can also be in a hypernym-hyponym relation as we see below. The synset 'exercise.v.03' had one hypernym and one hyponym synset. Not all relations in WordNet are fully implemented in NLTK.

```
>>> wn.synset('exercise.v.03').definition()
'give a workout to'
>>> wn.synset('exercise.v.03').hypernyms()
[Synset('work.v.12')]
>>> wn.synset('exercise.v.03').hyponyms()
[Synset('warm_up.v.04')]
```

Within one synset, one meaning of a word, there can be many lemmas that are synonyms for one another, as demonstrated in the next code block.

Code 7.2.1 — NLTK WordNet. Exploring Synsets and Lemmas

```
from nltk.corpus import wordnet as wn
exercise_synsets = wn.synsets('exercise', pos=wn.VERB)
for sense in exercise_synsets:
    lemmas = [l.name() for l in sense.lemmas()]
    print("Synset: " + sense.name() + "(" +sense.definition() +
        ")  \n\t Lemmas:" + str(lemmas))
```

```
Synset: exert.v.01(put to use)
 Lemmas:['exert', 'exercise']
Synset: practice.v.01(carry out or practice; as of jobs and professions)
 Lemmas:['practice', 'practise', 'exercise', 'do']
Synset: exercise.v.03(give a workout to)
 Lemmas:['exercise', 'work', 'work_out']
Synset: exercise.v.04(do physical exercise)
 Lemmas:['exercise', 'work_out']
Synset: drill.v.03(learn by repetition)
 Lemmas:['drill', 'exercise', 'practice', 'practise']
```

7.2.1 Traversing the hierarchy

The code below demonstrates traversing the hierarchy up to the top-most noun level of 'entity'.

Code 7.2.2 — NLTK WordNet. Traversing the hierarchy

```
hyp = dog.hypernyms()[0]
top = wn.synset('entity.n.01')
while hyp:
    print(hyp)
    if hyp == top:
        break
    if hyp.hypernyms():
        hyp = hyp.hypernyms()[0]
```

```
Synset('canine.n.02')
Synset('carnivore.n.01')
Synset('placental.n.01')
Synset('mammal.n.01')
Synset('vertebrate.n.01')
Synset('chordate.n.01')
Synset('animal.n.01')
Synset('organism.n.01')
Synset('living_thing.n.01')
Synset('whole.n.02')
Synset('object.n.01')
Synset('physical_entity.n.01')
Synset('entity.n.01')
```

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Unlike nouns, for verbs there is no uniform top level synset. Tracing the synset for the verb ‘dog’ gives ‘pursue’ as the hypernym, which makes sense. The hypernym of ‘pursue’ is ’travel’, so this is a bit of semantic drift. The online notebook corresponding to this section shows the full code.

7.2.2 Word Similarity

A similarity score between two word senses can be extracted from WordNet, where the similarity ranges from 0 (little similarity) to 1 (identity).

```
dog = wn.synset('dog.n.01')
cat = wn.synset('cat.n.01')
print(dog.path_similarity(cat))
#  0.2

hit = wn.synset('hit.v.01')
slap = wn.synset('slap.v.01')
print(wn.path_similarity(hit, slap))
#  0.14285714285714285
```

Another similarity metric, Wu-Palmer, looks at common ancestor words.

```
wn.wup_similarity(dog, cat)
# 0.8571428571428571

wn.wup_similarity(hit, slap)
# 0.25
```

7.2.3 Word Sense Disambiguation

The term *homonyms* refers to words that share the same form but have unrelated meanings. If two words share the same spelling they are called *homographs* and if they share the same sound they are called *homophones*. Examples are:

- bank: a financial institution, or sloping land near a body of water
- bat: a piece of baseball equipment, or a flying nocturnal mammal
- club: an instrument for striking, or a social organization

A *polysemous* word has many related meanings. Again, ‘bank’ is an example because it can refer to the financial institution or the building:

```
The bank was constructed in 1801.
That bank charges high fees.
```

Yet another term is *metonymy*, which refers to using one aspect of a concept to refer to other aspects. For example, the ‘White House’ may refer to the building or the current administration.

Consider these two sentences:

```
Joe went fishing down by the river bank.
Joe took his money to the bank to make a deposit.
```

The meaning of ‘bank’ is obvious to humans, but how is an NLP system to determine which meaning of ‘bank’ is intended in each sentence? This NLP task is called *word sense disambiguation*, or WSD for short. One fairly simple approach to WSD is to look at the words around the ambiguous word. The Lesk algorithm looks at context words and compares them to the words in dictionary glosses. The dictionary entry that shares the most overlap words is chosen. The Lesk algorithm is implemented in NLTK, and is demonstrated in the next code block.

Code 7.2.3 — **NLTK WSD.** The Lesk algorithm.

```
from nltk.wsd import lesk

sent = ['I', 'went', 'to', 'the', 'bank', 'to', 'deposit', 'money', '.']
print(lesk(sent, 'bank', 'n'))

# output:
Synset('savings_bank.n.02')
```

Current state-of-the-art WSD approaches use machine-learning approaches that outperform lexical approaches such as the Lesk algorithm.

7.3 SentiWordNet

SentiWordNet is a lexical resource built on top of WordNet that assigns 3 sentiment scores for each synset: positivity, negativity, and objectivity. You can import SentiWordNet in NLTK.

The SentiWordNet corpus needs to be downloaded once, using the Python console:

```
>>>import nltk
>>>nltk.download('sentiwordnet')
```

Code 7.3.1 — **SentiWordNet.** Get scores for a synset.

```
from nltk.corpus import sentiwordnet as swn

breakdown = swn.senti_synset('breakdown.n.03')
print(breakdown)
print("Positive score = ", breakdown.pos_score())
print("Negative score = ", breakdown.neg_score())
print("Objective score = ", breakdown.obj_score())

# output
<breakdown.n.03: PosScore=0.0 NegScore=0.25>
Positive score =  0.0
Negative score =  0.25
Objective score =  0.75
```

A simple approach to performing sentiment analysis on a document is to iterate over the tokens and count how many are positive and how many are negative. This naive approach is demonstrated below.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Code 7.3.2 — SentiWordNet. A Naive Sentiment Analysis.

```
sent = 'that was the worst movie ever'
neg = 0
pos = 0
tokens = sent.split()
for token in tokens:
    syn_list = list(swn.senti_synsets(token))
    if syn_list:
        syn = syn_list[0]
        neg += syn.neg_score()
        pos += syn.pos_score()

print("neg\tpos counts")
print(neg, '\t', pos)
# output
neg pos counts
1.0   0.0
```

7.4 VADER

VADER (Valence Aware Dictionary and sEntiment Reasoner) is a rules-based NLP tool designed for sentiment analysis in social media, created by Hutto and Gilbert. To install use pip or pip3:

```
pip3 install vaderSentiment
```

The code block below shows how to import and use VADER on a sample data set that is in the VADER GitHub repo.

Code 7.4.1 — VADER. Usage Example

```
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer

analyzer = SentimentIntensityAnalyzer()

with open('VADER_data/movieReviewSnippets_GroundTruth.txt', 'r') as f:
    lines = f.read().splitlines()

for line in lines[:5]:
    i, sentiment, text = line.split('\t')
    vs = analyzer.polarity_scores(text)
    print('\n', text, '\n\t VS=', str(vs))
```

```
The Rock is destined to be the 21st Century's new ''Conan''
    and that he's going to make a splash even greater than Arnold
     Schwarzenegger, Jean Claud Van Damme or Steven Segal.
VS= {'neg': 0.0, 'neu': 0.923, 'pos': 0.077, 'compound': 0.3612}
```

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

```
The gorgeously elaborate continuation of ''The Lord of the Rings''
    trilogy is so huge that a column of words cannot adequately
    describe co writer/director Peter Jackson's expanded vision of J.R.R.
    Tolkien's Middle earth.
VS= {'neg': 0.0, 'neu': 0.783, 'pos': 0.217, 'compound': 0.8069}

Effective but too tepid biopic
VS= {'neg': 0.0, 'neu': 0.661, 'pos': 0.339, 'compound': 0.2617}

If you sometimes like to go to the movies to have fun, Wasabi is
     a good place to start.
VS= {'neg': 0.0, 'neu': 0.648, 'pos': 0.352, 'compound': 0.8271}

Emerges as something rare, an issue movie that's so honest and
    keenly observed that it doesn't feel like one.
VS= {'neg': 0.086, 'neu': 0.65, 'pos': 0.264, 'compound': 0.6592}
```

7.5 Collocations

Up to this point we have been discussing words in isolation. When two or more words usually occur together with a frequency greater than chance would suggest, the words may form a *collocation*.

A key indication that two words form a collocation is that you cannot substitute synonyms. For example, 'wild rice' does not mean the same as 'unruly rice'. The sample texts in NLTK work with the `collocations()` method because they are special NLTK Text objects, as shown below. If you want to find collocations on your own text, you have to convert it to an NLTK Text object first.

```
>>> text4.collocations()
United States; fellow citizens; four years; years ago; Federal
Government; General Government; American people; Vice President; Old
World; Almighty God; Fellow citizens; Chief Magistrate; Chief Justice;
God bless; every citizen; Indian tribes; public debt; one another;
foreign nations; political parties
```

Collocations are important in a variety of NLP applications. For example, in a machine translation system, the collocation 'strong tea' cannot be replaced by 'powerful tea'. Collocation algorithms first extract bigrams (sets of words that occur next to each other), then search for bigrams that occur together more than chance would indicate. As we can see in the example above, some of the collocations are just coincidence. For example, 'every citizen' is not really a collocation because it could be replaced with 'all citizens' or 'each citizen', depending on the context.

One way to find collocations in text is to use point-wise mutual information, pmi, using the formula below. Take the probability of x and y occurring next to each other, and divide this by the probability of x multiplied by the probability of y. The probability of a word is the frequency of that word divided by the total number of words.

$$log_2 \frac{P(x,y)}{P(x) * P(y)} \tag{7.1}$$

The pmi value can be positive or negative. A pmi value of 0 means that x and y are independent, so x+y (x followed by y) does not form a collocation. If the pmi is positive, it means that x and y occur together more than expected by chance, and is likely a collocation. Similarly, a negative pmi gives evidence that x+y is not a collocation. The range of values for pmi is from negative infinity to the minimum of (-log p(x), -log p(y), which is generally a 2-digit integer or less.

The NLTK object `text4` contains 149,797 tokens, and 149,796 bigrams. The phrase 'fellow citizens' occurred 61 times, 'fellow' occurred 128 times and 'citizen' occurred 240 times. Plugging these values into the formula gives a pmi score of 8.2.

$$log_2 \frac{P(x,y)}{P(x) * P(y)} = log_2 \frac{61/149796}{128/149797 * 240/149797} = 8.2 \tag{7.2}$$

Now consider the phrase 'the citizens', which occurs 11 times, while 'the' occurs 9446 times and 'citizens' occurs 240 times. This phrase has a pmi score of -0.46.

$$log_2 \frac{P(x,y)}{P(x) * P(y)} = log_2 \frac{11/149796}{9446/149797 * 240/149797} = -0.46 \tag{7.3}$$

7.6 Summary

We have seen that we can learn a lot about the individual words in text using resources like NLTK and WordNet. WordNet organizes common nouns, verbs, adjectives, and adverbs in a hierarchy. Each word meaning is represented as a synset, a synonym set of related words. Word relations can be analyzed in terms of hypernyms, hyponyms, meronyms, holonyms, and troponyms. Various word similarity metrics are included in NLTK.

SentiWordNet is an extension of WordNet that includes three sentiment scores for each synset: positivity, negativity, and objectivity. The VADER algorithm is another useful library for sentiment analysis.

7.6.1 Quick Reference

Reference 7.6.1 Retrieve synsets

```
from nltk.corpus import wordnet as wn

wn.synsets('...')  # your word here
```

Reference 7.6.2 Traverse the NLTK Hierarchy

```
# see code in the GitHub

# traverse up from 'dog' synset
dog = wn.synset('dog.n.01')
hyper = lambda s: s.hypernyms()
list(dog.closure(hyper))
```

See the GitHub for examples on finding word similarity, using SentiWordNet, and VADER.

7.6.2 Practice

Download this data set from UCI: `https://archive.ics.uci.edu/ml/datasets/Sentiment+Labelled+Sentences#`

Subset the data to just the first 100 sentences.

Exercise 7.1 In a Python program or notebook, classify the first 100 sentences and calculate accuracy:

1. Read in the data, separating into a list of sentences and a corresponding list of labels
2. Create a POS-tagged list of sentences from the input sentences by summing the scores of individual words. Make sure you ask for the correct type ‘a’, ‘r’, ’v’, ’n’, depending on the POS.
3. Classify a sentence as positive (0) if the positive score is >= the negative score; otherwise, classify it as negative (1).
4. Calculate an accuracy score as the number of correctly classified sentences divided by the total number of examples.
5. Give examples of misclassified sentences and provide analysis of why you think the sentence was not classified correctly.

■

7.6.3 Going Further

- Read more about WordNet on the official site: `https://wordnet.princeton.edu/`
- SentiWordNet’s GitHub: `https://github.com/aesuli/SentiWordNet`
- VADER’s GitHub: `https://github.com/cjhutto/vaderSentiment`

8. N-gram Models

Up to this point we have been analyzing text mainly by word counts, by part of speech, and by word co-occurrences in collocations. The next logical step is learning about text from sequences of words. Mentally fill in the blank in the following sentence:

```
Before the movie begins, please turn off your _____________.
```

Chances are, you thought 'phones'. One of the ways that people learn is by pattern recognition. Hearing 'turn off your phone' many times in different contexts (movies, meetings, airplane landings) reinforces this pattern. Context is important. In a movie theater, 'turn off your phones' makes sense, but on a web site, a more common pattern might be 'turn off your ad blocker.' If you type 'please turn off your' into google, phones is one of the top suggestions. NLP applications can learn patterns as well, but how?

8.1 N-grams

An n-gram is a sliding window over text, n words at a time. Unigrams take one word at a time, bigrams take two words at a time, trigrams take 3 words at a time. Above 3, they are usually just called n-grams with n specified as 4, 5, etc. N-grams can be used to create a probabilistic model of language. In order to learn such a language model, a corpus (body of text) is needed. The choice of corpus will greatly influence the language model. A model learned from Shakespeare will be quite different than a model learned from a science textbook. Consider the following toy corpus, a lower-case version of a classic nursery rhyme:

```
jack be nimble.
jack be quick.
jack jump over the candlestick.
```

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

The next code block uses Python to get a list of unigrams and a list of bigrams for this corpus.

Code 8.1.1 — N-gram Model. Using Python

```
raw_text = "jack be nimble. jack be quick. jack jump over \
       the candlestick."

from nltk import word_tokenize
unigrams = word_tokenize(raw_text)

bigrams = [(unigrams[k], unigrams[k+1]) for k in range(len(unigrams)-1)]
```

The unigrams are:

```
['jack',
 'be',
 'nimble',
 '.',
 'jack',
 'be',
 'quick',
 '.',
 'jack',
 'jump',
 'over',
 'the',
 'candlestick']
```

And the bigrams are:

```
[('jack', 'be'),
 ('be', 'nimble'),
 ('nimble', '.'),
 ('.', 'jack'),
 ('jack', 'be'),
 ('be', 'quick'),
 ('quick', '.'),
 ('.', 'jack'),
 ('jack', 'jump'),
 ('jump', 'over'),
 ('over', 'the'),
 ('the', 'candlestick')]
```

A common practice in building n-gram models is to add start and stop symbols to the beginning and end of each sentence. We will not add start symbols in this toy example, but let the period serve as an end symbol.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

N-grams are probabilistic models. What is the probability of ‘jack be’ occurring in this corpus? It is the probability: P(jack, be) = P(jack)P(be | jack), or more generally:

$$P(w_1, w_2) = P(w_1)P(w_2|w_1) \tag{8.1}$$

In this example, P(jack) = number of unigrams that are ‘jack’ divided by the number of unigrams, 3/13. The P(be | jack) is the count of ‘jack be’ bigrams divided by the count of ‘jack’, 2/3. So the probability of ‘jack be’ is 3/13 * 2/3 = 15%.

What is the probability of ‘jack jump’? P(jack, jump) = P(jack)P(jump | jack) = 3/13 * 1/3 = 7.7%. So we are about twice as likely to see ‘jack be’ than ‘jack jump’. This is shown in equation form below, where C(*) is for count.

$$P(w_1, w_2) = P(w_1)P(w_2|w_1) = \frac{C(w_1)}{\Sigma_n C(w_n)} \frac{C(w_1 w_2)}{C(w_1)} \tag{8.2}$$

What if we want to know the probability of a certain sequence of 4 words? This would be:

$$P(w_1, w_2, w_3, w_4) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)P(w_4|w_1, w_2, w_3) \tag{8.3}$$

This computation could get messy, so a Markov assumption is made, namely that the probability will only look at the previous word. This means:

$$P(w_1, w_2, w_3, w_4) = P(w_1)P(w_2|w_1)P(w_3|w_2)P(w_4|w_3) \tag{8.4}$$

which we can generalize to:

$$\Pi_{k=1}^{n} P(w_k|w_{k-1}) \tag{8.5}$$

To facilitate calculating probabilities in code, dictionaries of counts are created. NLTK has an ngram() function we can use, as shown below.

Code 8.1.2 — Building N-gram Dictionaries. Using Python and NLTK

```
from nltk import word_tokenize
from nltk.util import ngrams

raw_text = "jack be nimble. jack be quick. jack jump over \
the candlestick"

unigrams = word_tokenize(raw_text)
bigrams = list(ngrams(unigrams, 2))

unigram_dict = {t:unigrams.count(t) for t in set(unigrams)}
bigram_dict = {b:bigrams.count(b) for b in set(bigrams)}
```

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

The unigram dictionary entry for ‘jack’ is:

```
'jack': 3
```

The bigram dictionary entry for ‘jack be’ is:

```
('jack', 'be'): 2
```

8.2 Smoothing

What is the probability of ‘jack be smart’? When we try to compute P(smart|be) there will be a problem because ‘smart’ has a count of 0. Multiplying by zero will zero-out the probability which is definitely not the desired result. The problem of zero counts is called the *sparsity problem* because our data cannot possibly contain every sequence of possible words.

One approach to dealing with 0 counts is smoothing. The idea of smoothing is to fill in zero values with a little bit of probability of the overall mass. This makes the distribution smoother. There are dozens of smoothing techniques commonly used, but here we discuss a simple one, Laplace smoothing, sometimes called add-one smoothing. The idea is to add 1 to the 0 count so that it is not zero. We don’t know ahead of time which counts are zero, so 1 is added to all counts. To offset this, the total vocabulary count, V, is added to the denominator to balance it out. Recall the unsmoothed probability of a word:

$$P(w_i) = \frac{C(w_i)}{N} \tag{8.6}$$

where C(*) is a count function and N is the total number of unigrams. With smoothing, this becomes:

$$P(w_i) = \frac{C(w_i) + 1}{N + V} \tag{8.7}$$

Laplace smoothing is easy to implement but does not perform well because it adjusts all the probabilities too aggressively. Another approach is to replace zero counts with counts of words that occur only once. This is a simplified version of the Good-Turing smoothing technique.

$$P(w_0) = \frac{N_1}{N} \tag{8.8}$$

which is saying that the probability of a word we have not seen before is about the same as the probability of a word that occurs only once. Finding that probability involves counting the number of singletons N_1 and dividing by the total word count, N. Another approach is to use 1/N as the probability.

Simple implementations of these smoothing techniques are shown in the next code block. As shown in the notebook in the GitHub, calling this function to calculate the probability of ‘jack be nimble’ with different smoothing techniques is:

```
probability with simplified Good-Turing is 0.33333
probability with laplace smoothing is 0.00909
log prob is -4.70048 == 0.00909
```

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

The online notebook shows that calling the function with ‘jack be smart’ yields a very low probability, near zero. The Laplace smoothing adjusted the probabilities too aggressively resulting in the phrase that was in the data, ‘jack be nimble’ having too low a probability and the phrase with the 0-count token, ‘jack be smart’ having too high a probability compared to the modified Good-Turing smoothing.

Code 8.2.1 — Calculating probabilities with Smoothing. Simplified implementations

```
import math

def compute_prob(text, unigram_dict, bigram_dict, N, V):
# N is the number of tokens
# V is the number of unique tokens, the vocabulary size

    unigrams_test = word_tokenize(text)
    bigrams_test = list(ngrams(unigrams_test, 2))

    p_gt = 1
    p_laplace = 1
    p_log = 0  # sometimes used to prevent overflow

    for bigram in bigrams_test:
        n = bigram_dict[bigram] if bigram in bigram_dict else 0
        n_gt = bigram_dict[bigram] if bigram in bigram_dict else 1/N
        d = unigram_dict[bigram[0]] if bigram[0] in unigram_dict else 0
        if d == 0:
            p_gt = p_gt * (1 / N)
        else:
            p_gt = p_gt * (n_gt / d)
        p_laplace = p_laplace * ((n + 1) / (d + V))
        p_log = p_log + math.log((n + 1) / (d + V))

    print("\nprobability with simplified Good-Turing is %.5f" % (p_gt))
    print("probability with laplace smoothing is %.5f" % p_laplace)
    print("log prob is %.5f == %.5f" % (p_log, math.exp(p_log)))
```

8.3 N-grams for Language Generation

In order to use N-grams counts in NLP applications the N-Grams are converted to probabilities. The probability of a unigram is just the count of that unigram divided by the total number of words:

$$P(w_i) = \frac{C(w_i)}{N} \tag{8.9}$$

The probability of a bigram is the count of that bigram divided by the count of the first word:

$$P(w_i w_{i+1}) = \frac{C(w_i w_{i+1})}{C(w_i)} \tag{8.10}$$

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

The next code block shows how to create probability dictionaries using dictionary comprehensions.

Code 8.3.1 — Create Probability Dictionaries. For Unigrams and Bigrams.

```
u_probs = {t:unigrams.count(t)/len(unigrams) for t in set(unigrams)}
b_probs = {b:bigrams.count(b)/unigrams.count(b[0]) for b in set(bigrams)}
```

The next code block takes a very naive approach to generating language. Given a start word, the function looks through the bigram probabilities to find the bigram with the start word in the first position that has the highest probability. It then concatenates this to the phrase it generates. This continues until the last token added was a period.

Code 8.3.2 — Generate Text. A Naive Approach.

```
def naive_gen(start_word, u_probs, b_probs):
    phrase = [start_word]
    while phrase[-1] != '.':
        candidate_next = {k:b_probs[k] for k in b_probs
                          if k[0] == phrase[-1]}
        candidate_next = sorted(candidate_next.items(),
                         key=lambda x:x[1], reverse=True)
        if not candidate_next:
            break
        phrase += [candidate_next[0][0][1]]  # [0] = first bigram
        print(phrase)

    return phrase
```

It will come as no surprise that a start word of 'jack' returned the phrase 'jack be quick.'

```
naive_gen('jack', u_probs, b_probs)
'jack be quick .'
```

The generator is limited by the small size and simple approach. Generally using bigrams does not get better results. Trigrams works better and higher n-grams can work better still. Let's see if NLTK can do better with this tiny corpus.

Code 8.3.3 — Generate Text. NLTK

```
import nltk
nltkText = nltk.Text(unigrams)
nltkText.generate()
```

The output is:

```
be nimble . . jack jump over the candlestick over the candlestick jack
be quick . jack be quick . . jack be nimble . the candlestick be quick
. over the candlestick the candlestick . be nimble . jack jump over
the candlestick quick . jack be nimble . jack be quick . jack be quick
. . . jack jump over the candlestick jack be nimble . over the
```

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

```
candlestick jack be quick . . nimble . jack jump over the candlestick
jack be nimble . jack be nimble . jump over the candlestick jack be
quick .
```

The NLTK generate() function gets better results from a bigger corpus, as in the Inaugural texts corpus.

```
from nltk.book import *
text4.generate()
```

The generated text is:

```
occur , and especially the truth that democratic government has innate
capacity to govern its affairs aright through the Province ceded , by
any timid forebodings of evil were not to overtake them while I
possess the property of the peaks and the exercise of free and firm on
the farm , in view that the best of my countrymen will ever find me
ready to confer their benefits on countless generations yet to make
its promise for all generations ." , remains essentially unchanged .
cost of the Rocky Mountains . abuses of an ever - expanding American
dream
```

The online notebooks give further examples of generation with larger texts. Creating dictionaries of counts over a large corpus can be time-consuming. A good idea is to write a Python script to create the dictionaries once and pickle (serialize) them. Then, other Python scripts can simply read in the pickle files and unpickle them. Given a dictionary named my_dict, the following code block shows how to pickle and unpickle. Notice that pickle files are binary files and need to be read as 'rb' and written to as 'wb'.

Code 8.3.4 — Pickle. Compress and Save Data

```
import pickle

# pickle a dictionary
with open('my_pickle', 'wb') as handle:
    pickle.dump(my_dict, handle)

# unpack
with open('my_pickle', 'rb') as handle:
    new_dict = pickle.load(handle)
```

8.4 Perplexity

One way of evaluating NLP applications is to do an *extrinsic* evaluation by having human annotators evaluate the result using a predefined metric. Extrinsic evaluations tend to be time-consuming and relatively expensive, so they are used sparingly, often at the end of a project. Another method of evaluation is *intrinsic* evaluation using some metric to compare models. A common metric for language models is perplexity. Typically a small amount of test data is set aside for calculating perplexity. Perplexity will measure how well the language model predicts the text in the test data. The following shows a formula for calculting perplexity, PP:

$$PP(W) = P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \tag{8.11}$$

This is the inverse probability of seeing the words we observe, normalized by the number of words. This is an exponentiation of the entropy, the average number of bits needed to encode the information in a random variable. We want the model to have low perplexity, which corresponds to low entropy, which corresponds to having less chaos in the data. Perplexity can also be viewed as measuring a branching factor, the number of choices we have for the next word given the current word. More specialized text will tend to have lower perplexity than text from a range of sources. NLTK has a perplexity method that can be applied to an n-gram model.

8.5 Summary

N-grams create very useful language models that are part of many NLP systems. Common applications include:

- spelling correction
- machine translation
- speech recognition
- auto suggestion typing/messaging and searching

8.5.1 Quick Reference

Reference 8.5.1 NGrams from NLTK

```
from nltk import word_tokenize
from nltk.util inport ngrams

raw_text = " . . ."

unigrams = word_tokenize(raw_text)
bigrams = list(ngrams(unigrams, 2}
```

Reference 8.5.2 Serialize and unserialize

```
import pickle

# pickle a dictionary
with open('my_pickle', 'wb') as handle:
    pickle.dump(my_dict, handle)

# unpack
with open('my_pickle', 'rb') as handle:
    new_dict = pickle.load(handle)
```

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

8.5.2 Practice

In a Python program or notebook, import one of NLTK's text objects.

Exercise 8.1 Explore bigrams:

1. Build a bigram dictionary from the tokens in the text
2. Print out the 10 most common bigrams
3. Print out the 10 lease common bigrams
4. Generate text using the bigrams
5. Analyze the results in terms of reasonableness or strangeness compared to natural text

■

8.5.3 Going Further

- Ngrams are available from COCA (Corpus of Contemporary American English): `https://www.ngrams.info/`
- Play with Google's Ngram viewer: `https://books.google.com/ngrams`
- Jurafsky and Martin's Book (`https://web.stanford.edu/~jurafsky/slp3/`) contains a detailed look at ngrams along with explanations of several smoothing methods
- ACL has a list of software solutions for language models: `https://aclweb.org/aclwiki/Language_modeling_software`

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

III

Part Three: Sentences

Part Three

In Part Three we examine how words are combined into sentences. The structure of a sentence supports the intended meaning. Linguists use the term *generative grammar* to indicate the rules in our heads that provide the framework for sentences, so that we don't just spout word salads. As we will see, these rules are hierarchical and recursive, allowing infinite expression.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

9. CFG Grammar

A **formal grammar** is a set of rules for rewriting strings recursively to form a correct structure. Formal languages are small and precise compared to the vast unruly wilderness of natural human language. Natural human languages have grammar, a structure, that is referred to as **syntax**. Natural language syntax is difficult to pin down in an exhaustive set of rules because natural language is complex and changes over time. Further, natural language has formal and informal versions, as well as regional variations. The complexity of natural language is demonstrated in *garden path sentences*, sentences that take you through a maze of possible parses until you understand what is meant. Garden path sentences are grammatically correct under at least one interpretation. The following is a classic example of a garden path sentence:

```
The horse raced past the barn fell.
```

This sentence is confusing because the first and most obvious mental parse is to assume that 'raced' is the verb, but the verb in the sentence is 'fell'. Here, 'raced past the barn' is a phrase modifying 'horse.'

Formal languages are more constrained, and therefore better behaved. Formal languages arose independently in the study of natural languages by linguists, and in the development of programming languages by computer scientists.

9.1 Context-Free Grammar

A Context-Free Grammar, CFG, is a set of production rules, a start symbol, and a set of terminal symbols. A CFG defines the form of the formal language, it has nothing to say about meaning in any context. CFGs can be used in one of two ways:

- To generate valid statements in the formal language
- To parse a statement to check if it is valid

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

The CFG concept is attributed equally to the work of linguist Norm Chomsky in the late 1950s, and to the work of John Backus and Peter Naur in developing the Algol programming language. Database students are familiar with BNF notation in query syntax.

9.1.1 An Example

Here is an example of a CFG:

```
'S'   ->  [['NP', 'VP']],
'NP'  ->  [['DT', 'NN'], ['DT', 'JJ', 'NN']],
'VP'  ->  [['VB'], ['VB', 'PP']],
'PP'  ->  [['P', 'DT', 'NN']],
'JJ'  ->  ['happy', 'sad', 'silly'],
'DT'  ->  ['a', 'the'],
'NN'  ->  ['cat', 'dog', 'clown'],
'VB'  ->  ['plays', 'runs'],
'P'   ->  ['with', 'near', 'beside']
```

The jupyter notebook 'CFG_1' in the github represents these rules as a dictionary, where the key is the production rule name, and the values are represented by a list of terminals and non-terminals.

The start symbol is 'S' for 'start' or 'sentence'. Production of a sentence starts with 'S'. The first production rule shows that a sentence can be parsed into a noun phrase, NP, followed by a verb phrase, VP. The second production rule shows that a NP can be parsed into: `DT NN`, that is, a determiner followed by a noun, or a NP can be parsed into `DT JJ NN`, a determiner, an adjective, a noun. Similarly, the third rule shows two kinds of verb phrases: One with just a verb, and another with a verb followed by a prepositional phrase. There is only one way to parse a prepositional phrase in this grammar.

The last four production rules have a word category, such as an adjective on the left and the *terminal* symbols on the right. There are two determiners, three nouns, two verbs, and three prepositions. Quite a limited language, but complex enough to generate many sentences.

9.1.2 Generating Sentences

The online notebook shows the full code for generating sentences from this small set of rules. An interative function is written as well as a recursive one. Here is the recursive function.

Code 9.1.1 — CFG. Function to generate sentences.

```
# expand function - recursive version
def expand(expanded, expand_me, rules):
    if isinstance(expand_me, list):
        for token in expand_me:
            r = random.randint(0, len(rules[token])-1)
            replacement = rules[token][r]
            expand(expanded, replacement, rules)
    else:  # append the terminal
        expanded.append(expand_me)

    return expanded
```

The function is called with an empty list for 'expanded', and a single start token 'S' in the list 'expand_me'. The keys of the dictionary are the rule names (LHS). What is retrieved may be a list of terminal tokens or a list of non-terminals. To make things more interesting, a random selection is made when more than one option is available. In the recursive call to expand(), the randomly retrieved rhs 'replacement' is the new 'expand_me' for the recursive call.

```
The function output for 5 sentences is shown below:
# generate 5 sentences
for i in range(5):
    new_S = expand([], ['S'], rules)
    print(new_S)

# output
['a', 'silly', 'clown', 'runs']
['a', 'clown', 'runs']
['the', 'sad', 'clown', 'runs']
['the', 'dog', 'runs', 'with', 'the', 'cat']
['the', 'sad', 'cat', 'plays']
```

To keep things simple, the terminals in this grammar were chosen so that the nouns all make sense with the verbs, and the determiners work with any of the nouns. Such a simple grammar was able to produce many sentences. Such a simple generator could even serve as the engine of an interactive reading tutor with small children.

How many sentences could be generated with this grammar? There is only one sentence form: NP VP, but there are two NP forms and two VP forms. These combine to form 1 x 2 x 2 = 4 sentence patterns. Filling in the possible terminals gives us about 100 possible sentences just from these simple production rules.

New production rules can easily be added to the grammar, as shown in the online notebook:

```
rules['VP'].append(['CP', 'JJ'])
rules['CP'] = ['is']
```

The new rules account for copular sentences where a linking verb like 'be' connects the subject and a description adjective, as in the sentence *She is smart.* Using the terminals (also called the vocabulary or *lexicon*), the following sentences are now possible.

```
['the', 'cat', 'is', 'silly']
['the', 'silly', 'clown', 'is', 'sad']
```

9.2 Constituency

An important concept in syntax is *constituency*, the observation that groups of words combine to form a structure that acts as a single unit. For example noun phrases (NP), verb phrases (VP), and prepositional phrases (PP) have constituency. Sample phrases are underlined below. The first sentence has a noun phrase, 'the book', and a prepositional phrase, 'on the table'. The second sentence has two noun phrases, one for the subject, and one for the direct object of the verb 'chased.' The third sentence shows that the verb phrase in the sentence is a constituent. Constituent structure is hierarchical. Constituents can contain other constituents.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

- The book is on the table.
- The carefree girl chased the dancing butterfly.
- She danced her cares away.

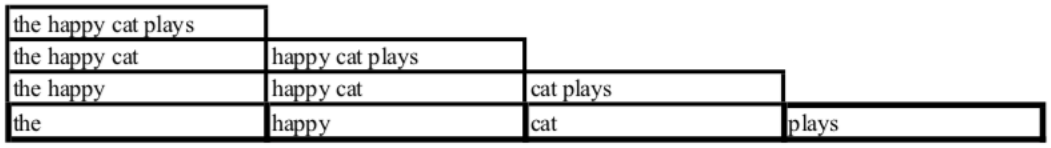
Notice that the production rules of the simple grammar above capture this notion of constituency by defining valid forms for NP, VP and PP.

A phrase has a *head word* which will be a noun for a NP, a verb for a VP, and the preposition for a PP. The head words are not identified in the production rules because they are not significant. However, in examining different kinds of sentence parses in the next chapter, the head word becomes important.

9.3 Parsing

A code block earlier in the chapter showed how production rules can be used to generate valid sentences. The same rules can be used to check if an input sentence is valid. The online notebook 'CFG2' gives an example of exploring this manually with some Python code. The notebook code provides a simplified introduction to a bottom-up approach to parsing called the CYK algorithm. A standard CYK implementation uses dynamic programming to explore every possible parse given the input sentence and grammar. Here, we will look at a simple example using a spreadsheet and verbal explanations.

The big picture is shown below. The sentence to be parsed is *the happy cat plays*. Since grammars are hierarchical by nature, the parse takes a hierarchical approach as well. This bottom-up approach starts at the bottom with each token considered independently. Moving up a level, words are considered, in order, two at a time. Moving up another level, words are considered three at a time. Finally at the top level, all four words of the sentence are together.



the happy cat plays			
the happy cat	happy cat plays		
the happy	happy cat	cat plays	
the	happy	cat	plays

Figure 9.1: Hierarchical Explorations of Tokens

9.3.1 Parsing: Round One

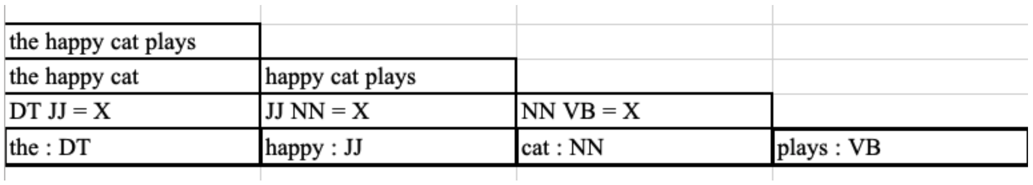
The first step is to take the individual tokens and assign a part of speech. This is done by looking for the word on the right-hand side (RHS) of a production rule, and classifying it according to the rule on the left-hand side (LHS). For the sample sentence and grammar, this results in :

```
[['DT'], ['JJ'], ['NN'], ['VB']]
```

Each part of speech (POS) is in a list. For this simple grammar there happens to be only one possible POS for each token. However, that is usually not the case. Consider the word 'sense', which could be a VB as in *I sense danger* or a NN as in *My spidey sense is tingling*. The word 'sense' would have two possible pos that would have to be considered: `[['VB'], ['NP']]`

9.3.2 Parsing: Round Two

Now the POS list is considered two at a time. There is no RHS pattern for 'DT JJ' or 'JJ NN' or 'NN VB', so all of these are Xd out and we move up to the next level.

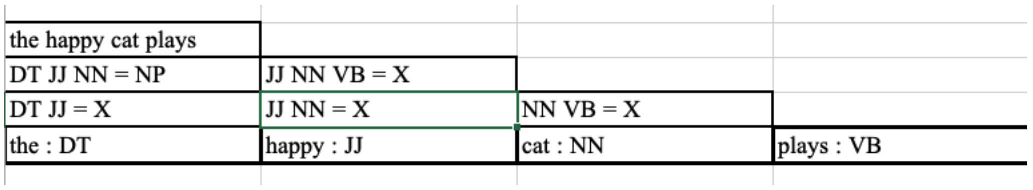


the happy cat plays			
the happy cat	happy cat plays		
DT JJ = X	JJ NN = X	NN VB = X	
the : DT	happy : JJ	cat : NN	plays : VB

Figure 9.2: Search for Patterns

9.3.3 Parsing: Round Three

Next, tokens are examined 3 at a time. The pattern ‘DT JJ NN’ is in the dictionary, it is a NP. We can replace ‘DT JJ NN’ with ‘NP’. The pattern ‘JJ NN VB’ is not a legal pattern, so it is Xd out.



the happy cat plays			
DT JJ NN = NP	JJ NN VB = X		
DT JJ = X	JJ NN = X	NN VB = X	
the : DT	happy : JJ	cat : NN	plays : VB

Figure 9.3: Tokens in Windows of Three

9.3.4 Parsing: FInal Round

The sentence so far has been parsed as ‘NP VB’. There is not ‘NP VB’ rule but ‘VB’ can be rewritten as ‘VP’. Now we have ‘NP VP’ which matches pattern ‘S’. Since the parse successfully reached ‘S’, this sentence is a valid sentence in this grammar.

9.4 Bracket Notation and Trees

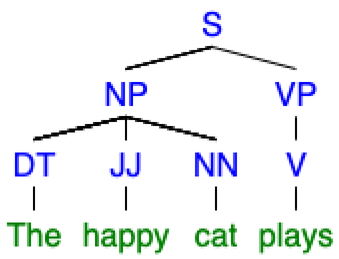


Figure 9.4: Parse Tree

Parses are conventionally displayed in a tree structure or bracket notation. Several nice interactive syntax tree generators are available online. Here’s a nice one: `http://mshang.ca/syntree/`

The bracket notation and the tree visualization give you the same information, but in different forms. The hierarchy is easier to see in the tree visualization.

The bracket notation groups each token and its POS in brackets. Then these are grouped hierarchically into larger and larger units until S:

```
[S [NP [DT The] [JJ happy] [NN cat]] [VP [V plays]]]
```

9.5 PCFG

Algorithms that parse CFGs have to do an exhaustive search of every possible parse because with some sentences, more than one parse is possible. A natural evolution of CFGs is to consider the most likely choice at each point in the parse. This is as simple as assigning a probability to each production rule.

As parsers continued to improve, probabilistic approaches dominated. Currently, the top syntax parsers use neural networks that were trained on huge amounts of sentences.

9.6 Treebanks

How can a PCFG know what probabilities to assign? Probabilities can be learned from a Treebank. A Treebank is a lexical resources that is a collection of sentences that have been parsed and annotated by experts. The Penn Treebank includes annotated sentences from multiple corpora such as the Brown Corpus, The Wall Street Journal, and more. Treebanks exist for other languages besides English, notably Arabic and Chinese.

Grammar rules can be extracted from a Treebank. These lists of rules can be quite lengthy, even numbering in the thousands. Here's a glimpse why:

```
VP -> VB PP                # go to the store
VP -> VB PP PP            # go to the store in my car
VP -> VB PP PP PP       # go to the store in my car on the tollway
and more . . .
```

Language is recursive in nature. We can go on and on and on. CFG rules can be written to capture this recursion, as demonstrated in these rules, and illustrated in Figure 9.5.

```
Rules with recursion:
NP -> NN
NP -> Noun PP
PP -> Prep NP
```

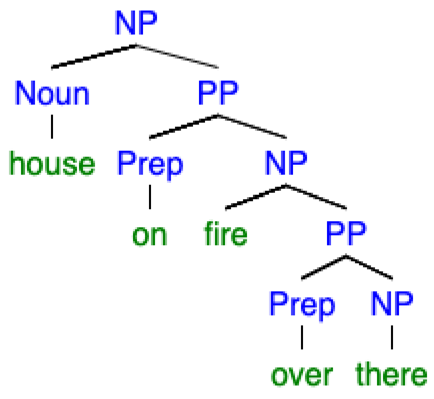


Figure 9.5: Recursive NP

The Penn Treebank Wall Street Journal corpus contains about a million words, and about 17K distinct rule types. We can look at the Penn Treebank in NLTK. The following code examples show how to identify individual files in the corpus. For a given file, we can print words, tagged words, and parsed sentences, as shown in the online notebook, 'Treebanks'.

First the corpus is loaded, then we look at a few file ids for later reference:

```
from nltk.corpus import treebank
treebank.fileids()[:10]
['wsj_0001.mrg',
 'wsj_0002.mrg',
 'wsj_0003.mrg',
. . .
```

Next we can look at words and POS tags for a file.

Code 9.6.1 — Load the treebank. Look at words and POS

```
print(treebank.words('wsj_0003.mrg'))
 ['A', 'form', 'of', 'asbestos', 'once', 'used', '*', ...]

print(treebank.tagged_words('wsj_0003.mrg'))
[('A', 'DT'), ('form', 'NN'), ('of', 'IN'), ...]
```

A parsed sentence can be examined with this code:

```
print(treebank.parsed_sents('wsj_0003.mrg')[0])
```

The output is shown in Figure 9.6.

9.7 Chunking

A *chunk* in NLP terminology is a word or sequence of words that acts as a constituent of a sentence. Chunks are often phrases such as noun phrases, verb phrases, prepositional phrases and so on. NLTK includes a conll2000 corpus, which is a chunking corpus of CoNLL (Conference on Computational Natural Language Learning). CoNLL is a yearly conference organized by SIGNLL (ACL's Special Interest Group on Natural Language Learning). The following code shows how to access the corpus, and output chunks for two sample sentences.

Code 9.7.1 — Chunk CoNLL Corpus. Output Chunks

```
from nltk.corpus import conll2000
for tree in conll2000.chunked_sents()[:2]:
    print(tree)
```

Below, we see the chunks in the second sentence. You can see that phrases are primarily phrase constituents of a sentence. We will see chunking in other parses in the next chapter.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

```
(S
  Chancellor/NNP
  (PP of/IN)
  (NP the/DT Exchequer/NNP)
  (NP Nigel/NNP Lawson/NNP)
  (NP 's/POS restated/VBN commitment/NN)
  (PP to/TO)
  (NP a/DT firm/NN monetary/JJ policy/NN)
  (VP has/VBZ helped/VBN to/TO prevent/VB)
  (NP a/DT freefall/NN)
  (PP in/IN)
  (NP sterling/NN)
  (PP over/IN)
  (NP the/DT past/JJ week/NN)
  ./.)
```

9.8 Summary

A formal grammar consists of a set of tokens and production rules. A context-free grammar, CFG, is a formal grammar that can be used to produce valid sentences, or check if sentences are valid. Formal grammars build constituents, phrases, from tokens. These phrases combine to form more complex phrases, clauses, or sentences. A probabilistic CFG, PCFG, can learn probabilities to assign to a parse from treebanks.

9.8.1 Going Further

Further exploration of CFGs:

- Chapter 12 of Jurafsky and Martin `https://web.stanford.edu/~jurafsky/slp3/12.pdf`
- A Python implementation of the CYK parser by Rob McHardy `https://github.com/RobMcH/CYK-Parser`
- An interactive CYK example `https://en.wikipedia.org/wiki/CYK_algorithm`

```
(S
  (S-TPC-1
    (NP-SBJ
      (NP (NP (DT A) (NN form)) (PP (IN of) (NP (NN asbestos))))
      (RRC
        (ADVP-TMP (RB once))
        (VP
          (VBN used)
          (NP (-NONE- *))
          (S-CLR
            (NP-SBJ (-NONE- *))
            (VP
              (TO to)
              (VP
                (VB make)
                (NP (NNP Kent) (NN cigarette) (NNS filters))))))))
    (VP
      (VBZ has)
      (VP
        (VBN caused)
        (NP
          (NP (DT a) (JJ high) (NN percentage))
          (PP (IN of) (NP (NN cancer) (NNS deaths)))
          (PP-LOC
            (IN among)
            (NP
              (NP (DT a) (NN group))
              (PP
                (IN of)
                (NP
                  (NP (NNS workers))
                  (RRC
                    (VP
                      (VBN exposed)
                      (NP (-NONE- *))
                      (PP-CLR (TO to) (NP (PRP it)))
                      (ADVP-TMP
                        (NP
                          (QP (RBR more) (IN than) (CD 30))
                          (NNS years))
                        (IN ago)))))))))))))
  (, ,)
  (NP-SBJ (NNS researchers))
  (VP (VBD reported) (SBAR (-NONE- 0) (S (-NONE- *T*-1))))
  (. .))
```

Figure 9.6: Annotation of a Sentence in the Penn Treebank

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

10. Syntax and Parsing

In the previous chapter, a formal grammar was discussed that could be used to assign a syntactic structure to an input sentence. In this chapter we look at NLP tools that parse natural language sentences. Syntax parsers input a plain text sentence, and output the sentence annotated with syntactic structure. Natural language is complex and prone to ambiguity. The first ambiguity that a parser will have to deal with is part of speech ambiguity. Many parsers perform POS tagging as an initial step. If the POS tagging is wrong, the errors will affect the accuracy of the parse. Consider the sentence: The luxurious *feel* of the coat made her *feel* beautiful. The first 'feel' in the sentence is a noun, but the very same word form appears later in the sentence as a verb.

Parsers must also deal with **structural ambiguity**, which is ambiguity in assigning a syntax structure to a sentence. This kind of ambiguity is illustrated in an old Marx brothers' joke: 'I shot an elephant in my pajamas. How he got in my pajamas I'll never know.' Here's a parse of the two interpretations of the sentence:

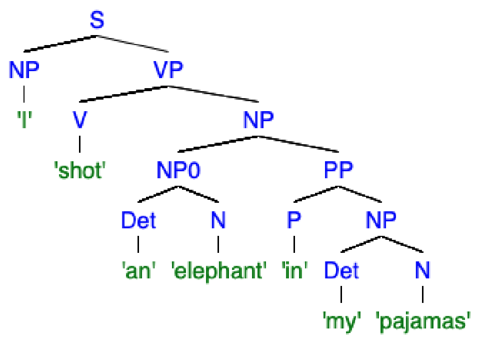


Figure 10.1: The elephant is in the pajamas

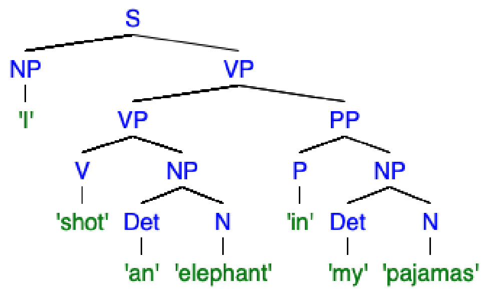


Figure 10.2: The shooter is in the pajamas

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

The ambiguity in this sentence comes from the attachment of the prepositional phrase 'in my pajamas'. In the parse on the left, the PP attaches to the elephant; whereas on the right, it attaches to the verb. This **attachment ambiguity** occurs whenever a phrase can be moved to more than one place in the syntax tree. Another kind of ambiguity is **coordination ambiguity**, which occurs with coordinating conjunctions such as 'and' and 'or'. Consider the phrase *old men and dogs*. Does this mean old men and old dogs, or old men and dogs of any age? As humans we would make a decision based on context, but this could prove troublesome to parsers.

The parse shown in the figure above is a **phrase structure grammar parse**, or PSG. Two other common forms of parses are the **dependency parse**, and a **semantic role label parse**. Each of these is discussed in more detail below. There are many freely available syntax parsers for each of these parser types. This chapter will provide information about many of the most popular parsers.

10.1 PSG Parse

The PSG (phrase structure grammar) parse is similar to the formal CFG discussed in the previous chapter in that it organizes sentence constituents into a hierarchy of phrases. Modern PSG parsers were trained on millions of annotated sentences in order to model the structure of natural language, which is much more complex and varied than the formal language of the CFG. Consider the sentence: John murdered the butler in the living room. The bracket form of the parse is:

```
[S [NP [DT The] [NN butler]]
     [VP [VBN murdered]
         [NP [NNP John]]
            [PP [IN in] [DT the] [NP [NN living] [NN room]]]
```

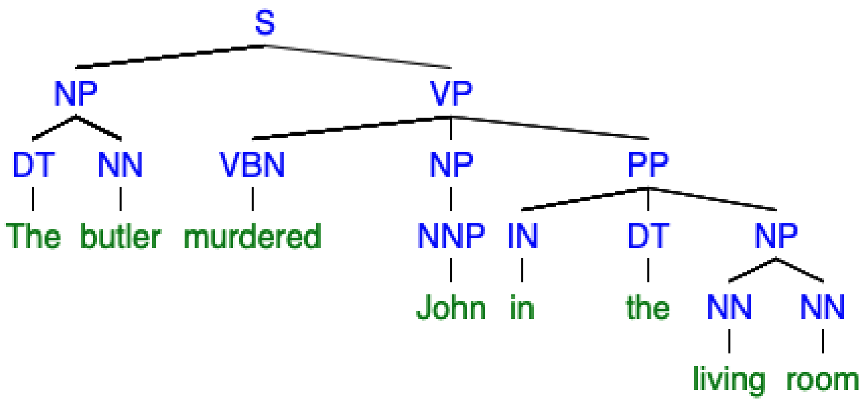


Figure 10.3: Phrase Structure Parse

The bracket form can be visualized in a tree hierarchy. Figure 10.3 shows the visualization service available here: `http://mshang.ca/syntree/`. Reading top-down, the visualization shows that the sentence consists of a noun phrase, NP, followed by a verb phrase, VP. The noun phrase is further broken down into tokens with POS tagging. The VP consists of a verb, a NP, and a PP. Each of these is further broken down until tagged tokens are reached at the bottom level.

Another visualization tool is available from AllenNLP, an open-source NLP research library created by the Allen Institute for Artificial Intelligence, which is closely linked with the University of Washington. AllenNLP offers many NLP tools, including PSG parsing, which they call *constituency parsing*. A technical paper [1] from the Allen Institute gives an overview of the NLP platform. The underlying technology uses PyTorch, an open-source machine learning library primarily developed by Facebook's AI group. The AllenNLP demos shared in this chapter were created on their demo web site: `https://demo.allennlp.org/`

The AllenNLP visualization of the sample sentence shows levels in expandable boxes rather that a tree. However, the same hierarchical concept is illustrated. Figure 10.4 shows that the sentence, shown in the topmost box, is divided into a NP and a VP. These two phrases are shown in the second level.

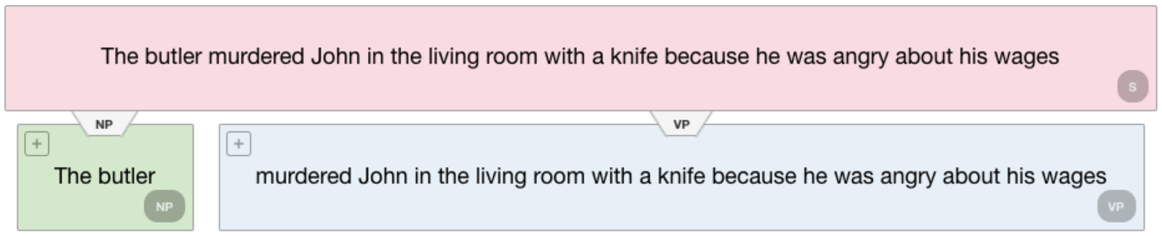

The butler murdered John in the living room with a knife because he was angry about his wages
S
NP
VP
The butler
NP
murdered John in the living room with a knife because he was angry about his wages
VP

Figure 10.4: Top Level Phrase Structure

Each layer of the AllenNLP visualization has + symbols to expand phrases deeper into the hierarchy. As with the tree visualization, the hierarchies continue until the POS-labeled tokens are revealed at the lowest level.

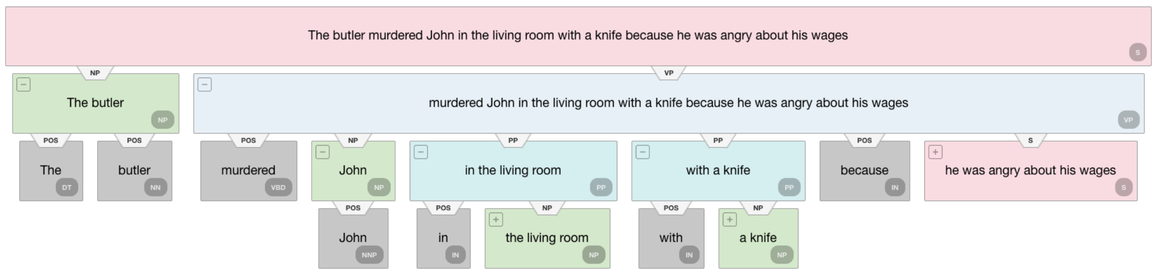

The butler murdered John in the living room with a knife because he was angry about his wages
S
NP
VP
The butler
NP
murdered John in the living room with a knife because he was angry about his wages
VP
POS
The
DT
POS
butler
NN
POS
murdered
VBD
NP
John
NP
PP
in the living room
PP
PP
with a knife
PP
POS
because
IN
S
he was angry about his wages
S
POS
John
NNP
POS
in
IN
NP
the living room
NP
POS
with
IN
NP
a knife
NP

Figure 10.5: Adding Levels to the Phrase Structure

Figure 10.6 shows the expansion of the subordinate clause: *because he was angry about his wages*. Recall that a clause expresses a complete proposition with a subject and a predicate. The first clause in the sentence, *The butler murdered John in the living room with a knife*, is an independent clause. This second clause provides additional information about the motivation of the butler. The dependent clause starts with the subordinating conjunction *because*. Looking at the expansion below, you can see how the dependent clause would be a complete sentence if the word 'because' were removed.

[1] `https://allennlp.org/papers/AllenNLP_white_paper.pdf`

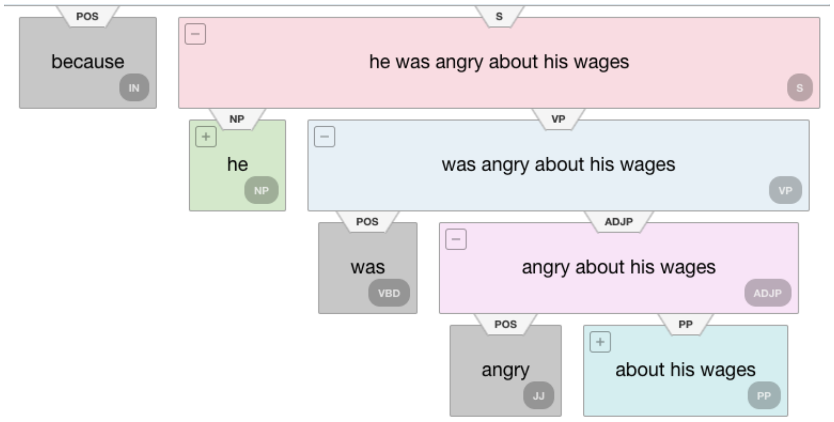


Figure 10.6: Clause Structure

10.2 Dependency Parse

A dependency parse shows relationships between the words of a sentence in an acyclic graph. The main predicate is the root node. The following dependency parse is of the sample sentence using the AllenNLP Demo page.The root of this sentence is the verb 'murdered'. The root has two direct dependents, the subject 'butler' and the direct object 'John'. Each link in the acyclic graph of a dependency parse is labeled according to the Stanford dependencies. A list of the Stanford dependencies is shown in Figure 10.11.

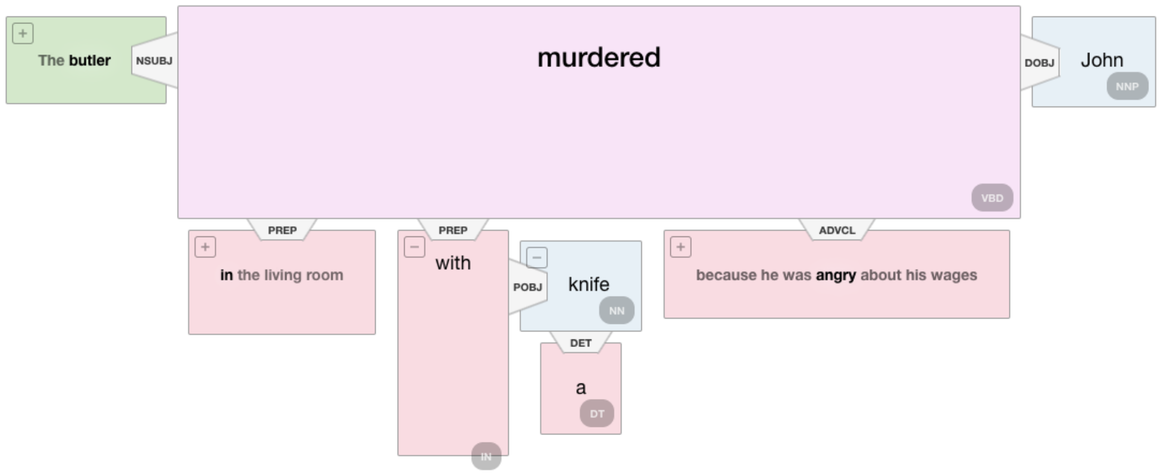


Figure 10.7: Dependency Parse

The major relations identified in this sentence:

- nsubj (butler) is the nominal subject of the verb
- dobj (John) is the direct object of the verb
- prep: two prepositional phrases are identified. The second one is further expanded to see the dependencies within the prepositional phrase
- advcl is the subordinate clause

The Stanford CoreNLP demo `https://corenlp.run/` of the same sentence shows the complete dependency parse and labels.

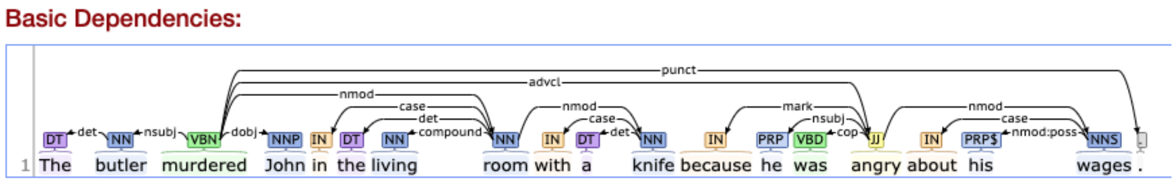

Basic Dependencies:
punct
advcl
nmod
case
det
compound
nmod
case
det
mark
nsubj
cop
nmod
case
nmod:poss
det
nsubj
dobj
DT NN VBN NNP IN DT NN NN IN DT NN IN PRP VBD JJ IN PRP$ NNS .
1 The butler murdered John in the living room with a knife because he was angry about his wages .

Figure 10.8: Dependency Parse, Another View

indexsyntax parsers!Stanford CoreNLP

Both dependency parses identify a copular structure in the adverbial clause *because he was angry about his wages* whereas the PSG parse did not. What is a copular clause?

10.2.1 Copular Clauses

A copular (aka linking) verb is a verb that links the subject of the sentence with some information about the subject. Consider the following two sentences:

```
John is handsome.
John is a lawyer.
```

In the first sentence, the copula ‘is’ links the subject, John, with an adjective describing John, handsome. The predicate in this sentence is ‘handsome’, not ‘is’. Likewise in the second sentence, the copula links the subject with the noun ‘lawyer’. The predicate is the noun ‘lawyer’.

Shouldn’t the verb be the predicate? There is evidence that the copula verb ‘be’ should be considered the predicate. The first piece of evidence is that the verb changes tense. The verb ‘is’ could be replaced by ‘was’ in these two sentences when referring to the past. The second piece of evidence is that the copula verb matches the subject. If the subject of these two sentences is changed to ‘Michael and John’, then the copula will change to ‘are’. The evidence that the copula verb is not the predicate reveals itself when the copular clause is moved into a subordinate position, as in these two constructions:

```
Mary thinks that John is handsome.
Mary thinks John handsome.
```

In the first construction, the copular clause remains the same, but in the second the ‘is’ becomes unnecessary. This is the strongest evidence that the copular verb ‘be’ is not the predicate. The following figures illustrate the copular construction from the Stanford CoreNLP online demo:

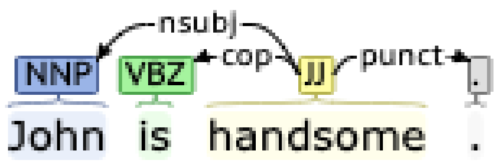


Figure 10.9: The predicate is JJ

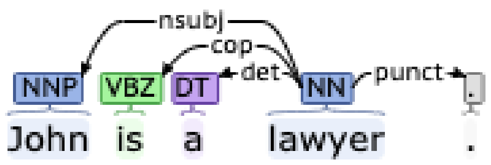
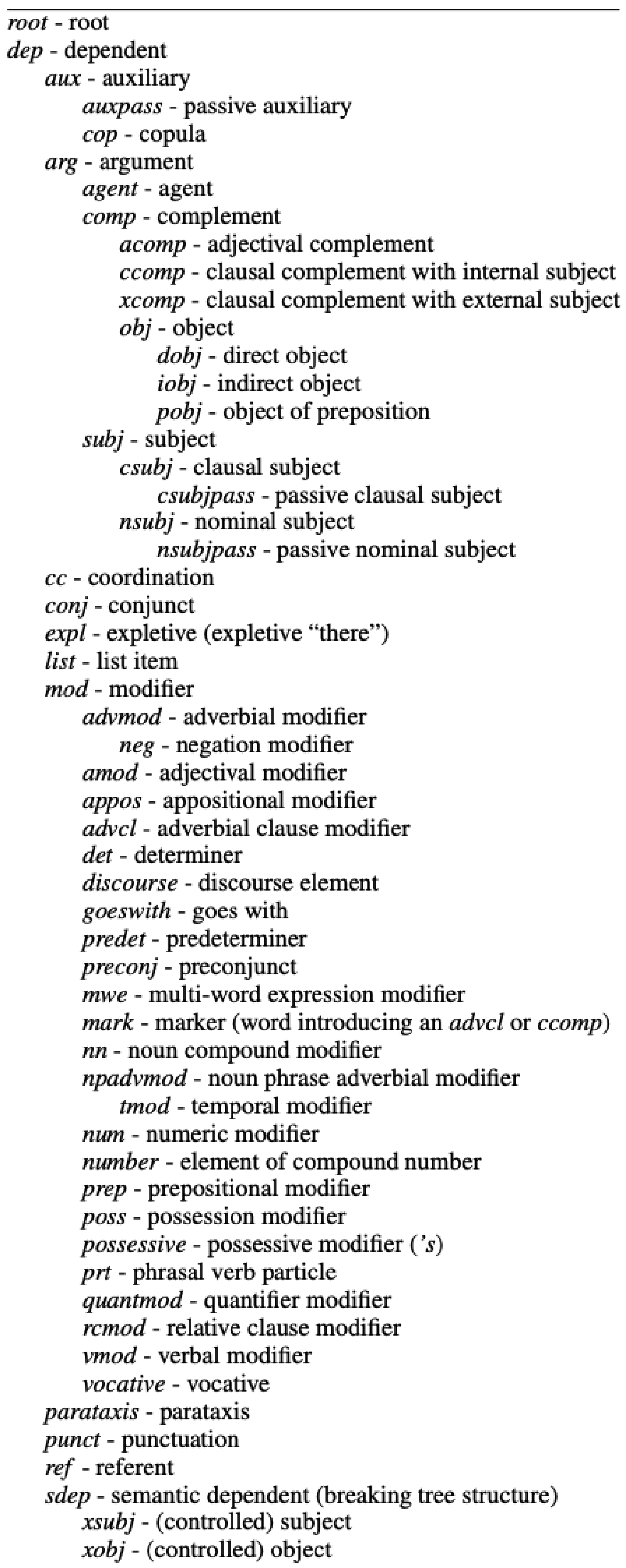


Figure 10.10: The predicate is NN



root - root

dep - dependent

- *aux* - auxiliary
  - *auxpass* - passive auxiliary
  - *cop* - copula
- *arg* - argument
  - *agent* - agent
  - *comp* - complement
    - *acomp* - adjectival complement
    - *ccomp* - clausal complement with internal subject
    - *xcomp* - clausal complement with external subject
    - *obj* - object
      - *dobj* - direct object
      - *iobj* - indirect object
      - *pobj* - object of preposition
  - *subj* - subject
    - *csubj* - clausal subject
      - *csubjpass* - passive clausal subject
    - *nsubj* - nominal subject
      - *nsubjpass* - passive nominal subject
- *cc* - coordination
- *conj* - conjunct
- *expl* - expletive (expletive "there")
- *list* - list item
- *mod* - modifier
  - *advmod* - adverbial modifier
    - *neg* - negation modifier
  - *amod* - adjectival modifier
  - *appos* - appositional modifier
  - *advcl* - adverbial clause modifier
  - *det* - determiner
  - *discourse* - discourse element
  - *goeswith* - goes with
  - *predet* - predeterminer
  - *preconj* - preconjunct
  - *mwe* - multi-word expression modifier
  - *mark* - marker (word introducing an *advcl* or *ccomp*)
  - *nn* - noun compound modifier
  - *npadvmod* - noun phrase adverbial modifier
    - *tmod* - temporal modifier
  - *num* - numeric modifier
  - *number* - element of compound number
  - *prep* - prepositional modifier
  - *poss* - possession modifier
  - *possessive* - possessive modifier (*'s*)
  - *prt* - phrasal verb particle
  - *quantmod* - quantifier modifier
  - *rcmod* - relative clause modifier
  - *vmod* - verbal modifier
  - *vocative* - vocative
- *parataxis* - parataxis
- *punct* - punctuation
- *ref* - referent
- *sdep* - semantic dependent (breaking tree structure)
  - *xsubj* - (controlled) subject
  - *xobj* - (controlled) object

Figure 10.11: Stanford Dependencies

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

10.2.2 Stanford Dependencies

indexsyntax parsers!Stanford CoreNLP

Figure 10.11 lists the Stanford Dependencies. A technical paper describes the meaning of each label: [2] In recent years, research has been done on creating a set of universal dependency relations that can be applied to most languages. The Universal Dependencies are slightly different. A link to the Universal Dependencies is provided in the Basic Dependencies pdf, linked in the footnote below. The Basic Dependencies pdf is worth reviewing also because it explains many important concepts.

The top level of the dependencies is 'dep', which is used whenever a more specific relation cannot be found. As discussed above, the 'root' will be the verb or an adjective/noun for copular clauses. The main dependencies of the predicate are subjects, agents, and complements. Auxiliary and copular verbs are dependents of the predicate. These types of verb are important for syntactic structure.

The subject is 'nsubj' for active sentences and 'nsubjpass' for passive sentences. Complements are constructions in the predicate that are more complex than simple direct and indirect objects. Examples of each type of complement are available in the Stanford Dependencies Manual.

Modifiers provide additional information in the sentence, but are not central players in the sentence. Examples of modifiers are prepositional phrases, adjective and adverbial clause modifiers. In the sample sentence diagrammed above, 'in the living room' and 'with a knife' are prepositional phrases that give additional information about what happened, but they aren't central to the main idea of who murdered whom. The adverbial clause 'because he was angry about his wages', provides additional information, giving motivation for the main action of the sentence.

This distinction between arguments and modifiers becomes more explicit in the next parse to be discussed, the semantic role label parse.

10.2.3 Common Sentence Patterns

One of the advantages of the dependency parse is that it explicitly labels the subject of the sentence. The dependency parse also labels any other major argument of the verb such as objects and complements. Objects are familiar to most people from school grammar: direct and indirect objects. Complements are less familiar; complements complete what the predicate is expressing.

There are three types of complements in the standard dependencies:

- acomp: adjectival complement; an adjective completes the meaning of the verb
- ccomp: a dependent clause with an internal subject
- xcomp: a dependent clause with no internal subject

Table 10.1 below helps identify some of the most common syntactic patterns in declarative sentences, where 'S-V' stands for 'subject-verb'.

Different verbs accept different numbers and kinds of objects or complements. In Table 10.1, notice that the verb 'lost' can have either no object, or it can have a direct object, but it cannot have both an indirect and direct object. The verb 'handed' can handle both an indirect object (the Packers), and a direct object (the game) in the sample sentence in the table. Linguists use the term *verb valency*, or *verb frames* to describe the number and types of arguments that verbs can accept.

In Table 10.1, notice the difference between ccomp and xcomp: ccomp is a dependent clause with an internal subject, but xcomp has an external object. The xcomp clause often begins with an infinitive 'to-V' or a gerund (ing form).

[2] https://www.asc.ohio-state.edu/demarneffe.1/papers/depling.pdf

Pattern	Example	Comment
S-V	The Cowboys lost.	no object or complement
S-V-dobj	The Cowboys lost the game.	direct object
S-V-iobj-dobj	The Cowboys handed them the game.	both direct and indirect objects
S-V-acomp	He looked tired.	'tired' completes the meaning
S-V-ccomp	I hope that you get the job.	'you' is the internal subject
S-V-xcomp	I love to take long walks.	no internal subject in clause
S-V-xcomp	I love walking in the woods.	no internal subject in clause

Table 10.1: Common Sentence Patterns

10.3 SRL Parse

indexsyntax parsers!SRL The Semantic Role Label parse (SRL) determines the role of sentence constituents relative to the predicate. SRL labels identify who did what to whom, when, where, how, and why. SRL is sometimes called *shallow semantic parsing* because SRL extracts meaning (semantics) from the sentence, going beyond the syntax. There are two categories of labels: arguments and modifiers. Arguments indicate the actors in the sentence, or the persons/objects acted upon. Modifiers give more details such as time and place of the action. Consider the sentence: *The butler murdered John in the living room with a knife because he was angry about his wages.* A visualization of the arguments and modifiers was created using the SRL Demo from AllenNLP, and is shown below:

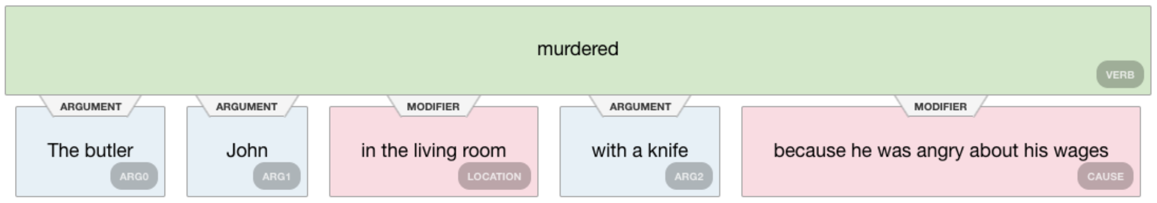


Figure 10.12: SRL Parse

The parse identifies the main predicate, murdered, in the green (top) bar. Arguments are in blue (dark grey) boxes, and modifiers are shown in pink (light grey) boxes.

10.3.1 Arguments

Arguments are numbered starting at 0. Generally, argument 0, Arg0, is the agent of the sentence, the one doing the action. Arg1 is often the passive element. In this chapter's sample sentence, the butler is Arg0 and John is Arg1 because the butler did the murdering and John was the one murdered. Labeling the butler as Arg0 and John as Arg1 is consistent, regardless of whether the sentence is written in active or passive voice:

```
Arg0 [The butler] murdered Arg1 [John].
Arg1 [John] was murdered by Arg0 [the butler].
```

Arg2 is often the instrument of the action, in this case, 'with a knife'. This shows that an argument can be a prepositional phrase, as well as subjects, objects, and other more weighty phrases.

The argument labels were developed using a proposition bank (PropBank) corpus developed by many researchers at the University of Colorado, Boulder. Verbs tend to fall into different categories, depending on what types of arguments they can take. Consider the verb 'ask'. In most sentences, there will be an Arg0 for the asker, an Arg1 for the one asked, and an Arg2 for the thing asked:

```
Arg0 [Mary] asked Arg1 [John] Arg2 [to go on a date].
```

Common argument meanings are found in Table 10.2. The numbering of the argument will vary by verb frame. Typically, Agent is A0 and Patient is A1.

Arg	**Meaning**	textbfExample
Agent	Entity doing the action	**John** broke the window.
Patient	Entity that is acted upon	John broke the **window**.
Instrument	Entity used in action	John broke the window with a **hammer**.
Beneficiary	Entity recipient	John gave the ring to **Mary**.

Table 10.2: Common Arguments

10.3.2 Modifiers

Modifiers are not arguments and are not necessary to complete the meaning of the sentence. Modifiers add useful information. Examples are TMP (temporal), LOC (location), DIR (where to/from), MNR (manner, how), PRP/CAU (purpose, causation, why). In the sample sentence, *in the living room* is a LOC modifier, and *because he was angry about his wages* is a CAU modifier.

Common modifiers are found in Table 10.3. The modifiers have more consistent meaning across verbs than do arguments. More information can be found in the PropBank annotation guidelines: `https://verbs.colorado.edu/~mpalmer/projects/ace/PBguidelines.pdf`

Mod	**Meaning**	**Example**
DIR	Motion along a path	John threw the papers **in the trash.**
LOC	Where the action happened	John was born **in Texas**.
MNR	How the action was performed	John broke the window **violently**.
TMP	When the action happened	John was born **in 1960**.
CAU	Reason for action	John moved to NY **because of his job**.
PNC	Motivation for an action	John saved money **for his move**.

Table 10.3: Common Arguments

The SRL parse has been used by some researchers to extract logical propositions from text. An example is extracting events from Wikipedia articles.[3]

10.4 Stanford CoreNLP

The Stanford NLP Group is one of the top NLP programs in the country. Their flagship toolkit, Stanford CoreNLP, provides a multitude of annotations for text, including tokenization, POS tagging,

[3] `http://ceur-ws.org/Vol-779/derive2011_submission_10.pdf`

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

dependency parsing, and more. The main page for the code is here: `https://stanfordnlp.github.io/CoreNLP/`. Installing CoreNLP is a 3-step process: (1) Download CoreNLP, (2) Unzip the download, and (3) Update your classpath variables to point to the expanded directory.

indexsyntax parsers!Stanford CoreNLP You can download CoreNLP for English and other languages. CoreNLP provides a customizable pipeline of annotations. Raw text is typically tokenized into sentences and tokens, then annotated with POS tags, and whatever annotations are specified in the code.

CoreNLP is written in Java, and you should use at least Java 8 to run the code. Within the CoreNLP download folder is a sample shell script to run CoreNLP from terminal. Also in that folder, is a sample Java demo program. A sample run is shown in a YouTube video for this chapter.

Below, a few key things to know will be pointed out. First, you will need for your IDE to be able to find the necessary Java jar files for CoreNLP. These can be attached in various ways depending on your IDE.

You can specify the annotations you want as shown below. Then run the annotations on your text. The demo code in the downloaded folder shows how to output the results.

Code 10.4.1 — Stanford CoreNLP. Java implementation

```
// Select the annotations
Properties props = new Properties();
props.setProperty("annotators", "tokenize, ssplit, pos, lemma, ner, parse,
            dcoref, sentiment");

// Set up the pipeline
StanfordCoreNLP pipeline = new StanfordCoreNLP(props);

// Run all the selected Annotators on this text
pipeline.annotate(annotation);
```

The first three annotations shown above tokenize the text, do sentence splitting, and POS tagging. These three will probably be needed for most applications. Annotating lemmas and named-entities may be useful. The parse annotation creates a dependency parse. The dcoref annotation creates inter-sentence links between nouns and referrents. This feature will slow down the annotations, so remove it if you don't need it. Finally, sentiment attaches an overall sentiment to each sentence.

10.4.1 Python Wrapper Stanza

Stanza is a Python wrapper for CoreNLP. Stanza can be installed with pip/pipe3. Once stanza is installed, the next step is to download English models.

```
$pip3 install stanza
$stanza.download('en')
```

The following code block shows how to set up the pipeline and annotate text. Not as many annotations are available (at the time of this writing) in Stanza as in the Java implementation. The code block below shows how to extract dependency parses.

Code 10.4.2 — Stanford CoreNLP. Stanza Python wrapper

```
import stanza

# set up the pipeline
nlp = stanza.Pipeline('en')

# set up the doc object on text
text = "Barack Obama was born in Hawaii.  He was elected president in 2008."
doc = nlp(text)

for sentence in doc.sentences:
    sentence.print_dependencies()
```

```
('Barack', 4, 'nsubj:pass')
('Obama', 1, 'flat')
('was', 4, 'aux:pass')
('born', 0, 'root')
('in', 6, 'case')
('Hawaii', 4, 'obl')
('.', 4, 'punct')
('He', 3, 'nsubj:pass')
('was', 3, 'aux:pass')
('elected', 0, 'root')
('president', 3, 'xcomp')
('in', 6, 'case')
('2008', 3, 'obl')
('.', 3, 'punct')
```

The online notebook shows how to run the code in a CoreNLP client-server setup, which may give faster performance when annotating many sentences at a time.

10.5 spaCy

indexsyntax parsers!spaCy The spaCy library provides industrial-strength NLP functions, and runs on Windows, Mac, or Unix. The nice thing about spaCy is that it is written in Python, and so it is easy to incorporate in your Python programs.

An online notebook shows how to install spaCy with pip/pip3, and how to download the small, medium, and large language models. The larger the model you use, the more accurate it is likely to be at the cost of a bit slower run time.

The first steps in using spaCy are to import spacy, load a model, and create a spaCy object on raw text. This is shown in the code block below. The medium model 'md' was selected.

The online notebook shows how to extract information token by token, including lemmas, POS, and more.

Code 10.5.1 — spaCy. Create an annotated object

```
import spacy

# load a model
nlp = spacy.load('en_core_web_md')

text = "Barack Obama was born in Hawaii.  He was elected president in 2008."

# create a spacy object
doc = nlp(text)
```

The next code block shows how to output dependency parse info. There is also a visualization package, displacy, demonstrated in the online notebook. The online notebook also shows some sample code to extract the subject and predicate from a sentence.

Notice in the code below that the organization in the doc is by token instead of sentence. The features extracted are the token, its dependency label, the text of its head word, and POS of its head word, and a list of its children.

Code 10.5.2 — spaCy. Output dependency parse

```
# output dependency

for token in doc:
    print(token, token.dep_, token.head.text, token.head.pos_,
          [child for child in token.children])
```

10.5.1 Other Parsers

The AllenNLP demo page `demo.allennlp.org` has a 'Usage' button next to the 'Demo' button that shows code used to generate the demo results. An online notebook shows how to extract an SRL parse using code inspired from the Usage example.

Another SRL parser is SENNA. There is a pdf in the GitHub demonstrating how to use SENNA, and the related video demonstrates usage. SENNA was one of the first programs to show the utility of training word embeddings with neural networks over massive amounts of data. We will revisit embeddings in the Deep Learning portion of the book.

10.6 Summary

This chapter looked at three types of sentence parsers, and a few toolkit implementations. The PSG parse is good for breaking a sentence into a hierarchy of phrases. This is a helpful first step for tasks such as noun phrase extraction and classification. The dependency parse explicitly identifies the subject of a sentence. The dependency parse also gives a deeper level of information about verb phrases by breaking down verb constituents into types: objects, or dependent clauses of varying types. The SRL parse clarifies who did what to whom with the numbered arguments, and modifiers can clarify semantics.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

10.6.1 Quick Reference

Reference 10.6.1 Dependency parse with Stanza

```
import stanza

# set up pipeline
nlp = stanza.Pipeline('en')

# set up the doc
text = " . . . "
doc = nlp(text)

for sentence in doc.sentences:
    sentence.print_dependencies()
```

Reference 10.6.2 Dependency parse with spaCy

```
import spacy

# load a model
nlp = spacy.load('en_core_web_md')

text = " . . . "

# create a spacy object
doc = nlp(text)

for token in doc:
    print(token, token.dep_, token.head.text, token.head.pos_,
        [child for child in token.children'])
```

10.6.2 Practice

Exercise 10.1 Come up with a fairly complex sentence with at least 12 words.

1. Hand draw a PSG tree of the sentence
2. Hand draw a dependency parse of the sentence
3. List and label the arguments and modifiers in an SRL parse of the sentence
4. Which parser(s) do you like best, least, and why?

■

10.6.3 Going Further

Links to parsers covered in this chapter:

- Stanford CoreNLP: `https://stanfordnlp.github.io/CoreNLP/`
- spaCy: `https://spacy.io/`
- Allen NLP: `https://allennlp.org/`

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

11. Annotated Parses

An annotated parse is any syntax parse that seeks to add more information than the standard parses explored in the last chapter. This chapter explores two approaches: the AMR parse, and a customized parse. indexsyntax parsers!AMR indexsyntax parsers!annotated

11.1 AMR

Abstract Meaning Representation (AMR), was first presented to the broader NLP community at the 2015 NAACL conference in Denver. The project involves numerous international researchers from top NLP and Linguistics departments. An AMR bank of annotated English sentences is available, and progress in this project can be tracked at `https://amr.isi.edu/`

```
# ::snt At that moment I was very busy trying to unscrew a bolt that had
        got stuck in my engine .
# ::save-date Thu Jan 21, 2016 ::file lpp_1943_298.txt
(t / try-01
      :ARG0 (i / i
            :ARG1-of (b2 / busy-01
                  :degree (v / very)))
      :ARG1 (u / unscrew-01
            :ARG0 i
            :ARG1 (b / bolt
                  :ARG1-of (s / stick-01
                        :ARG2 (e / engine
                              :poss i))))
      :time (m / moment
            :mod (t2 / that)))
```

The snippet above is a sample annotated sentence from the AMR website, with ID information about the sentence omitted. The structure of the sentence is shown with indents. As in a dependency parse, the root is the verb. The notation is a derivative of Penman notation[1]. Words have many meanings, the AMR identifies the particular meaning with numeric notation, as in `try-01`. The PropBank lexicon is used to identify the word sense.

In the parse above related to the SRL parse, Arg0 is 'I' and Arg1 is 'unscrew'. Within the Arg0 information, the additional modification of the subject 'very busy' is added. Within the Arg1 information, further information about what is being unscrewed from what is provided. Finally, there is a time modifier.

AMR recognizes concepts that are represented by variables, and constants which occur only once and do not need to be represented by variables. The concepts can be tracked by variable throughout several sentences in text.

AMR does not encode some word classes like prepositions and pronouns. AMR annotates named entities and provides wiki links when possible for named entities. There is much more to the AMR parses, since AMR needs to address the complexity of forms of English sentences and meanings. AMR is expected to have widespread utility for large-scale NLP projects. Further, given the structure, it would be possible to convert the annotation to a semantic representation of text which would be useful in logic programming.

The AMR project has dozens of researchers, even more human annotators, and large grants. Further, the work is primarily available to large institutions that can afford to pay hefty fees to the Linguistic Data Consortium (LDC). For these reasons, you may need to create your own custom parse. The next sections shows one approach that can be easily implemented in Python code, using standard parsers.

11.2 DeconStructure Algorithm

This section shares an approach I developed for my dissertation work, which began in 2013. I wanted some information from a dependency parse, and I wanted other information from the SRL parse, so I devised a data structure that would hold all the information I needed for a declarative sentence. The structure was then used to create questions from the declarative sentences. Additionally, I devised an algorithm to extract the information from various parses and combine it into the data structure. I called the algorithm the DeconStructure algorithm because it deconstructs a sentence into parts that are reassembled into a structure that encapsulates what the sentence is communicating.

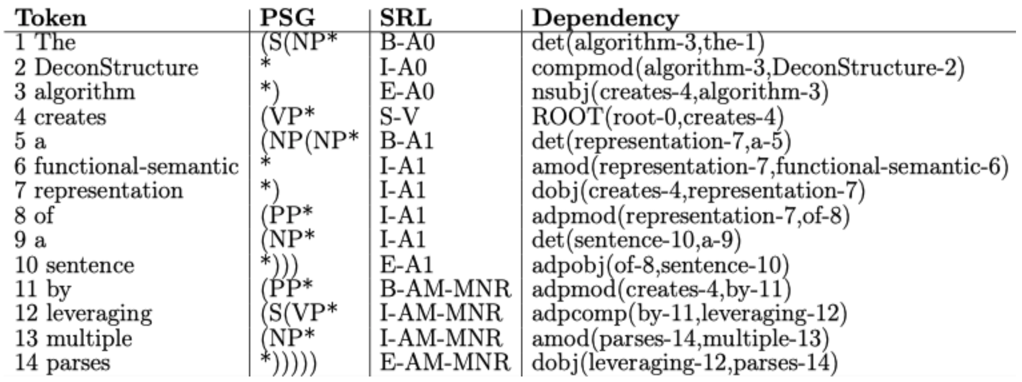
Figure 11.1 compares the PSG, SRL, and dependency parses for a sample sentence. Figure 11.2 shows the top-level information combined in the DeconStructure object, making it ready for question generation, as well as other NLP tasks. From this structure for this sentence, the following questions could be easily generated:

```
What does the DeconStructure algorithm create?
How does the DeconStructure create a functional-semantic
       representation of a sentence?
```

The DeconStructure algorithm is presented below. In the deconstruction phase, the sentence is parsed with both a dependency parse and an SRL parse. Additionally, word lemmas and parts of speech are gathered, along with named entity information. In the structure formation phase, the algorithm first divides the sentence into one or more independent clauses, then utilizes information from the parser to identify clause components, assigning each a label that represents its function within the clause.

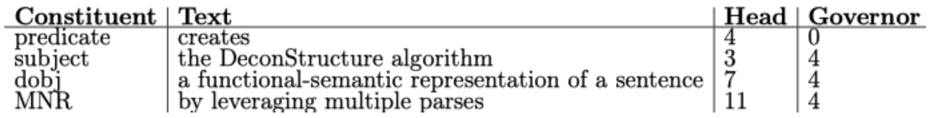
[1]`https://penman.readthedocs.io/`

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***



Token	PSG	SRL	Dependency
1 The	(S(NP*	B-A0	det(algorithm-3,the-1)
2 DeconStructure	*	I-A0	compmod(algorithm-3,DeconStructure-2)
3 algorithm	*)	E-A0	nsubj(creates-4,algorithm-3)
4 creates	(VP*	S-V	ROOT(root-0,creates-4)
5 a	(NP(NP*	B-A1	det(representation-7,a-5)
6 functional-semantic	*	I-A1	amod(representation-7,functional-semantic-6)
7 representation	*)	I-A1	dobj(creates-4,representation-7)
8 of	(PP*	I-A1	adpmod(representation-7,of-8)
9 a	(NP*	I-A1	det(sentence-10,a-9)
10 sentence	*)))	E-A1	adpobj(of-8,sentence-10)
11 by	(PP*	B-AM-MNR	adpmod(creates-4,by-11)
12 leveraging	(S(VP*	I-AM-MNR	adpcomp(by-11,leveraging-12)
13 multiple	(NP*	I-AM-MNR	amod(parses-14,multiple-13)
14 parses	*)))))	E-AM-MNR	dobj(leveraging-12,parses-14)

Figure 11.1: Parser Comparison



Constituent	Text	Head	Governor
predicate	creates	4	0
subject	the DeconStructure algorithm	3	4
dobj	a functional-semantic representation of a sentence	7	4
MNR	by leveraging multiple parses	11	4

Figure 11.2: Top-Level DeconStructure Information

A central underpinning of the DeconStructure algorithm is that sentence structure is a key aspect of natural language understanding. There is a reason we don't talk in word salads. The structure of our sentences communicates our thoughts, particularly in expository text. The question that arises from looking at the different types of parses is: How well do any of these types of parses correspond to how humans parse sentences, as we listen to a speaker or read text? Chomsky, as far back as the 1950s, proposed that we have an internal grammar in our minds that allows us to make sense of language. This idea has been controversial since its publication, but recent research in neuroscience has found some evidence that Chomsky was on the right track, although the research provides no insight as to whether these structures are innate or developed through experience. Using magnetoencephalography, researchers at NYU were able to identify distinct cortical activity that concurrently tracked auditory input (stripped of acoustic cues) at different hierarchical levels: words, phrases, sentences [2]. In other words, a hierarchy of neural processing underlies grammar-based internal construction of language. This exciting research may in the future be able to tease out what information these hierarchical processes actually encode. As that occurs, perhaps parsers could be developed in which sentence structure corresponds to our internally encoded structure. For example, it's doubtful that we hear the beginning of sentence and think: that's an NP. Rather, we think: that's what we are talking about, i.e., the subject. And the rest of the sentence is just telling us what action surrounded that subject and possibly other entities. While we await further advances from neuroscience, it seems practicable to continue research in sentence representation forms that correspond to an intuitive understanding of sentence meaning, such as the sentence representation discussed in this section, rather than a particular linguistic tradition.

[2] `https://www.nature.com/articles/nn.4186`

Algorithm 1 DeconStructure Algorithm

```
S ← set of parsed sentences
for each sentence s ∈ S do
    DivideIndepClauses(s)
    for each indepClause ic ∈ s: do
        Step 1: Add predicate complex
        ic[pred.label] ← predicate
        icRoot = pred.index
        Step 2: Add constituents
        for each dep ∈ dependencies do
            if dep.gov == icRoot then
                ic[const.label] ← dp
        Step 3: Add ArgMs to IC
        for each AM in ArgMs for icRoot do
            ic[AM.label] ← ArgM
        Step 4: Determine pp type
        for each pp in PPs do
            if pp == ArgN then
                pp.label = ppArg
            else
                pp.label = ppMod
        Step 5: Determine ic structure
        Determine ic type (passive, active, ...)
        Classify ic pattern
        Flag sentences with questionable parse
```

11.3 Summary

This chapter presented a very brief overview of the AMR project, and much more information can be found online: `https://amr.isi.edu/`. The AMR project is ongoing, so expect to hear more about its development and application going forward. The chapter also presented a more cost-effective and time efficient algorithm that can be implemented in Python for custom NLP applications. More details about the DeconStructure algorithm and its application to question generation can be found in my dissertation, available at `http://www.karenmazidi.com`

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

IV

Part Four: Documents

Part Four

Part Four examines what can be learned from larger sections of text. Documents are organized into a body of text, a corpus, for NLP processing. Chapter 12 looks at how to build a corpus, or find an existing corpus suitable for a project. Chapter 13 looks at information extraction techniques to identify important words and phrases in text. Chapter 14 introduces the vector space model, in which documents are represented as vectors in a corpus space. The vector space model allows us to find similar documents. Chapter 15 looks at the NLP technique of Topic Modeling, which identifies topic words in a corpus. Chapter 16 gives a very brief overview of semantics.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

12. Finding or Building Corpora

A *corpus* (plural *corpora*) is a collection of linguistic data such as written texts. The word corpus literally means body, so a natural language corpus is a body of text. Corpora are used by linguists to explore language, or develop and test hypothesis. Corpora are often used by NLP practitioners as input to machine learning or other systems in order to form a model of language.

The choice of a corpus is important because each body of text will be unique in the focus and the types of language. A corpus of tweets is quite different from a corpus of books of the King James Bible. A good corpus for an NLP project will be representative of the type of language that the project seeks to model.

12.1 Using existing corpora

The Brown Corpus was the first million-word corpus of English. Developed in the late 1960s by linguists at Brown University, the Brown Corpus consists of diverse texts from over 500 sources, organized by type, such as news, fiction, and so forth. The Brown Corpus is included in the NLTK corpus, along with other corpora such as select works from Project Gutenberg, a web chat corpus, and much more. A large number of the corpora have been annotated with parts of speech, named entities, or sentiment if relevant. A list of corpora is provided in Chapter 2 of the NLTK Book.

A google search for NLP corpus, or data set, will show that more and more data sets are being created. Here are a few data sources worth exploring:

- `https://nlpforhackers.io/corpora/`
- `https://github.com/jojonki/NLP-Corpora`
- Two new natural language dialog data sets from Google, released in September 2019 `https://ai.googleblog.com/2019/09/announcing-two-new-natural-language.html`
- The Wikipedia page about text corpora lists a few resources `https://en.wikipedia.org/wiki/List_of_text_corpora`

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

12.2 Building a corpus

Specialized NLP applications will need specialized corpora. For example, a medical NLP system could gather resources from open journals, books, and other training material. An NLP project to develop a dialog bot for customer support might build a training corpus through the Wizard of Oz (WoZ) technique. The WoZ technique uses real people to dialog with humans, modeling best practices. The dialog system can then learn from this corpus of transcribed transactions.

Using web sources can be helpful for building a corpus. The large movie review dataset available from Stanford was created by web scraping movie reviews, and then using the star ratings to programmatically label each review as positive or negative.

Finding data that can be programmatically labeled is ideal, however some tasks may require human annotation. A relatively inexpensive way to get human annotation is through crowd sourcing resources like Amazon's Mechanical Turk.

A *gold standard* data set is one that is commonly agreed upon by practitioners to be reliably annotated. These data sets are labeled data sets, in which inter-annotator agreement is quantified by statistical means such as a Kappa score. An analysis of corpora annotated by human experts versus workers in Mechanical Turk, available on the ACL web,[1] found that given a sufficient number of MTurk workers (around 4), the results were often of comparable quality to annotation by known experts, but at a much cheaper cost.

12.2.1 Getting Twitter data

Many sites that are frequently scraped provide an API to control access to the site. Twitter provides a Python library, tweepy, for downloading tweets. The tweepy package can be installed via pip/pip3. To use tweepy, you will need both a Twitter account and a Twitter Developer account. Setting up access is thoroughly documented on the Developer page `https://developer.twitter.com/en/apply-for-access.html`. Set up will take about an hour. For the Developer account, you can select a free personal account with standard access. You can always upgrade later. After getting developer access, you will be taken to a documentation page that will walk you through the steps necessary for access keys. Setting up access to tweepy is illustrated below, where you will fill in your personal keys.

Code 12.2.1 — Downloading Tweets. Setting up authorization

```
consumer_key = 'your info here'
consumer_secret = 'your info here'
access_token = 'your info here'
access_token_secret = 'your info here'

# make sure to install tweepy first
import tweepy as tw

auth = tw.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)
api = tw.API(auth, wait_on_rate_limit=True)
```

The next code block shows a search for hashtags 'nlp' and 'ml', filtered by retweets. A date was also provided to limit tweets by date.

[1] https://www.aclweb.org/anthology/D08-1027

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

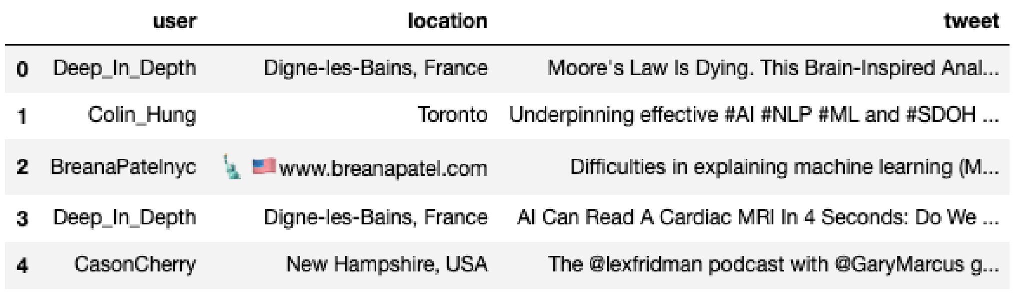
The actual search happens with the tw.Cursor() method. Arguments provided are the search words, date, language, and number of items. For this demo, just 5 tweets were retrieved.

The 'tweets' object that is returned by tw.Cursor() is iterable. Information about each tweet was extracted using a list comprehension. This list of lists was tossed into a Pandas data frame in order to make the output easier to read.

Code 12.2.2 — Downloading Tweets. Search and download code

```
# try a search
search_words = '#nlp+#ml -filter:retweets'
date_since = "2019-07-01"

# Collect 5 tweets about nlp
tweets = tw.Cursor(api.search,
              q=search_words,
              lang="en",
              since=date_since).items(5)

# save data in a list
user_location_text =
      [[tweet.user.screen_name, tweet.user.location, tweet.text]
                      for tweet in tweets]

# store it in a pandas data frame
import pandas as pd

df = pd.DataFrame(data=user_location_text,
                 columns=['user', 'location', 'tweet'])
df.head()
```

The output is shown below.



	user	location	tweet
0	Deep_In_Depth	Digne-les-Bains, France	Moore's Law Is Dying. This Brain-Inspired Anal...
1	Colin_Hung	Toronto	Underpinning effective #AI #NLP #ML and #SDOH ...
2	BreanaPatelnyc	🗽 🇺🇸www.breanapatel.com	Difficulties in explaining machine learning (M...
3	Deep_In_Depth	Digne-les-Bains, France	AI Can Read A Cardiac MRI In 4 Seconds: Do We ...
4	CasonCherry	New Hampshire, USA	The @lexfridman podcast with @GaryMarcus g...

Figure 12.1: Downloaded Tweets

12.3 Getting Wikipedia Data

Python library `wikipedia` is useful for downloading data from Wikipedia. For heavy downloading, other methods are suggested on Wikipedia's Help page. A link is provided in the online notebook. To install the package, use pip/pip3:

```
pip3 install wikipedia
```

Once the package is installed, it can be imported and used for downloading. The first code block shows how to get the first few sentences or paragraphs from an article in plain text.

Code 12.3.1 — Get a summary. First sentences from an article

```
import wikipedia

print(wikipedia.summary('Texas')
```

Once a specific page is accessed, several attributes of that page can be extracted as shown in the code below.

Code 12.3.2 — Article. Extract attributes

```
# set variable 'austin' to refer to a page
austin = wikipedia.page('Austin, Texas')

# extract title
austin.title

# extract the text from the article
austin_text = austin.content

# extract the links for the article
austin_links = austin.links
```

The online notebook gives further examples using the wikipedia package. Refer to the documentation for options for all methods.

Another online notebook provides an example of processing a dump of Wikipedia. The notebook provides a link to the Wikipedia dump download page, and code examples showing how to convert the dump to a text corpus using the gensim package.

12.4 Web Scraping

Extracting information from web pages, *web scraping*, can be achieved in many ways using different Python libraries. A program that crawls over the web from link to link extracting information is called a *web crawler*. This chapter looks at two libraries for web scraping:

- urllib - a Python library for handling URLs.
- Beautiful Soup - a library for extracting data from web sites. The latest version is Version 4.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

12.4.1 HTML

Most web pages use HTML (Hyper Text Markup Language) to form the structure of the web page. The following shows the structure of a simple web page.

```
<!DOCTYPE html>
<html>
<body>
  <h1>A level 1 heading</h1>
    <p>A paragraph</p>
  <h2>A smaller heading</h2>
    <p> another paragraph</p>
  <ol>
    <li> the first item </li>
    <li> the second item </li>
  </ol>
</body>
</html>
```

If you copy and paste the code above into a simple text editor, and save it with extension `html`, the file can be displayed as a web page by opening it with a web browser instead of a text editor.

Notice that the HTML tags are often in pairs, with a start tag < and and end tag </. The tags are not case sensitive, but lowercase tags are typically used for html. A tag that starts with <! is a comment. The first line of code above tells web browsers that this is an html document, and is required. All of the html for the page must be within the opening `<html>` and closing `</html>` tags. The part of the page that is visible in a web browser is contained in the body tags.

Heading tags can be from h1 to h6, with each higher number giving a smaller heading. White space in html is ignored and just serves to make the code more readable. Paragraph tags, p, are used for text blocks. Ordered or unordered lists are made with `ol` and `ul` tags, with each item surrounded by `li` tags. Indents can make a long list more readable for coding. Indents do not affect how the lists are displayed.

Head

The body tags are necessary for a web page to display something interesting, but the `<head>` tags are not required. The most common use for head tags is to create a title that will be displayed in a browser tab.

```
<head>
  <title>An Interesting Title</title>
</head>
```

CSS

There are formatting attributes that can be used within tags to adjust colors, fonts, and more. However, the recommended approach is to separate content from formatting. HTML tags are used for the structure and content of web pages. Cascading style sheets, CSS, are used to control the appearance of html elements.

The CSS code is usually stored in another file and imported within the head tags. The main advantage of using CSS is that the pages of a web site can have a uniform look, without having to explicitly code each element in each page. Each element type is formatted once in the style sheet and the defined styles will apply to all pages that import the css.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

```
<head>
<link rel="stylesheet" type="text/css" href="mystyle.css">
</head>
```

Since CSS is more about web design than content, it is not relevant to web scraping and so CSS won't be described here. Many excellent tutorials for CSS and web design exist on the web.

Image tags

An image tag provides all the information to display the image within `<img ...>`, so there is no need for a closing tag. In the example below, the src argument gives the file source. Optional width and height arguments specify the size of the image. The optional alt argument gives text that will display if the image cannot be found, and also is used by html readers for the visually impaired.

```
<img src="myfolder/myimage.png" width="500" height="400"
                  alt="Description here">
```

Links

Links to other pages are enclosed in <a> tags. The link text is what is displayed to the user with formatting that indicates it is a hyperlink. Notice the forward slash at the end of the html reference address. Some web browsers will automatically add this but some will not which causes two requests to the server. To make a hyperlink open in a new page, add the target attribute to the tag: `<a href="..." target="_blank">`.

```
<a href="https://www.somepage.com/index.html/" > link text </a>
```

Tables

Tables can be created with table tags:
- table tags define the start and end of the table
- tr tags define table rows
- td tags define data in the row, essentially creating columns
- th tags define a heading row which will be bold

The table created with the simple table tags below displays as follows. If you display the html, you will notice that the table is not formatted nicely. There are deprecated formatting attributes that could be used to control the display of the table elements. Instead, cascading style sheets, CSS, should be used.

ID Name GPA

101 Sally Smith 3.8

102 Mark Jones 3.24

```
<table>
  <tr>
    <th>ID</th>
    <th>Name</th>
    <th>GPA</th>
  </tr>
  <tr>
    <td>101</td>
    <td>Sally Smith</td>
    <td>3.8</td>
  </tr>

  <tr>
    <td>102</td>
    <td>Mark Jones</td>
    <td>3.24</td>
  </tr>
</table>
```

Learning More HTML

For most web pages, you can right-click on the page and choose View Page Source from the pop-up menu. You will see the basic HTML tags discussed above. Other tags you will see are script tags for javascript, and meta tags for keywords and other information used by web crawlers. Another common tag is div, which is just used to group elements together.

This is enough HTML background to start web scraping. There are many excellent free resources on the web for learning more about HTML.

12.4.2 How Web Sites Work

A **website** is just a collection of web pages and other files that share a common domain name, like amazon.com or wikipedia.org. A **web server** is a computer that receives requests for a web page from a browser and returns the content of the page. This works through a client-server system. The web browser is the client that makes requests from the web server that then makes a response. Clients and servers communicate through a protocol, HTTP Hypertext Transfer Protocol. Any transmission of information on the internet is done through TCP/IP Transmission Control Protocol and Internet Protocol that packages content for transmission and makes sure it arrives. A server will send web page information in chunks called packets, using TCP/IP protocol. A web browser collects these packets and displays the web page.

The HTTP protocol handles different types of requests and responses between clients and servers, including:

- GET - to request data from a resource
- POST - to create or update a resource
- PUT - similar to POST but will not create multiple resources if multiple PUTs are issued
- DELETE - to delete a resource

When a web browser wants to display a web page, the domain name is translated to an IP address through the DNS domain name system. DNS servers provide lookup of IP addresses for web sites. When you connect to the Internet, you are identified by your IP address, which is generally provided by your ISP internet service provider. A common standard for IP addresses is IPv4 which looks like four numbers separated by periods, where each number can be 0 to 255. The following could be an IPv4 address:

```
72.14.255.255
```

IPv4 addresses are 32 bits which allows for about 4 billion unique addresses. With the growth of the Internet, more addresses are needed and so a new protocol IPv6 is being used. IPv6 is 128 bits, written in hex with each group of digits separated by colons. This could be an IPv6 address:

```
2001:0db8:85a3:0000:0000:8a2e:0370:7334
```

Status Codes

An HTTP request will return content if possible but other conditions return a status code. Here are some common status codes that indicate something went wrong with the request:

- 400 Bad Request - the server did not understand your request
- 403 Forbidden - your client does not have access to the content
- 404 Not Found - the URL is not recognized
- 408 Request Timeout - the server shut down a request it felt had gone idle

This section described a very high-level view of communication between web browsers and web servers, with a focus on just information needed to understand web scraping.

12.4.3 Respecting Web Sites

A common misconception about content on the web is that if it's on the web, it's free content. Legally, the content provider owns the content and has the right to determine how that content can be used by others. When scraping data of a particular web site, make sure to read that site's Terms and Conditions. Many sites do not allow others to use the data for commercial purposes.

Limiting the rate at which you are scraping a web site can help prevent access issues with the web site for all users, as well as prevent you from getting blocked from the site. Scraping during off-peak hours can limit the impact of your crawler.

Before scraping a particular web site, check if they have provided an API. If a website has an API, they may block other kinds of access.

The robots.txt file

Many web sites have a robots.txt file that specifies how a site should be crawled or indexed for search engines. The file can be found on the root directory, ex: `www.some_website.com/robots.txt` The information in a robots.txt file can be hard to interpret. Here is an example to get started:

```
# Google
User-agent: Googlebot
Disallow: /nogooglebot/

# Everyone else
User-agent: *
Allow: /
```

This simple robots.txt file has a rule for Google's web crawler, specified in the User-agent field. The disallow field says that Google should not crawl in the /nogooglebot/ directory or any subdirectories. The second rule has * for User-agent, meaning everyone else. The Allow field has just the root /, so that means all directories can be crawled. In contrast, `Disallow: /` would mean that you can't access anything.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

12.4.4 Downloading text using the urllib library

The urllib library is part of Python, so there is nothing to install. The urllib library has several modules that work with URLS:

- urllib.request for opening and reading URLs
- urllib.error for handling exceptions raised by URL requests
- urllib.parse to parse URLs
- urllib.robotparser to parse robot.txt files

The urllib library is simple to use, and is useful for downloading text, as shown in the code below. A url is specified as a string. In this case, it is the address of a novel. The url is opened with the urlopen() method, then the page is read. Optionally, the content can be decoded. The page below was decoded with utf-8 encoding.

Code 12.4.1 — Using urllib. Reading text

```
from urllib import request

url = "http://www.gutenberg.org/files/2554/2554-0.txt"

with request.urlopen(url) as f:
    raw = f.read().decode('utf-8-sig')
print('len=', len(raw))
raw[:200]
```

The output from the code is shown below. More output is shown on the Jupyter notebook in the GitHub.

```
len= 1176966
'The Project Gutenberg EBook of Crime and Punishment, by Fyodor Dostoevsky'
```

The urllib modules can be used to download everything from web pages, but when a web page contains html, another library such as Beautiful Soup make the process easier.

12.4.5 Using the Beautiful Soup library

The Beautiful Soup library makes it easier to extract information from html tags. The `beautifulsoup4` library can be installed with pip/pip3: `pip3 install beautifulsoup4`

The example below extracts text from a web page news article coded in html. The code below first uses urllib to open the web page and read the content. Then a soup object is created. Finally, text is extracted from the soup object.

The text that was extracted is fairly clean but as shown in the Jupyter notebook online, there is still a bit of web page navigation and other material that is not really the text of the news article. The notebook also shows some text processing in Python to clean up the text.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Code 12.4.2 — Using Beautiful Soup. Extracting text

```
import urllib
from urllib import request
from bs4 import BeautifulSoup

url = 'https://nyti.ms/2uAQS89'
html = request.urlopen(url).read().decode('utf8')
soup = BeautifulSoup(html)

# extract text
text = soup.get_text()
```

Extracting paragraph tags

The next code chunk shows how to extract all p tags. These could then be processed further to extract text within the paragraphs. The code uses the same soup object created above, but selects just p tags.

Code 12.4.3 — Using Beautiful Soup. Extracting p tags

```
for p in soup.select('p'):
    print(p.get_text())
```

Extracting links

The next code chunk shows how to extract hyperlinks, 'a' tags. Just 10 are printed with this code.

Code 12.4.4
```
counter = 0
for link in soup.find_all('a'):
    counter += 1
    if counter > 10:
        break
    print(link.get('href'))
```

12.4.6 Alternatives

This section showed how to do web scraping with Beautiful Soup. Other popular frameworks for web scraping in Python are Scrapy and Selenium.

12.5 Practice

Exercise 12.1 Practice with the wikipedia API.

1. Use the wikipedia API to download page(s) on a chosen topic
2. Extract and remove links in the content. Count the percentage of links that are Wikipedia pages versus off-site pages
3. Use preprocessing techniques learned in earlier chapters to clean the remaining text
4. Use techniques learned earlier to find the most common important words in the remaining text

■

Exercise 12.2 Practice with the tweepy API.

1. Set up tweepy
2. Search for hashtags of your choice within a data range
3. Research nltk.tokenize.casual for tokenizing the tweets
4. Perform the text processing of your choice on the tweets

■

Exercise 12.3 Build a simple web crawler.

1. Starting with a url of your choice, scrape all the links off the page
2. Add the links to a queue, which can be as simple as a Python list
3. In a loop, go through each link in the queue, gathering more links to add to the end of the queue
4. Include a counter in your loop that stops when a certain number of links are found

■

12.6 Summary

This chapter showed how to get data from Twitter, Wikipedia, and via web scraping.

- A good discussion of best practices for compiling linguistic corpora can be found here: `http://icar.cnrs.fr/ecole_thematique/contaci/documents/Baude/wynne.pdf`
- An excellent example of compiling a gold-standard data set is found in a paper by Janyce Wiebe and others: `https://dl.acm.org/doi/10.3115/1034678.1034721`
- For unusual and historic texts, see the Oxford Text Archive: `https://ota.bodleian.ox.ac.uk/repository/xmlui/`

13. Information Extraction

The task of *information extraction*, IE, involves identifying specific, important information in unstructured text. The information to be extracted could be key terms, proper names of people and organizations, dates, times, events, and more. Older approaches to IE involved rule-based approaches, while more recent approaches involve machine learning. Information extraction can fall under the umbrella of *text mining*. A related field to IE is *information retrieval*, which involves finding documents in a corpus based on keyword search and other criteria.

IE is an extensive topic that can't be covered completely in one chapter. Here, the focus will be on three aspects: finding important words in a document or set of documents, finding and classifying entities: people, place, and organization names in text, and open information extraction.

13.1 Finding important words

Part 2 of the book showed how to find the most common words in text. The most frequent words are not necessarily the most important words. Apart from stop words, which can be removed, there are common words that just don't give much information about the importance of a word in a document. What is needed is a metric that can down-vote these common words in order to find words that reveal information about the content of a document. A good choice is idf, inverse document frequency, which discounts words that appear in many documents. The idf score is often combined with term frequency to form a tf-idf score. The tf-idf metric cannot be applied to a single document, a corpus of documents is needed for the calculations.

Term frequency is the count of a given term, t, in a document divided by the total number of words in the document, d. A lot of research has gone into finding the best formulas for tf and idf. Often, the log is taken of the term frequency. Here is a simpler formula that works well:

$$tf = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \tag{13.1}$$

The inverse document frequency is computed by first dividing the number of documents in the corpus, N, by the number of documents in which the term t appears at least once. Then the log is taken of this quantity. In the following equation, little d refers to one document, big D refers to the set of documents in the corpus.

$$idf = log\left(\frac{N}{|d \in D \& t \in d|}\right) \tag{13.2}$$

And tf-idf is just multiplying tf by idf. The more a term appears in a document, the higher the tf. The more the term appears in many documents, the lower the idf, which makes the overall tf-idf lower.

The jupyter notebook 'tfidf' in the GitHub shows how to build a tf-idf dictionary. Four texts were used, representing the text of one chapter in 4 different textbooks: anatomy, business law, economics, and geography. Each of the 4 documents is lower cased and tokenized, keeping only tokens that are not stop words and are alpha. The number of unique words in this small corpus is a little over 4,000.

First, a tf dictionary for each document is created using the function shown in the next code block. Using a Counter() object would be more efficient, but the code is written in simple Python for clarity.

First, the term frequencies are computed by simply counting. Then the term frequencies are normalized by the number of tokens in the document.

Code 13.1.1 — Create a tf dictionary. Function Definition

```
def create_tf_dict(doc):
# returns a normalized term frequency dict
# doc is a set of processed tokens

    tf_dict = {}
    tokens = word_tokenize(doc)
    tokens = [w for w in tokens if w.isalpha()
          and w not in stopwords]

    # get term frequencies
    for t in tokens:
        if t in tf_dict:
            tf_dict[t] += 1
        else:
            tf_dict[t] = 1

    # normalize tf by number of tokens
    for t in tf_dict.keys():
        tf_dict[t] = tf_dict[t] / len(tokens)

    return tf_dict
```

The word 'work' occurs in all four document. Below are the normalized term frequencies for each document.

```
tf for "work" in anat = 0.0004604051565377524
tf for "work" in buslaw = 0.0027739251040221915
tf for "work" in econ = 0.0006854009595613434
tf for "work" in geog = 0.0009285051067780873
```

The function shown in the next code block calculates idf for each term in the corpus vocabulary. First, the number of docs (4 in this case) is divided by the count of how many docs the term appears in. The log is taken of this quantity. The denominator has +1 added to avoid dividing by zero. Unfortunately this can result in a negative idf. In order to avoid this, +1 was added to both the numerator and the denominator. The idf for 'work' is 0 because it occurs in all documents. The idf for 'inflation' was around 0.9 because it occurs in only one document.

Code 13.1.2 — Create an idf dictionary. One dict for the corpus

```
import math

idf_dict = {}

vocab_by_topic = [tf_anat.keys(), tf_buslaw.keys(),
                  tf_econ.keys(), tf_geog.keys()]

for term in vocab:
    temp = ['x' for voc in vocab_by_topic if term in voc]
    idf_dict[term] = math.log((1+num_docs) / (1+len(temp)))
```

Now we can create a tf-idf dictionary for each document by multiplying the tf for each term in a document by the idf of the term in the corpus.

Code 13.1.3 — tf-idf. Function Definition

```
def create_tfidf(tf, idf):
    tf_idf = {}
    for t in tf.keys():
        tf_idf[t] = tf[t] * idf[t]

    return tf_idf
```

The online notebooks shows that the dictionaries are sorted to find the most important terms.

```
anatomy: sympathetic, system, autonomic, parasympathetic, receptors

business law: damages, party, breach, nonbreaching, remedies

economics: inflation, prices, index, price, basket

geography: islands, island, antartica, ozone, pacific
```

Looking at the top words from each document gives a good indication of what the document is about. However, seeing 'island' and 'islands' indicates that lemmatization would have been helpful.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

In this section we looked at how to calculate tf-idf for a corpus using a fairly general tf-idf formula. The tf-idf approach has received a lot of academic interest over the years. Consequently, there are many variations of the formulas for both tf and idf. There are many pre-built tf-idf functions in various libraries.

In the machine learning chapters, we will use the tf-idf vectorizer in sklearn. A notebook in the GitHub Part 4 (Exploring ACL2020abstracts) gives an example of extracting tf-idf terms with sklearn. See the second half of the notebook.

13.2 NER: Named Entity Recognition

The Named Entity Recognition (NER) task is to identify and label sequences of words that represent names of entities such as persons, countries, organizations and more. NER is an important component of many NLP systems such as question answering or information retrieval tasks. Consider the following sentence taken from *The Wall Street Journal* on 10/3/2019:

> The European Union's top court gave judges in the bloc broader power to order the removal of Facebook (FB 1.13%) posts, dealing a fresh blow to the U.S. tech giant as it faces growing regulatory headwinds on both sides of the Atlantic.

Running this sentence through AllenNLP NER, using both models on available on the demo page, identified the following named entities:

```
ORG:  European Union
ORG:  Facebook
ORG:  FB
PERCENT:  1.13 %
GPE:  U.S.
LOC:  Atlantic
```

13.2.1 Evolution of NER

Like many NLP approaches, early NER attempts used hand-crafted rules and external resources such as lexicons and ontologies. These rules-based approaches were overshadowed by machine learning approaches that required extensive feature engineering. A popular statistical approach is conditional random fields, CRFs. More recent work uses neural networks, which can do their own feature engineering.

Many NLP frameworks provide NER annotations. In addition to AllenNLP discussed above, other NER systems are available through Stanford CoreNLP and SpaCy.

13.2.2 Stanford NER

As of this writing, Stanford NER uses a combination of CRF (Conditional Random Field) sequence models, and rules. The NER recognizes a total of 24 classes:

- Names: PERSON, LOCATION, ORGANIZATION, MISC
- Numeric: MONEY, NUMBER, ORDINAL, PERCENT
- Temporal: DATE, TIME, DURATION, SET
- Additional classes: EMAIL, URL, CITY, STATE_OR_PROVINCE, COUNTRY, NATIONALITY, RELIGION, (job) TITLE, IDEOLOGY, CRIMINAL_CHARGE, CAUSE_OF_DEATH, Twitter, etc. HANDLE

The next code block demonstrates using the Python implementation of CoreNLP for NER on 4 sentences of various topics that contain named entities and dates. The full notebook, NER Comparison, is available in the GitHub. The input text is as follows:

> The Hawaiian Islands became the fiftieth US state in 1959. Since the passage of the Social Security Indexing Act of 1972, the level of Social Security benefits increases each year along with the Consumer Price Index. The leading case, perhaps the most studied case, in all the common law is Hadley v. Baxendale, decided in England in 1854. Lyndon Baines Johnson (August 27, 1908 – January 22, 1973), often referred to as LBJ, was an American politician who served as the 36th president of the United States from 1963 to 1969.

Code 13.2.1 — NER. Stanford CoreNLP

```
from stanfordnlp.server import CoreNLPClient

import os
os.environ['CORENLP_HOME'] =
         r'/your path here/stanford-corenlp-full-2018-10-05'

# set up the client
with CoreNLPClient(annotators=['tokenize','ssplit','pos','lemma','ner'],
         timeout=60000, memory='16G') as client:
    # submit the request to the server
    ann = client.annotate(text)

    print('\nTokens \t POS \t NER')
    sentence_count = 1
    for sentence in ann.sentence:
        print('\nSentence', sentence_count)
        for token in sentence.token:
            if token.ner != 'O':
                print (token.word, '\t', token.pos, '\t', token.ner)

        sentence_count += 1
```

The full output is visible online. Stanford CoreNLP recognized 22 entities, including examples in the selected output below. However, 'LBJ' was labeled an organization instead of a person.

```
Sentence 1
Hawaiian   JJ   LOCATION
Islands   NNPS   LOCATION
fiftieth   NN   ORDINAL
US   NNP   COUNTRY
1959   CD   DATE
. . .
Sentence 3
```

```
Hadley   NNP   PERSON
Baxendale   NNP   PERSON
England   NNP   COUNTRY
1854   CD   DATE
```

13.2.3 SpaCy NER

The SpaCy NER system uses models trained on the OntoNotes 5 corpus, a hand-annotated corpus covering multiple genres of text. SpaCy NER labels 18 entity types: PERSON, NORP (nationalities or religious or polical groups), FAC (facilities), ORG, GPE, LOC, PRODUCT, EVENT, WORK_OF_ART, LAW, LANGUAGE, DATE, TIME, PERCENT, MONEY, QUANTITY, ORDINAL, CARDINAL.

Code to run SpaCy NER is shown below. This code is part of the same notebook mentioned above.

Code 13.2.2 — NER. SpaCy

```
import spacy
nlp = spacy.load('en_core_web_md')

doc = nlp(text)

for ent in doc.ents:
    print(ent.text, ent.label_)
```

The SpaCy output recognized 18 entities, and correctly identified LBJ as a person. The full output is shown below.

```
Hawaiian Islands GPE
fiftieth ORDINAL
US GPE
1959 DATE
the Social Security Indexing Act LAW
1972 DATE
Social Security ORG
each year DATE
Hadley v. Baxendale PERSON
England GPE
1854 DATE
Lyndon Baines Johnson PERSON
August 27, 1908 - January 22, 1973 DATE
LBJ PERSON
American NORP
36th ORDINAL
the United States GPE
1963 to 1969 DATE
```

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

13.3 Open IE

The NLP subfield of *Information Extraction* (IE) extracts structured information from unstructured plain text. The developers of an IE system first must decide what kinds of things to look for in text, and sometimes what kinds of text the system should be tailored for. Many systems extract information in tuples of the form of type(arg1, arg2). For example, the tuple was-born-in(Barack-Obama, Hawaii) represents the fact that Obama was born in Hawaii.

These extracted tuples can be fed into logic reasoning systems, creating a knowledge base, or for downstream NLP applications. The NER discussed above is an example of IE, as is event extraction, relationship extraction, and email meta data extraction. Application of IE is quite diverse, from building an index for a digital library or search engine, to extracting person profiles from resumes, and more.

13.3.1 Examples from AllenNLP

Sentences from a U.S.History text were used in the ALLenNLP Demo (`demo.allennlp.org`), interface to show examples of open AI. The first sentence identified arg0 and arg1 from the opening phrase of the sentence. Notice that this result is quite similar to the SRL parse.

```
While monarchies dominated eighteenth-century Europe, American revolutionaries
were determined to find an alternative to this method of government.
```

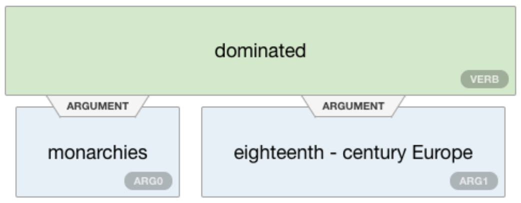


Figure 13.1: AllenNLP IE Example 1

Notice that nothing was extracted from the main clause of the sentence. In the example below, information was extracted from the main clause but not the subordinate clause. These examples show that open IE is not a completely solved NLP problem. There is room for improvement.

```
Jefferson's Declaration of Independence affirmed the break with England
but did not suggest what form of government should replace monarchy,
the only system most English colonists had ever known.
```

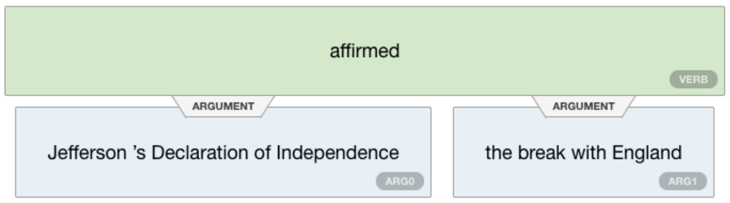


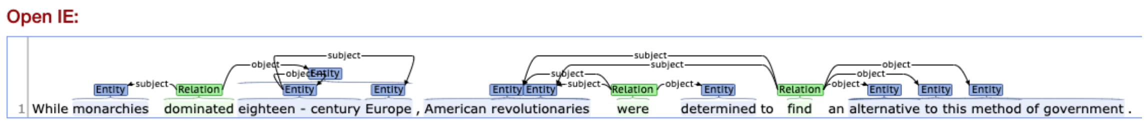
Figure 13.2: AllenNLP IE Example 2

13.3.2 Stanford CoreNLP Open IE

Stanford's Open Information Extraction system uses linguistic structure to find relationships in data. The advantage of this approach is that it enables relation extraction for more than just a set of predefined relations. As explained on the software's main page (`https://nlp.stanford.edu/software/openie.html`), independent and dependent clauses are extracted from a sentence. Then the clauses are shortened, then further reduced to OpenIE triples. The Stanford OpenIE software is part of Stanford CoreNLP. The web page referenced above has a sample program to use with Java 8 or higher.

The following is a screen shot from Stanford's online demo at `https://corenlp.run/` Three relations were found, with verbs dominated, were, and find. The relations are:

- dominated(monarchies, eighteen-century Europe)
- were(Amerian revolutionaries, determined)
- find(American revolutionaries, alternative to this method of government)



1 While monarchies dominated eighteen - century Europe , American revolutionaries were determined to find an alternative to this method of government .

Figure 13.3: Stanford IE Example

13.4 Practice

Exercise 13.1 Practice with tf-idf.
1. Use the wikipedia API to download multiple pages on a chosen topic
2. Preprocess the pages using NLP techniques learned earlier
3. Use tf-idf to find the most important words in the remaining text
4. Sort the words by importance

Exercise 13.2 Practice with NER.
1. Use the wikipedia API to download page(s) on a chosen topic
2. Following the sample code in the NER notebook in the GitHub, extract NER with the platforms of your choice
3. Compare the results of the platforms

Exercise 13.3 Practice with Open IE.
1. Use the wikipedia API to download a page on a chosen topic
2. Using AllenNLP's OpenIE, extract information from the page(s)
3. Write a Python function to combine the extracted information into sentences to form a simple summary of the text

13.5 Summary

There are many different tasks under the umbrella of Information Extraction: keyword extraction, NER, Open IE, and more. Information extraction is generally not a goal in itself but a means to an end in a larger NLP project. Therefore, it is an important skill to have.

Examples of downstream NLP projects that might use IE include:

- populating an ontology
- extracting events for an emergency preparedness app that scours tweets for happenings
- text summarization tasks
- question answering systems

13.6 Further Reading

- The classic book on Information Retrieval is by Manning, Raghavan, and Schutze: `https://nlp.stanford.edu/IR-book/`
- A recent (2018) survey of NER techniques is provided in a COLING paper by Yadav and Bethard, available on the ACL web `https://www.aclweb.org/anthology/C18-1182.pdf`

14. Vector Space Models

Vector space models are used in information retrieval and document ranking systems to find relevant documents in a corpus. Vector space models can also be used to cluster similar documents, as well as categorize them. Other applications in which vector space models have been used include sentiment analysis applications, and plagiarism detection applications.

14.1 Overview

A *vector space model* represents documents in a corpus as vectors in a common vector space. In the figure below, a and b represent two documents. The cosine of the angle between the two vectors is a measure of how similar the documents are. Cosine ranges from -1 to +1. The vectors are composed of positive values representing term frequencies or tf-idf, which means that the cosine similarity will actually range between 0 and 1. Cosine values close to 0 indicate little similarity.

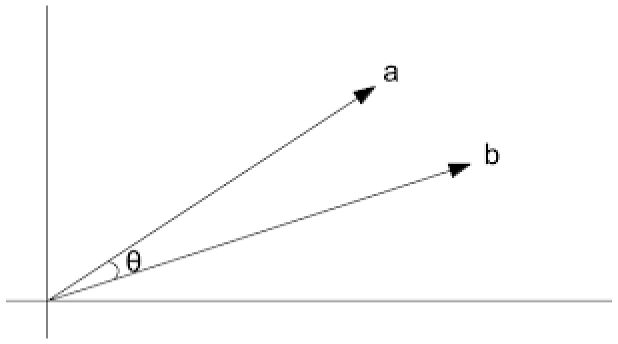


Figure 14.1: Cosine Similarity

14.1.1 Representing documents as vectors

A vector space model represents documents as vectors in a shared vector space. Consider an example of two documents, a and b, with a combined vocabulary of 10 terms:

- a: Her favorite hobbies are reading, walking, knitting, and learning new things.
- b: His favorite hobbies are reading, discussing politics, and walking the dog.

Removing stop words and punctuation, and lemmatizing tokens, each document is now a list of tokens.

A = [favorite, hobby, read, walk, knit, learn, thing]

B = [favorite, hobby, read, discuss, politics, walk, dog]

And the vocabulary is the set of all words in the corpus, alphabetized:

Vocab = [discuss, dog, favorite, hobby, knit, learn, politics, read, thing, walk]

After creating the alphabetized vocabulary, each document can now be represented as a numeric vector where each position in the vector represents the corresponding token in the alphabetized vocabulary, and each element is the count of that token in the document.

```
A = [0, 0, 1, 1, 1, 1, 0, 1, 1, 1]
B = [1, 1, 1, 1, 0, 0, 1, 1, 0, 1]
```

14.1.2 Cosine similarity

Cosine similarity is computed as the dot product of the two vectors, normalized by the vector norms:

$$cos(\theta) = \frac{A \cdot B}{||A||||B||} \tag{14.1}$$

If the cosine is large (closer to 1) then the vectors are closer, indicating that the documents are similar. A cosine value of 1 would mean that the documents are identical.

For the two vectors above, A dot B is 4. The norm of A and B are both 2.65. This gives us:

$$\frac{4}{2.65 * 2.65} = .57 \tag{14.2}$$

This value is closer to 1 than 0, although not by much. The two documents are somewhat similar. This brings up a good question: what is a reasonable cut-off point for deciding that two documents are similar? This will actually depend on the corpus and the task. One approach would be to randomly sample documents and compute cosine similarity, then find an average. Anything above this average would be considered fairly similar. Another approach is to select a threshold like 0.75. Again, the optimal threshold will depend on the application, and whether it should it err towards finding too many or too few similar documents.

14.2 Vector space model from scratch

In order to see exactly how vectors are created from documents, this section shows a Python approach that vectorizes documents from scratch. A later section will demonstrate a vectorizer from sklearn. The full code in this chapter is in the GitHub Jupyter notebook ‘Vector Space Model’.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

14.2.1 Creating vectors from documents

Text preprocessing is very important for the vector space model. The corpus used is 4 text documents representing 4 subjects of anatomy, business law, economics, and geography. These documents were divided into two sections so that the corpus consists of 8 documents.

Each document is preprocessed by lower casing, tokenizing, removing non-alpha tokens, and removing stopwords. Each token was also lemmatized to normalize terms to the lemma form. Next, a vocabulary is created. The vocabulary is an alphabetized list of words in a document.

Code 14.2.1 — Building a vocabulary. Using sets and lists

```
vocab = set()
for doc in docs_preprocessed:
    doc_set = set(doc)
    vocab = vocab.union(doc_set)

vocab = sorted(list(vocab))
print('vocab length:', len(vocab))
vocab[:5]
```

In the code above, each document's tokens are placed in a set that is joined to the vocab set with the union function. The vocab set is converted to a list and sorted alphabetically. The output below shows the vocabulary is 3601 unique words. The first 5 alphabetically are displayed.

```
vocab length: 3601
['abandoned', 'abdominal', 'abide', 'ability', 'able']
```

The code below converts each doc to a vector. Each doc is a 'vector', represented as a Python list. Each element in the list represents the corresponding word in the alphabetized vocabulary list. Each element is the count of that vocabulary word in that document. Below the code, the first 10 elements of the vector for the first document are shown.

Code 14.2.2 — Convert documents to vectors. 'vectors' is a list of lists

```
vectors = []
for doc in docs_preprocessed:
    vec = [doc.count(t) for t in vocab]
    vectors.append(vec)
print(vectors[0][:10])
```

```
[0, 4, 0, 0, 0, 0, 0, 0, 0, 1]
```

14.2.2 Computing cosine similarity

The following code block shows a cosine similarity function implemented with numpy dot product and norm functions.

Code 14.2.3 — Cosine similarity. Using NumPy

```
from numpy import dot
from numpy.linalg import norm

# function to compute cosine similarity
def cos_sim(v1, v2):
    return float(dot(v1, v2)) / (norm(v1) * norm(v2))
```

Now that a function to compute cosine similarity is created, the code and output below shows the first document compared to all the 8 documents using cosine similarity. Of course, anat1 will have a perfect similarity with itself, 1.0. Also, it has very high similarity with anat2, 0.72. The similarity with other vectors is low. This shows that this simple approach got very good results in finding document similarity. A key to these good results is removing stop words. If stop words had not been removed, the documents would have all been more similar to each other.

```
# compute cosine similarity for the first doc, paired with all docs
for i, vec in enumerate(vectors):
    print('cosine similarity anat1 and vector', i+1, '=',
              cformat(cos_sim(vectors[0], vec), '.2f'))
```

```
cosine similarity anat1 and vector 1 = 1.00
cosine similarity anat1 and vector 2 = 0.72
cosine similarity anat1 and vector 3 = 0.05
cosine similarity anat1 and vector 4 = 0.05
cosine similarity anat1 and vector 5 = 0.06
cosine similarity anat1 and vector 6 = 0.06
cosine similarity anat1 and vector 7 = 0.06
cosine similarity anat1 and vector 8 = 0.10
```

14.3 Vector space model using sklearn

The code below shows how to use sklearn to vectorize documents. This code is obviously shorter than the code above, but the exercise of building code from scratch also built our understanding of what is happening in a vector space model.

Code 14.3.1 — Vectorizing docs. Using sklearn

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer()

docs2 = [' '.join(docs_preprocessed[i]) for i in range(len(docs))]

tfidf_docs = tfidf_vectorizer.fit_transform(docs2)
tfidf_docs.shape
```

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

The docs ‘shape’ is 8 by 3593, meaning that there are 8 docs sharing a vocabulary space of 3593 unique terms. The code below shows that cosine similarity results are similar to the results achieved in the from-scratch code earlier in the chapter. The difference is that the scratch code used term frequency but the sklearn code used tf-idf.

Code 14.3.2 — Cosine similarity. Using tf-idf vectors

```
from sklearn.metrics.pairwise import cosine_similarity

cosine_similarity(tfidf_docs[0], tfidf_docs)
```

```
array([[1.        , 0.70338879, 0.02614208, 0.0192754 , 0.02797854,
        0.02504257, 0.0273924 , 0.05087871]])
```

14.4 Summary

The vector space model is an interesting technique at the intersection of NLP and Information Retrieval. Documents are represented as vectors in a common vector space. This enables finding similar documents in a corpus. The cosine similarity metric measures how close two documents are. Cosine similarity ranges from 0 to 1, with values closer to 1 having the most similarity.

14.5 Practice

Exercise 14.1 Practice with cosine similarity.
1. Use the wikipedia API to download a page on a chosen topic
2. Preprocess the text using NLP techniques demonstrated in this chapter
3. Create a vocabulary of the remaining text
4. Create count vectors for each sentence in the text
5. Create some search query vectors
6. Run cosine similarity to see if you can extract the most relevant sentence for a given search query vector

■

14.6 Exploring further

There are many implementations of cosine similarity besides the sklearn implementation. For example, the gensim package’s cosine similarity uses special memory management techniques to enable handling of a very large corpus. Documentation is available here: `https://radimrehurek.com/gensim/similarities/docsim.html`

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

15. Topic Modeling

In topic modeling, a *topic* is a set of words; technically, a multinomial distribution over words. Documents are mixtures of topics. Topic modeling discovers these two things simultaneously: (1) topics in the corpus, and (2) which topics are present in documents.

15.1 Intuitive explanation

Figure 15-1, from an ACM article by David M. Blei [1] shows the big picture of topic modeling. Four topics are identified, represented by the page icons on the left side of the illustration. Each topic consists of a set of words, also highlighted in the text in the center of the illustration. On the right is an overview of the document, showing the distribution of topics in that document.

15.2 Mathematical techniques

The most common topic modeling algorithm is *latent Dirichlet allocation*, LDA. LDA is considered to be a generative probabilistic model with both observed and hidden variables. The observed variables are the words, and the hidden variables are the topics. The observed and hidden variables combine in a joint probability distribution A conditional distribution of the hidden variables (topics), given the observed words is called the posterior distribution.

$$p(\beta_{1:k}, \theta_{1:D}, z_{1:D} | w_{1:D}) = \frac{p(\beta_{1:k}, \theta_{1:D}, z_{1:D}, w_{1:D})}{p(w_{1:D})} \tag{15.1}$$

where betas are topics 1:K, thetas are topic proportions for each topic in each document, z represents topic assignment for a given word in a document, and the observed words are w.

[1] http://www.cs.columbia.edu/ blei/papers/Blei2012.pdf

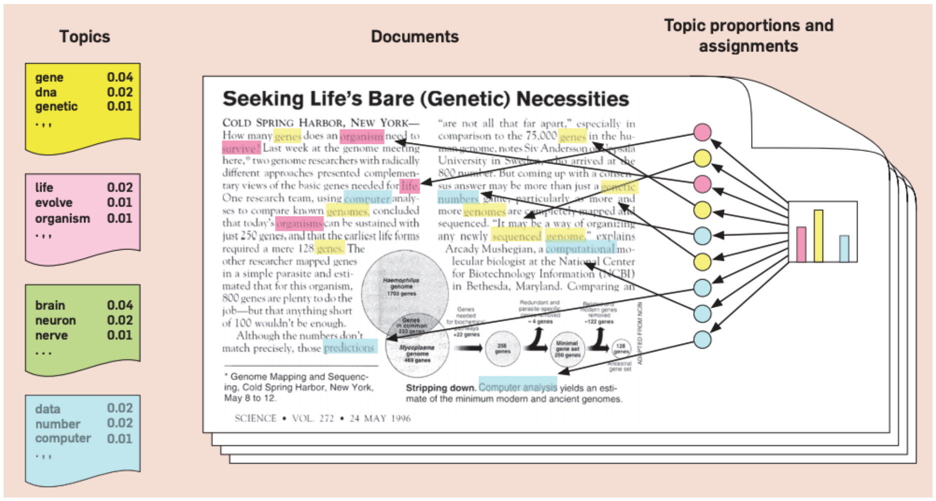


Figure 15.1: Topic Modeling

Computing the distribution for the variables in Equation 15.1 would be intractable, so a sampling technique is used. One such technique is Gibbs sampling, which is a Monte Carlo Markov Chain (MCMC) technique that starts with the variables at random values. Then, iteratively holding all variables constant but one, the algorithm repeatedly samples the data to get the best estimate of that variable. This process is repeated for each variable over hundreds or thousands of iterations, until convergence.

An alternative to an explanation of LDA with formulas is a graphical model, shown in Figure 15.2, from the same Blei paper. The shaded node is the observed data, the words. Unshaded nodes are hidden variables. The 'plate' representation indicates replication. Plate N represents collections of words within documents. Plate D represents the collection of documents in the corpus. Plate K represents the topics. The alpha hyperparameter of the LDA model represents document-topic density. The higher the alpha, the higher the number of topics assigned to each document. The eta hyperparameter represents topic-word density. The higher the eta, the more words per topic.

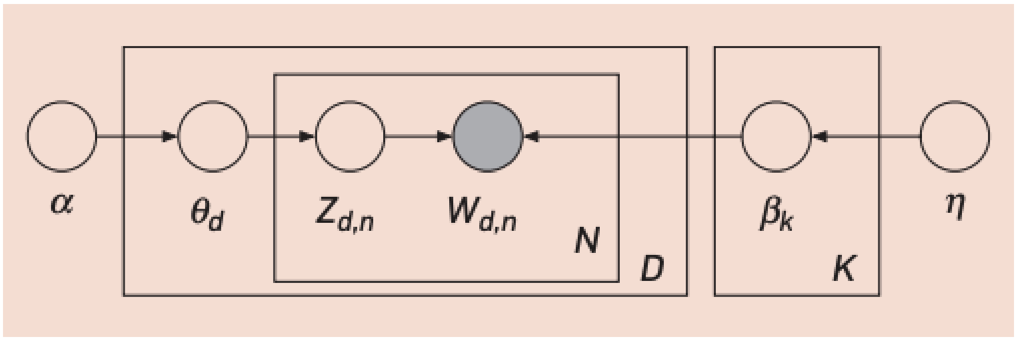


Figure 15.2: Topic Modeling Graphical Model

15.3 LDA tips

To get the best results from LDA, data should be preprocessed. Generally, lowercasing, removing punctuation and numbers is a good first step. Stop word removal is very important, and standard sets of stopwords may not be sufficient to capture common words that should not be part of a topic. Customizing the stopword list is a common preprocessing step. Lemmatizing is also a good idea because it reduces plurals to the singular form. Other approaches that are worth trying are to use only nouns and adjectives. Using limited parts of speech is fine because LDA uses a bag-of-words approach.

The number of topics is chosen beforehand, and the alpha and eta parameters can use default settings in some implementations, and fine-tuned later. Often, getting good results depends on multiple runs, trying different numbers of topics and fine tuning the hyperparameters if necessary.

The final tip is that LDA is hungry for data. The example later in this chapter uses a tiny, toy corpus and doesn't get very impressive results. The algorithm needs to see words co-occurring together in many instances to learn that they are related.

Visual inspection of the topics and words can help identify if a good number of topics was chosen. If the same topic word appears in many topics, then k is probably too large.

A metric that is often used to evaluate a topic model is *coherence*. There are many variations of this metric but all measure how much words in the topic tend to occur together in documents. Most versions of the metric will range from 0 to 1. Generally, any score below .5 indicates that a good model was not found, while anything above .8 is perhaps too good to be true. Remedies for a bad score include adjusting the hyperparameters, the number of topics, or getting better data.

15.4 LDA/LSI

Another approach for topic modeling is Latent Semantic Analysis, LSA, which is often called LSI (I for indexing) in NLP applications. LSI is a dimensionality reduction technique, essential reducing similar words to indexes. This dimensionality reduction is called Singular Value Decomposition, SVD. In most cases, LSI will be faster to train but LDA can get better results. Both LDA and LSI use a bag of words input matrix. Both techniques are demonstrated in the online notebook.

15.5 Topic modeling with Gensim

Gensim is an open-source Python library that provides many NLP functions, including topic modeling. Gensim can be installed with pip/pip3. There is a Jupyter notebook in the GitHub that goes through the code below. A toy corpus of 4 texts on anatomy, geography, economics, and business law is used. The average length of the documents is a little over 400 sentences. This corpus is not large enough to get good results, it is just being used to demonstrate the code.

Prior to the code below, the texts were preprocessed by lower casing, tokenizing, removing stopwords and non-alpha tokens. Each doc is now a list of tokens and the four lists are contained in a list of lists called `preprocessed_docs`. The code block below shows the necessary import as well as the constant set up for the number of topics. Each token is mapped to an id number in a dictionary. Then the lda model is built.

The online notebook shows different ways to extract the topics from the model. The topics are not well separated. For example, topic 0 contains key words from all 4 documents: damages, inflation, islands, autonomic. To quantify how bad a topic model this is, a couple of metrics are used in the notebook, perplexity and coherence. Perplexity has not demonstrated that it correlates well with human

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

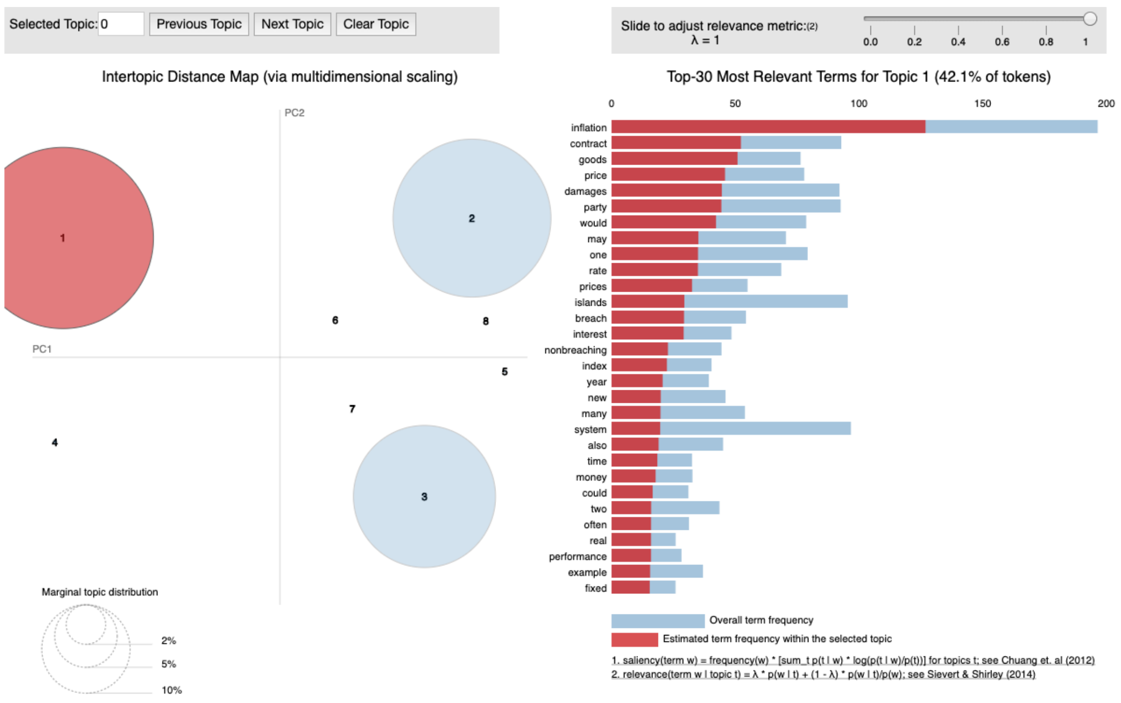
judgment. [2] A more commmonly used metric is coherence. The coherence score in the notebook was 0.365 which is a very poor score, as discussed earlier.

Code 15.5.1 — LDA. Building the topic model

```
from gensim import models, corpora
NUM_TOPICS = 8

# the dictionary maps words to id numbers
dictionary = corpora.Dictionary(preprocessed_docs)

lda_model = models.LdaModel(corpus=corpus,
        num_topics=NUM_TOPICS, id2word=dictionary)
```

15.5.1 Visualization

A nice visualization of LDA is provided by the library pyLDAvis. The online notebook demonstrates code for using this library and producing the graph below.



1. saliency(term w) = frequency(w) * [sum_t p(t | w) * log(p(t | w)/p(t))] for topics t; see Chuang et. al (2012)
2. relevance(term w | topic t) = λ * p(w | t) + (1 - λ) * p(w | t)/p(w); see Sievert & Shirley (2014)

Figure 15.3: Visualization of topic models

Numbers on the left half of the figure indicate the topics. There are significant topics shown in balloons. Good topic identification will show that the balloons are well separated in the quadrants. Balloons that are overlapping or too close together or not even showing up as in this figure, are indications that there are too many topics for this data.

[2] http://papers.nips.cc/paper/3700-reading-tea-leaves-how-humans-interpret-topic-models.pdf

When the mouse is hovered over a balloon, it changes to red (as seen above), and the important words in that topic are highlighted in red on the left side of the illustration.

On the top right of the visualization is a sliding relevance metric that determines the step distance for lambda values used when computing relevance.

15.5.2 Try again

Since this first attempt did not get good results, another attempt was made in the notebook. This time, additional stop words were removed, and the number of topics was reduced to 4. The coherence score was only slightly higher, 0.43, which is still not impressive. To get better results, more documents would be needed.

In the online notebook, the LSI model got a higher coherence score, 0.59, which is not great but better than the LDA. It seems that on this tiny corpus, LSI performed better than LDA.

15.6 Summary

Topic Modeling has received a lot of interest in recent years, yet, getting good results can be a bit of a mysterious process. An online notebook (Exploring ACL2020abstracts) uses a set of abstracts from accepted papers at the ACL 2020 conference to (1) extract topics, and (2) extract words with high tf-idf values, using sklearn.

15.7 Practice

Exercise 15.1 Exploring Topic Modeling.

- Search kaggle.com for challenges involving topic modeling
- Select a top rated notebook and replicate their results
- Try different techniques and parameters to see if you can improve their results

■

15.7.1 Further Reading

A prominent research in topic models is David M. Blei, now at Columbia University. His topic modeling page `http://www.cs.columbia.edu/~blei/topicmodeling.html` contains links to introductory materials, video talks, and source code for topic modeling.

Coherence measures the similarity of top-ranking words in topics using pointwise mutual information or other metrics. The more similar the top words are in a topic, the better the topic. Blei and others have continued work in determining how humans evaluate topic modeling results in this technical paper: `http://users.umiacs.umd.edu/~jbg/docs/nips2009-rtl.pdf`

Google provides an NLP service with many annotations such as NER, sentiment, syntax, and categories. These categories are predefined by Google, which is less flexible that building a custom topic modeling system. `https://cloud.google.com/natural-language/#overview`

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

16. Semantics

Earlier chapters described different ways to analyze syntax, the structure of sentences. Syntax supports the semantics, or meaning. Without syntax, getting the meaning out of text is hard:

```
meaning explores the semantics of morphemes words phrases of sentences and
```

Understanding the point is easier with correct syntax:

```
Semantics explores the meaning of morphemes, words, phrases, and sentences.
```

Semantics is the content someone wishes to convey to another person verbally or in writing. Linguists study semantics at word, phrase, sentence, and text levels, from different aspects including representation, denotation, and reference. Semantics is a field that branches out into many other fields such as the philosophy of language, the study of computer language, formal logic, and semiotics, the study of sign processes. The Italian novelist and semiotician Umberto Eco viewed cultural phenomena as communication.

To this point, many NLP techniques have been explored which provide information about the meaning of text:

- SRL shallow semantic parse: who did what to whom, how, why, when, where
- The tf-idf metric identifies important words that indicate what a document is about
- NER named entity recognition identifies the entities discussed in documents
- The vector-space model reduces documents to vectors, which can then be compared for similar content
- Topic modeling identifies topics in documents

This chapter explores approaches that delve deeper into semantics: formal logic, the semantic web, and more.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Before looking at these approaches, the question arises: *How does the human brain extract meaning from sentences?* The answer has been elusive for neuroscience. However, a recent study [1] reveals some interesting insights. Researchers used fMRI (functional Magnetic Resonance Imaging), as many brain studies have done in recent years. The cognitive science researchers at the University of Rochester developed a model that learned the neural activity of specific words, that is, which parts of the brain light up when a subject reads specific words. Further, the model learned to predict brain patterns for new sentences made out of recombination of the words. It is hoped that this exploration will help researchers in the future discover more about how meaning is encoded and retrieved in the human brain.

16.1 Logical representations

Logical representation of thought dates back to the Stoic philosophers of ancient Greece. Much later, Enlightenment mathematician Gottfried Leibniz formalized symbolic logic, but his work was largely unknown to later logicians such as George Boole and Augustus De Morgan, who reinvented the work. You can take several college courses on logical representations, tracing through its evolution to modern thinkers. However, all we have room for here is a brief overview of the field, in order to discuss how it could be used with NLP.

16.1.1 Propositional logic

In formal logic, also called symbolic logic, a single sentence or thought is represented as a proposition. A given proposition such as “the grass is wet”, can be abbreviated in subsequent logic with a letter, such as G or g. The constants True and False are also useful. Propositions are represented by letters so that they can be combined in more complex statements:

- $(\neg a)$ is true iff (if and only if) a is false
- $(a \wedge b)$ is true iff a and b are both true
- $(a \vee b)$ is true iff a or b or both are true
- $(\neg a \rightarrow b)$ is true iff b is true when a is false
- $(a \leftrightarrow b)$ is true iff a and b are both true or both false

Inference rules can be used to test validity of more complex logic. Here are two simple examples:

- $(P1 \wedge P2 \wedge P3 \cdots \vDash Pi)$ and-elimination
- $(P1 \vDash P1 \vee P2 \cdots)$ or-introduction

You may have had experience building proofs using propositional logic in a college course, and you may have gotten a headache, or you may have loved it. Most people have a binary reaction to proofs with propositional logic.

16.1.2 First-order logic

A limitation of propositional logic is that you can’t express truths about individual entities, and you can’t express generalizations. FOL (first-order logic), also called predicate calculus, is a more expressive logic than propositional logic. FOL can represent things, properties, and relations. However, FOL cannot easily represent categories, time, or events.

Constants, or atoms, can be used to represent entities, such as jane_smith, or texas. A predicate is an assertion. To assert that Jane is a Professor: `is-professor(jane_smith)`. To assert that Jane enjoys teaching: `enjoys(jane_smith, teaching)`. Predicates can be true or false. The *arity* of a

[1] https://blogs.scientificamerican.com/guest-blog/how-the-brain-decodes-sentences/

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

proposition is its number of arguments. This is analgous to the way that verbs can require 0, 1, or 2 objects.

Variables can represent unknown, or hypothetical constants, or all possible entities in a category. The universal quantifier $\forall$ means all entities represented by the variable. The existential quantifier $\exists$ indicates that there is at least one instance of the variable for which the statement is true. Examples:

- $\forall cow(X) \rightarrow moos(X)$ all cows moo
- $\exists cow(X) \rightarrow moos(X)$ there is at least one cow that moos

To formalize human sentences into FOL involves:

- Convert verbs to predicates
- Convert nouns and adjectives to constants
- Convert category concepts into variables

An NLP application for FOL could be an inference engine for Question Answering systems. Human language sentences can be translated into a first-order logic statement. The statement will be negated and added to the list of true statements. If no resolution is found with the combination of propositions, the statement must be true.

Implementing such a system often uses logic programming languages like Prolog. In a large knowledge base, the branching factor could be immense. Further, these systems are hard to scale up because the facts and inference rules generally have to be hand-crafted.

16.1.3 Lambda notation

An addition to FOL, proposed by Alonzo Church and other logicians, is lambda notation, which helps abstract FOL. The form is as follows:

$$(\lambda y.x(y,z))(ab) \tag{16.1}$$

The lambda-variable part is called the head. The dot separates the head variable(s) from the body x(yz). The head.body form the lambda function. The expression after the lambda function is fed into the function. In the example above, the result of the lambda function applied to (ab) would be: x(abz)

Another example:

$\lambda y.Mother-of(Jane,y)(Michael)$

resolves to:

$Mother-of(Jane,Michael)$

16.2 Semantic networks

Rules-based systems are often slow, can't handle exceptions, and can't handle probabilistic information. For example, if an entity is a bird, it probably flies, unless it's a penguin.

An alternative to having a knowledge base of facts, is to have a network of concepts, where links between concepts indicate relations. The development of the modern semantic web is often attributed to Ross Quillian. The WordNet system we used earlier in the course is an example of a semantic network. There is some indication that humans store related concepts in neural connections.

Consider the following propositions, and their representation in a semantic network:

```
mother_of(sue, john)
wife_of(sue, max)
```

```
age(sue, 34)
age(john, 5)
```

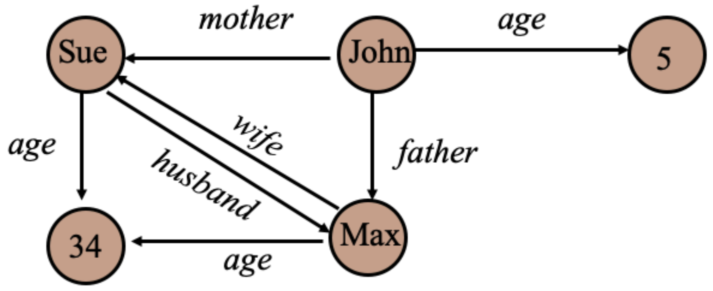


Figure 16.1: Semantic network

Reification is the process of representing an abstract concept as an object in a model. For example, a 'give' event can be represented as a relation between the giver, the recipient, and the object given. Common relations include:

- instance: Mary is an instance of a woman
- isa: Mary isa human (more general concept than woman)
- haspart: Mary has a face

Semantic networks implement inheritance by following isa links, and also multiple inheritance. Experiment with the Visual Thesaurus `https://www.visualthesaurus.com/` to see an example of a semantic network.

The technique of *spreading activation* is a search technique to find related concepts. As the activation spreads from node to node in the network, a decay factor is introduced to limit spreading to unrelated nodes. Further, nodes could be weighted to indicate similarity or distinction.

16.2.1 Ontologies

An ontology is a hierarchical vocabulary, usually restricted to a specific domain. Like a semantic network, an ontology typically arranges concepts in a hierarchy with labeled relations. The Web Ontology Language (OWL) can be used to specify taxonomies and structured knowledge networks. OWL languages (there are several variations) are built upon W3C XML standards for RDF (Resource Description Framework) objects.

There are numerous OWL system implementations that include inference engines. These systems can be used to customize an ontology for a specific domain, such as medicine. In addition, many start-ups are getting to this space, such as Cycorp `https://www.cyc.com/`

16.3 Current research

Every two years, ACL hosts an International Conference on Computational Semantics (IWCS). Common research topics include formal semantics, ontologies, annotation and representation schemes, improving NLP techniques, and much more.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

V

Part Five: Machine Learning Approaches

Part Five

Part Five gives an introduction to traditional machine learning approaches in NLP. Chapter 17 provides an overview of the field of machine learning. Chapter 18 is a tutorial on the most commonly used Python libraries for machine learning. Chapter 19 covers ways to represent text numerically for machine learning. Chapters 20 and 21 introduce Naive Bayes and Logistic regression for text classification. Chapter 22 provides an introduction to neural networks.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

17. Introduction to Machine Learning

This chapter provides an overview of the field of machine learning, particularly as it applies to natural language processing. The field of machine learning has a wide array of algorithms that can learn patterns from data. In the past few decades, machine learning approaches have outperformed traditional rules-based NLP approaches in many tasks. The next few chapters will explore many commonly used algorithms in NLP using SciKit-Learn. A later part of the book explores deep learning with Keras.

Figure 17.1 shows fields related to machine learning. Machine learning would not be possible without the fields surrounding it in the figure. Statistics and probability form the mathematical foundations of many of the algorithms we will learn in this book. AI and computer science pushed the frontiers of what computers could do, which made machine learning possible.

17.1 Machine Learning

We will use *machine learning* as an umbrella term for many closely related terms. Some statisticians call machine learning *statistical learning*. The field of *data science* and the task of *data mining* often involve machine learning techniques, coupled with more data exploration and analysis.

Machine learning has received varied definitions as the field developed. This book proposes the following definition:

Definition 17.1.1 — Machine Learning. Machine learning trains computers to accurately recognize patterns in data for purposes of data analysis, prediction, and/or action selection by autonomous agents.

The key words in this definition are: data, patterns, predictions, and actions, along with the caveat: accurate. Let's examine these in detail.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

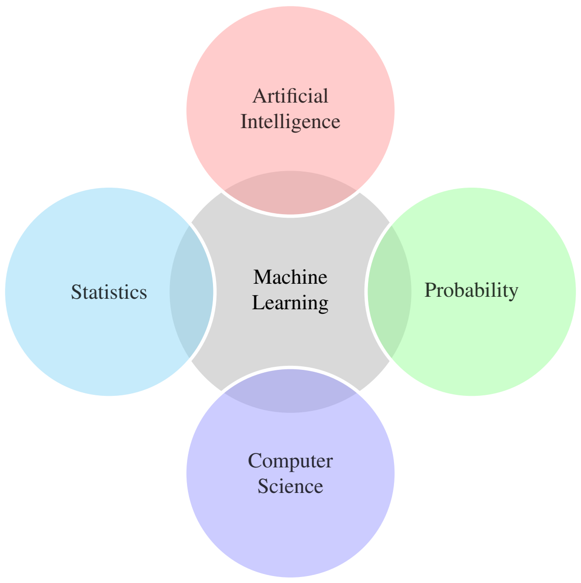


Figure 17.1: Fields Related to Machine Learning

17.1.1 Data

Nothing can be learned without data. The data and what we wish to learn from the data go hand in hand. For many learning scenarios, data takes the form of a table of values where each row represents one data example, and columns represent attributes or features of the examples. One learning scenario, clustering, seeks to group like instances. Another learning scenario, supervised learning, seeks to learn about one feature based on combinations of the other features. Data can take other forms besides tables, such as a set of actions weighted by features that represent the current environment.

When gathering data or using data from other sources, ethical considerations must always guide our actions. Who owns the data? Who are the subjects of the data? Has the data been anonymized? Did the subjects give consent for the use of the data? How will the analysis of this data be used? How might it impact the subjects in the data as well as the larger community?

17.1.2 Patterns

The best *general* pattern recognition machine is the human brain, but computers can in many cases beat human performance on narrowly defined tasks. The ability to recognize patterns in data enables algorithms to learn things like whether someone is a good credit risk, whether two people might be compatible, whether the object outlined in sensors is a human or a dark spot on the pavement.

When beginning a real-world machine learning project, how do you know what to look for? From raw data, organized data must be built. Once the data is organized, decisions must be made about what could be learned and what is important to learn from the data. These decisions often need to be in concert with domain experts and/or the owners and users of the data who wish to learn from it.

17.1.3 Predictions from Data

Learning patterns in data enables us to predict outcomes on future data – data the algorithm has never before seen. Predictions may simply involve finding similar instances in the data, or predicting a target value which may be quantitative or qualitative.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

In the examples in this book, generally we train algorithms on a portion of the data and use the remaining data to test and evaluate how well the trained model can perform on previously unseen data. This is a common situation in machine learning, sometimes called batch learning because the data is fed into the algorithm in one batch. There are other approaches, however, such as online learning in which the algorithm is continually learning from newly available data and being evaluated in real time. Online learning techniques can also be used when the available data is too large to be stored in memory. An alternate approach to handle big data is to do parallel distributed machine learning, often in the cloud with specialized software.

17.1.4 Accuracy

Predictions must be accurate or they are not predictions but random guesses. Typically, we want our predictions to beat some predetermined baseline approach. For example if 99% of the observations in a credit data set did not default on their loans, we want to beat a simple baseline that always guesses "not default" and has 99% accuracy. Machine learning makes use of many measurement techniques to gauge accuracy and evaluate performance of the algorithms. Many of these metrics are used to evaluate the training model itself and others will be used to evaluate performance on a held-out test set.

17.1.5 Actions

Every day more autonomous agents enter our lives, from smart thermostats, to automated assistants, to self-driving vehicles. These agents take actions based on what they have learned, and most continue to learn over time, usually by uploading data to a central learning repository. Some actions taken by autonomous agents will be controversial in the coming decades as ethical and legal issues evolve in response to humans co-existing with autonomous agents. The big players in AI are at the forefront of autonomous agents because they have the resources, expertise, and data to pursue big projects. In the future, we expect the development of autonomous agents to be available to more developers.

17.2 Machine Learning Scenarios

There are scores of machine learning algorithms with countless variations each. This book describes a few common algorithms, while providing a foundation to learn more algorithms. There are many ways to classify machine learning algorithms, and not all algorithms fit neatly into our categories but the following general categorizations serve as a helpful overview of the field.

17.2.1 Informative v. Active

Most machine learning algorithms covered in this book are **informative**. They provide information to us in the form of data analysis or prediction. These informative algorithms input data observations and output a model of the data that can then be used to predict outcomes for new data fed into the model. In contrast, the field of Reinforcement Learning teaches **active** agents to identify optimal actions given the current environment and what has been learned in past experience. The input to these algorithms for initial training comes in the form of data but some agents may continue to learn with sensors and other input methods that let them learn from the environment.

17.2.2 Supervised v. unsupervised learning

Informative algorithms are of two main types. The term **supervised learning** refers to scenarios where each data instance has a label. This label is used to train the algorithm so that labels can be predicted

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

for future data items. The term **unsupervised learning** refers to scenarios where data does not have labels and the goal is simply to learn more about the data.

17.2.3 Regression v. classification

Supervised learning algorithms fall into two major groups. In **regression**, our target (the label) is a real-numbered **quantitative** value, like trying to predict the market value of a home given its square footage and other data. In **classification**, our target is **qualitative**, a class, like predicting if a borrower is a good credit risk or not, given their income, outstanding credit balance, and other predictors.

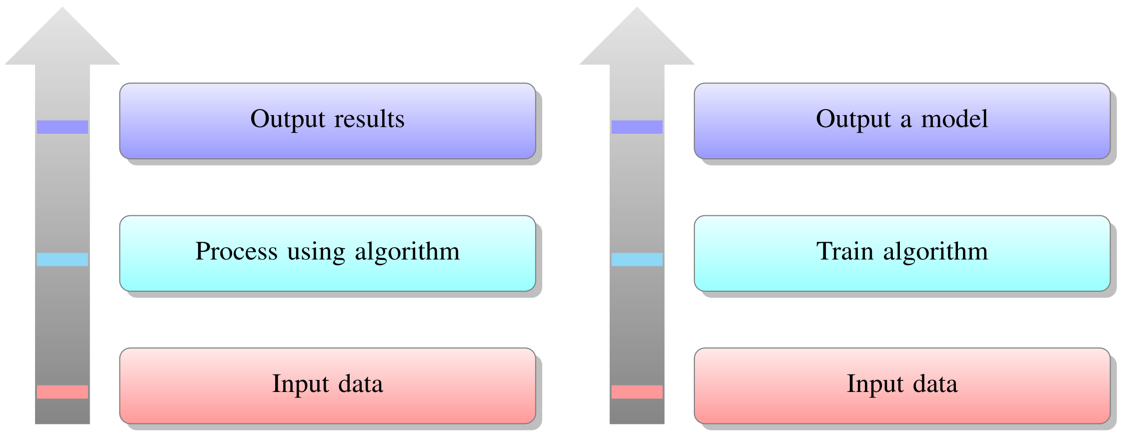


Figure 17.2: Traditional Programming (Left) v. Machine Learning (Right)

17.3 Machine learning v. traditional algorithms

Machine learning algorithms are different from traditional algorithms encountered in computer programming. In Figure 17.2 we see the traditional computer programming paradigm on the left: data is fed into code that processes it and outputs the results of the processing. In the same figure, we see the machine learning paradigm on the right: we feed data into an algorithm which builds and outputs a model of the data. In traditional programming, all knowledge is explicitly encoded in the algorithm by programmers using code statements, loops, and conditional statements. Therefore, knowledge must be known beforehand. In machine learning, knowledge is inferred from data. Knowledge is discovered.

Why do we need machine learning? Can't we just explicitly code algorithms for problems? There are two typical situations in which traditional programming cannot be used to solve problems. The first type is when it is not possible to encode all the rules needed to solve a problem. How would you encode rules for recognizing faces in photos? We don't even know the rules we use in our minds to recognize faces so it would not be possible to encode rules. However, we can train computers to recognize key edges and regions of photos that are likely to be faces. The second type of situation in which traditional programming cannot be used to solve a problem is when the scale of the problem is too large. If a company has huge amounts of customer data, it would take millions of human hours to find useful patterns in the data that could be then extracted programmatically. Machine learning algorithms can find patterns quickly in large amounts of data.

As we go through the material in this handbook, you will learn several machine learning algorithms. These algorithms typically have statistical and probabilistic foundations which we will explore. How-

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

ever, beyond the theory and technique, machine learning is also a craft as well as a science. The craft aspect of machine learning is gained through experience.

17.4 Terminology

Machine learning grew out of statistics and probability, computer science, as well as other fields. For this reason there are often multiple terms for the same thing. Let's start with names for data. The table below contains a sample data set (with headings). The table has 3 rows of data. Each row is a sample data point, also called an **example**, **instance**, or an **observation**. Each column in the table is an **attribute**, also called a **feature** or a **predictor**. The table shows 4 features: GPA, average number of hours studied per week, SAT score, and classification. The first 3 are **quantitative**, or numeric, features while Class is a **qualitative** feature because it can only take on one of a finite set of values. Qualitative features are also called **categorical data**.

GPA	Hours	SAT	Class
3.2	15	1450	Junior
3.8	21	1420	Sophomore
2.5	9	1367	Freshman

Table 17.1: Student GPA, Average Hours Studied/Week, SAT, Class

If we want to learn GPA as a function of the other 3 features, we say that GPA is our **target**, or **response**, variable while the other 3 are **features** or **predictors**. If we want to predict SAT then SAT would be our target and all other columns our predictors.

17.5 Probability Foundations

This book assumes that readers have had a prior course on probability but we will review some key concepts here. Random variables, often denoted by capital letters such as X, can be discrete or continuous. The probability of two variables X and Y is called the *joint* distribution, determined jointly by X and Y, $P(X,Y)$. The *conditional* distribution of P(X|Y) is given by:

$$P(X|Y) = \frac{P(X,Y)}{P(Y)} \tag{17.1}$$

Two important probability rules are the product rule and the sum rule. The product rule says that the joint probability of A and B can be calculated by multiplying the conditional probability of A given B by the probabiity of B.

$$p(A,B) = p(A|B)p(B) \tag{17.2}$$

The sum rule says that we can calculate the probability of A by finding the joint probability with B and summing over all possible values of b.

$$p(A) = \sum_b p(A,B) = \sum_b p(A|B=b)P(B=b) \tag{17.3}$$

The chain rule lets us take the joint probability of many variables as follows:

$$p(X_{1:D}) = p(X_1)p(X_2|X_1)p(X_3|X_2,X_1)...p(X_D|X_{1:D-1}) \tag{17.4}$$

The expectation of a random variable is also known as the mean, or first moment. The expectation of a discrete random variable is:

$$E(X) = \sum_i X_i P(X_i) \tag{17.5}$$

So the expected value of a fair die is 3.5:

$$E(X) = 1\times\frac{1}{6}+2\times\frac{1}{6}+3\times\frac{1}{6}+4\times\frac{1}{6}+5\times\frac{1}{6}+6\times\frac{1}{6} = 3.5 \tag{17.6}$$

The variance of a distribution is also called its second moment, and is represented by σ^2. When we take its square root, we have the standard deviation.

$$\sigma^2 = E[(X-\mu)^2] \tag{17.7}$$

The log trick is often used when multiplying probabilities to prevent underflow and possibly multiplying by 0. Instead of multiplying the probabilities, the log trick says to add the log of the probabilities.

17.6 Probability Distributions

There are a few probability distributions that occur frequently in machine approaches so it would be a good idea to review them here. Many of these are discussed also in context of their conjugate prior. Conjugate distributions are in the same family. The distributions are illustrated with R code. R is a statistical language that makes creating and visualizing probability distributions easy. The R code is provided but feel free to skip over the code blocks in the rest of the chapter and focus on the illustrations.

17.6.1 Bernouli, Binomial, and Beta Distributions

These distributions concern binary variables representing such things as the flip of a coin, wins and losses, and so forth. Our example will be shooting baskets for practice, where x=1 means that we made a basket and x=0 means that we missed. Let's say that I am shooting hoops for the first time and I made 2 baskets out of 10 tries. Given this data, my probability of making a basket is 0.2. The Bernouli distribution describes binary outcomes like this example. The Bernouli distribution has a parameter, μ, which specifies the average probability of the positive class.

$$Bernoulli(x|\mu) = \mu^x(1-\mu)^{1-x} \tag{17.8}$$

This gives us the probability that x is 1: $p(x=1) = 0.2^1 * .8^0 = 0.2$. And the probability that x is 0: $p(x=0) = 0.2^0 * .8^1 = 0.8$.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

The Bernoulli distribution is a special case of the binomial distribution in which the number of trials, N = 1. Now suppose we run our Bernouli trial N=100 times, that is, I shoot 100 baskets. Let X be the random variable which represents the number of baskets made. The binomial distribution of X where I made k baskets in N trials has the following probability mass function (pmf):

$$Binomial(k|N,\mu) = \binom{N}{k} \mu^k (1-\mu)^{N-k} \tag{17.9}$$

Let's let k=20 for our 100 trials. Will the outcome of the binomial be 0.2?

$$Binomial(20|100, 0.2) = \binom{100}{20} 0.2^{20} (1-0.2)^{80} = 0.09930021 \tag{17.10}$$

No, it is not because there are many ways we can get 20 baskets out of 100 trials, 100 choose 20, to be exact. You can get out your calculator to confirm that or use R command `dbinom(20, 100, 0.2)`.

What is the expected mean of our 100 trials? Our expected value will be $N\mu$ which in our case is $100 * 0.2 = 20$. Let's derive these values using R. First we make a vector of possible values for k, the number of baskets made. We could make anywhere from 0 to 100 baskets. Then we multiply each k by its probability and add them together following Equation 6.7 above for the mean of a discrete distribution. E is 20, as we expected. The plot in Figure 17.3 shows the expected value at the center with the variance, calculated as $N\mu(1-\mu)$, which is 16 in our example. If you `sum(dbinom(k, 100, 0.2))` you get 1.0 of course because this represents the total probability space.

Code 17.6.1 — Basketball. A Binomial Distribution.

```
k <- 0:100  # possible number of baskets for 100 tries
E <- sum(k * dbinom(k, 100, 0.2))
v <- 100 * .2 * .8
plot(k, dbinom(k, 100, 0.2))
```

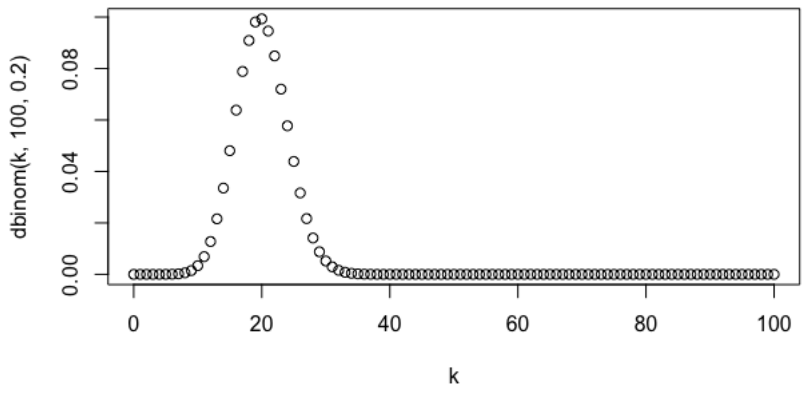


Figure 17.3: Binomial Distribution for 0.2

Now suppose that I got really lucky when I shot my first 3 hoops and made all 3 baskets. This gives me $\mu = 1.0$. It's unlikely I can keep this probability over the long haul. In fact, small sample sizes serve poorly as prior estimates of probabilities. Instead of a prior μ we really need a prior distribution over μ. We want this prior distribution to be a *conjugate* of our binomial distribution, meaning that we want it to be proportional to $\mu^x(1-\mu)^{1-x}$. The beta distribution is a good choice:

$$Beta(\mu|a,b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\mu^{a-1}(1-\mu)^{b-1} \tag{17.11}$$

In the above equation, the first term with the gammas serves as a normalizing constant to make sure that the total probability integrates to 1. The gamma function is commonly used in probability, and is an extension of the factorial function to the real numbers: $\Gamma(n) = (n-1)!$ and is also extended to complex numbers. The parameters a and b in our example will be the number of baskets made and missed, respectively. Beta distributions are commonly used in Bayesian approaches to represent prior knowledge of a parameter. The gamma function is defined as follows for positive real numbers or complex numbers with a positive real portion:

$$\Gamma(x) \equiv \int_0^\infty u^{x-1}e^{-u}du \tag{17.12}$$

Let's look at the beta distribution for our example in R. We use the rbeta() function to create 100 random beta samples with shape parameters 20 and 80. Then we plot this curve as the black line in Figure 17.4. Now suppose I take 15 more shots and make 10 of them. This will make a=30 and b=85. This updated curve is shown in red (grey) in Figure 17.4. The code below shows how to create the random beta samples with `rbeta(n, shape1, shape2)`. What will x look like? it is a vector of 100 random numbers drawn from a beta distribution with parameters a=20 and b=80. The mean will be 0.2 with the min around 0.1 and the max around 0.3. The code then draws the original curve in black and the updated curve in red. The `par(new=TRUE))` is used when you want to plot over an existing plot.

Code 17.6.2 — Basketball. A Beta Distribution.

```
x <- rbeta(100, 20, 80)
curve(dbeta(x, 20, 80), xlab=" ", ylab=" ", xlim=c(0.1,0.6),
     ylim=c(0,10))
par(new=TRUE)
curve(dbeta(x, 30, 85), xlab=" ", ylab=" ", xlim=c(0.1,0.6),
    ylim=c(0,10), col="red")
```

In the code and plot above, we updated the original black curve by adding baskets to a and misses to b. Our new probability given our data will be:

$$p(x=1|D) = \frac{a+m}{a+b+m+l} \tag{17.13}$$

where m represents the number of new baskets and l represents the number of new losses.

The equation above is a Bayesian estimate. In contrast, note that the MLE estimate is simply m/N. As m and l approach infinity, the Bayesian estimate converges to the MLE. For a finite data set, the posterior probability with be between the prior and the MLE.

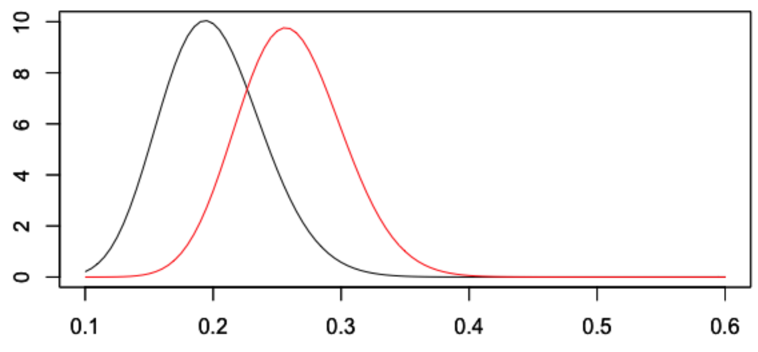


Figure 17.4: Beta Distribution for a=20, b=80

17.6.2 Multinomial and Dirichlet Distributions

We can extend the binomial distribution to the case where we have variables that are not binary but can take on more than 2 values. This is a multinomial distribution. The probability mass function of a multinomial distribution is:

$$Multinomial(m_1, m_2, ..., m_k | N, \mu) = \left(\frac{N}{m_1!m_2!...m_k!} \right) \prod_{k=1}^{K} \mu_k^{m_k} \tag{17.14}$$

Where k indexes over the number of classes, K, and each of the m_i represent the probability of that class, with the sum of all $m_i = 1$.

The iris data is an example of a multinomial distribution. There are 3 classes, and the data set has 50 examples of each class, an even distribution. If we want to put the 150 flowers in 3 boxes (classes) with even probability of being in each class, we could use the following R command.

Code 17.6.3 — Multinomial. Iris Example

```
rmultinom(n=10, size=150, prob=c(1/3, 1/3, 1/3))
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]   56   44   51   55   54   47   49   60   61    42
[2,]   36   48   54   44   42   46   58   50   40    61
[3,]   58   58   45   51   54   57   43   40   49    47
```

The output above shows us 10 vectors, left to right, because we said "n=10". Our size is 150 for each of our iris flowers, and they have an even distribution. What we see in each column are distributions of the 150 flowers. Each column sums to 150. What we see in each row are the number of flowers in each box (class). The mean values for the 3 classes are 54, 50.1, and 45.9.

If we selected 6 flowers at random, what is the probability that there will be 1 flower from class 1, 2 flowers from class 2, and 3 flowers from class 3?

Code 17.6.4 — Multinomial. Use dmultinom() for probabilities.

```
dmultinom(x=c(1,2,3), prob=c(1/3, 1/3, 1/3))
[1] 0.08230453
# check:
factorial(6)/(factorial(3)*factorial(2)*factorial(1))*
    0.333333^1*0.333333^2*0.333333^3
[1] 0.08230403
```

The Dirichlet distribution is the conjugate prior of the multinomial distribution. The Dirichlet distribution has k parameters, α, one for each class. So instead of X being 0 or 1, it can take on k values. In the following, α_0 is the sum of all alphas. The Dirichlet distribution:

$$Dir(\mu|\alpha) = \frac{\Gamma(\alpha_0)}{\Gamma(\alpha_1)...\Gamma(\alpha_k)} \prod_{k=1}^{K} \mu_k^{\alpha_{k-1}} \tag{17.15}$$

Consider a magic bag containing balls of K=3 colors: red, blue, yellow. For each of N draws, you place the ball back in the bag with an *additional* ball of the same color. As N approaches infinity, the colored balls in the magically expanded bag will be $Dir(\alpha_1, \alpha_2, \alpha_3)$.

You can think of the Dirichlet distribution as a multinomial factory.

Code 17.6.5 — Dirichlet. Output Distribution.

```
library(MCMCpack) # for function rdirichlet()
m <- rdirichlet(10, c(1, 1, 1))
m
             [,1]      [,2]       [,3]
 [1,] 0.015740801 0.3900641 0.59419507
 [2,] 0.295649733 0.3622780 0.34207224
 [3,] 0.464984547 0.4516325 0.08338300
 [4,] 0.365099590 0.3074731 0.32742729
 [5,] 0.065993901 0.2832624 0.65074371
 [6,] 0.252786635 0.6786473 0.06856608
 [7,] 0.049175200 0.4904748 0.46034997
 [8,] 0.297815089 0.2121868 0.48999815
 [9,] 0.005201826 0.3076536 0.68714457
[10,] 0.326711959 0.4160060 0.25728208
mean(m[,1])
[1] 0.2139159
> mean(m[,2])
[1] 0.3899679
> mean(m[,3])
[1] 0.3961162
```

We asked for 10 distributions with our alphas all equal to 1. The sum of every row, which is every distribution, is 1.0. The mean of the columns for these 10 examples are 0.2, 0.38, and 0.39. When run with 1000 examples the means were 0.34, 0.33 and 0.32.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

17.6.3 Gaussian

The Gaussian or normal distribution is used for quantitative variables. Two parameters define its shape: the mean μ and the variance σ^2. The probability density function is:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma}} exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{17.16}$$

Let's plot a few Gaussians to see how the mean and variance influence the shape. All 3 distributions are generated with the dnorm() function, and all have a mean of 0. Different means would shift the curves right or left. The three curves have different standard deviations.

Code 17.6.6 — Gaussians. Normal distributions.

```
curve( dnorm(x,0,1), xlim=c(-4,4), ylim=c(0,1) )
curve( dnorm(x,0,2), add=T, col='blue' )
curve( dnorm(x,0,.5), add=T, col='orange' )
```

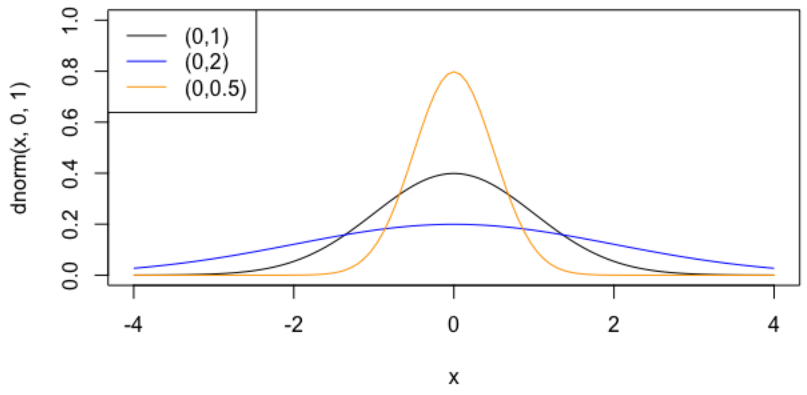


Figure 17.5: Gaussian Distributions

The rnorm() function in R generates random numbers, following a normal distribution. The d in the dnorm() function is for density, as in pdf, probability density function. The dnorm() function returns the pdf for the normal distribution specified by the parameters.

The pdf of the Gaussian above in Equation 5.18 is for a single variable x. The Gaussian can be extended to a D-dimensional vector **x** in which case it is called a multivariate Gaussian and has the pdf shown below where μ is now a vector of means, Σ is a DxD covariance matrix, $|\Sigma|$ is the determinant.

$$f(x) = \frac{1}{\sqrt[D/2]{2\pi}\sqrt{|\Sigma|}} exp\left(-\frac{(x-\mu)^T(x-\mu)}{2\Sigma}\right) \tag{17.17}$$

17.7 Probability distributions in NLP

What do statistics and probability have to do with text? Actually, everything throughout the book has been implicitly exploring distributions and text. Word frequencies and related measures such as tf-idf will follow a probability distribution The ngram approach is a probabilistic approach to solving NLP problems. Topic modeling uses a variety of statistical approaches. However, the machine learning approaches discussed in the next few chapters explore more purely statistical approaches to solving NLP problems.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

18. NumPy, pandas, Scikit-Learn, Seaborn

This chapter is an introduction to the major packages used for machine learning in Python. Examples will look at numeric data used in machine learning in general, not specifically NLP machine learning. The next chapter will show how to convert text to numeric data so that these tools can be used. Here are the most commonly used libraries in Python machine learning:

- NumPy - numerical array processing
- pandas - data manipulation
- Scikit-Learn - machine learning algorithms
- Seaborn - plotting

Later chapters will look at machine learning algorithms in the Keras package.

The package manager pip is installed with python. After installation, pip is used to install or upgrade packages. As of this writing, the latest version of pip is 19. You can read the documentation for more information. All of the following commands are run from the command line, not from within Python.

Code 18.0.1 — pip. Using pip package manager.

```
pip install x            # install package x
pip install x==2.01      # install x version 2.01
pip install x>=2.01      # install at least x version 2.01
pip install --upgrade x  # update package x
pip uninstall x           # uninstall package x
pip --version            # find your version
```

If you have a Mac or other computer with Python 2 installed, your pip for Python 3 will be pip3 instead of just pip. If you want to upgrade your pip, type the following at the command line:

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

```
python -m pip install --upgrade pip
```

First, make sure your pip is updated. Then use the commands above to install/upgrade numpy, pandas, sklearn, and seaborn.

18.1 NumPy

NumPy is a library for scientific computing. NumPy is designed to work efficiently with multidimensional arrays. The key data structure in NumPy is the array, which is a multi-dimensional object where elements must be of the same type. NumPy arrays are more efficient in terms of storage and computation that native Python lists. The *rank* of the array is the number of dimensions and the *shape* is specified by a tuple of integers representing the length of each dimension. Let's see how to create a numpy array from a Python list.

Code 18.1.1 — numpy. Create a 1d numpy array.

```
import numpy as np
a = np.array([1,2,3,4,5], float)

print(a[2])
3.0
```

In the code above, we first imported NumPy and associated it with the identifier 'np'. Then we used the `np.array()` function to create a numpy array from a Python list, making each element of type float. Notice that we index a NumPy array the same as a list. The array that was created is one-dimensional. Let's look at a two-dimensional array. It will create two rows, each a 3-column list.

Code 18.1.2 — NumPy. Create a 2d numpy array.

```
b = np.array([[1,2,3], [4,5,6]], int)

b[1,1]
5
```

NumPy arrays can be reshaped, and there are built-in functions to retrieve information about the array. The output of this code block is shown immediately below the code. The code first fills a 1d array with 0-9. Then these 10 elements are reshaped to be a 5x2 array. Notice that len() returns the length of the first axis.

Code 18.1.3 — NumPy. Shape and rank.

```
c = np.array(range(10), float)
print("c originally: ", c)
c = c.reshape(5, 2)

print("c with the new shape: \n", c)
print("The new shape is: ", c.shape)
print("Length = ", len(c))
```

```
c originally:  [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
c with the new shape:
 [[ 0.  1.]
 [ 2.  3.]
 [ 4.  5.]
 [ 6.  7.]
 [ 8.  9.]]
The new shape is:  (5, 2)
Length =  5
```

The following code block shows different ways to initialize and create arrays.

Code 18.1.4 — NumPy. Create and initialize arrays.

```
c.fill(0) # c is overwritten with zeros
d = np.zeros((2,3)) # d is a 2x3 array of zeros
```

You can use Python mathematical operators like + and - on NumPy arrays, and they will work on an element-by-element basis. There are functions that are applied to the entire array, such as sum(), mean() and min(). An example is `np.mean(c)`, which will take the mean of the elements in array c. In the code below we see examples of sum(), mean(), min() and max(). Notice in the last example that we can apply these functions to only one axis or the entire array.

Code 18.1.5 — NumPy. Operations on arrays.

```
np.sum(array1)  # sum all elements
np.mean(array1) # find the average of all elements
range = np.max(array1) - np.min(array1) # find the range
f = np.array(range(10), float).reshape(5,2)  # not working ???
print(f)
[[ 0.  1.]
 [ 2.  3.]
 [ 4.  5.]
 [ 6.  7.]
 [ 8.  9.]]
print(f.mean(axis=0)) # find mean of columns
[ 4.,  5.]
```

NumPy has random number generators as shown below. Set a seed to get reproducible results. Notice that the next-to-last line of code below that used a boolean selection criterion for indexing of the array. In the last line of code the array is sorted. The sort was performed on rows by default.

```
np.random.seed(17)
rand_array = np.random.rand(2,3)
print(rand_array)
[[ 0.6375209   0.57560289  0.03906292]
 [ 0.3578136   0.94568319  0.06004468]]
```

```
# boolean indexing
print(rand_array[rand_array>.2])
[ 0.6375209   0.57560289  0.3578136   0.94568319]

# sort
print(np.sort(rand_array))
[[ 0.03906292  0.57560289  0.6375209 ]
 [ 0.06004468  0.3578136   0.94568319]]
```

NumPy tutorials and reference can be found in the docs: `https://numpy.org/doc/`

18.2 Pandas

The package `pandas` is a powerful data analysis toolkit for structured data. A Pandas **series** handles one-dimensional data while a Pandas **data frame** handles two-dimensional data. Pandas is often used for reading in data frames such as csv files. That is demonstrated next. To follow along, download the Heart.csv data from the github. Notice that we specify which column serves as the row identifier. The head() function displays the first 5 rows by default.

Code 18.2.1 — Pandas. Read in Data.

```
import pandas as pd
df = pd.read_csv('Heart.csv', index_col='ID')
print(df.head())
```

There are many ways to access elements, as shown next. We see two ways to access the first row of the Age column.

Code 18.2.2 — Pandas. Access data elements.

```
print(df['Age'][1])
print(df.Age[1])
```

Pandas has a couple of accessors, `loc` which uses indices, and `iloc` which uses index positions. Both specify the indices in row, col order. Both of the following retrieve the same element.

Code 18.2.3 — Pandas. Accessors loc and iloc.

```
print(df.loc[1, 'Age'])
print(df.iloc[0, 0])
```

The following examples demonstrate subsetting and slicing data frames. Slicing works the same as for Python lists.

Code 18.2.4 — Pandas. Subsetting and Slicing.

```
df_new = df[['Sex', 'Age']]
print(df_new.head())
print(df['Age'][:5])
```

When a single column of a pandas data frame is selected, it will be a series by default. To force it to be a data frame, surround the selection with double brackets.

Code 18.2.5 — Pandas. Subsetting and Slicing.

```
df_not = df['Sex']     # not a data frame
df_new = df[['Sex']] # is a data frame
```

Pandas documentation can be found here: `https://pandas.pydata.org/docs/`

18.3 Scikit-Learn

The Scikit-Learn project began in 2007 and made its first public release in 2010. Scikit-Learn is an open-source project, supported by an international team of developers, releasing new versions about every 3 months. The community also has some great tutorials. Our goal here is just to see some simple examples so we can get started with the sklearn package.

In order to practice with sklearn we will look at the built-in iris dataset. The iris data is a well-known data set that has 4 flower measurements as features and one target: the species, which is one of three types of iris flower. The built-in data sets are objects that have a `.data` member variable which holds the data array which is of size `n_samples`, `n_features`. If it is a supervised learning data set it will have response variables stored in the `.target` member. In sklearn, a `Bunch` is like a dictionary with key-value pairs. First we look at some data loading and exploration on the built-in iris data set. The output of each print statement is shown immediately below that statement.

Code 18.3.1 — Scikit-Learn. Data Exploration.

```
from sklearn import datasets
iris = datasets.load_iris()
print(iris.data[:5])  # first 5 rows of data
[[ 5.1  3.5  1.4  0.2]
 [ 4.9  3.   1.4  0.2]
 [ 4.7  3.2  1.3  0.2]
 [ 4.6  3.1  1.5  0.2]
 [ 5.   3.6  1.4  0.2]]

print(iris.target[:5]) # first 5 labels
[0 0 0 0 0]

print('iris shape is ', iris.data.shape)  # get the shape of the data
iris shape is  (150, 4)

print(type(iris))
<class 'sklearn.utils.Bunch'>

print(iris.keys())
dict_keys(['data', 'feature_names', 'target', 'DESCR', 'target_names'])
```

```
print(iris.target)
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]

print(iris.target_names)
['setosa' 'versicolor' 'virginica']
```

To get set up for supervised machine learning, variable X will represent a data frame of predictor variables, and y will represent the target variable. There is a nice synergy between Pandas and Scikit-Learn so that it is easy to convert data between the libraries as shown in below.

Code 18.3.2 — Scikit-Learn. Set Up Data.

```
X = iris.data
y = iris.target

import pandas as pd
df = pd.DataFrame(X, columns=iris.feature_names)
df.head()
```

The next code block demos the k-nearest neighbors algorithm in sklearn, here used to classify iris species. The data is divided into a train and test set using an sklearn function.

Code 18.3.3 — Scikit-Learn. kNN.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.cross_validation import train_test_split  # outdated
X_train, X_test, y_train, y_test =
  train_test_split(X, y, test_size=0.3, random_state=21, stratify=y)
knn = KNeighborsClassifier(n_neighbors=7)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
knn.score(X_test, y_test)
0.9555555555555556
```

This section has just shown the basics in sklearn. The next few chapters will expand on this foundation, with many machine learning examples using sklearn. The sklearn site has excellent documentation: `https://sklearn.org/`

18.4 Seaborn

Many Python packages exist for plotting, with new packages coming out all the time. However, this chapter explores Seaborn because it is quite simple to use and produces nice-looking graphs. The online notebook gives a few examples of plotting on the iris data. The X data is converted to a data frame as shown in the last section. The plot code below assumes that the data is in data frames which will normally be the case. Converting the iris data to data frames is shown below.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

```
# load iris
iris = datasets.load_iris()
X = iris.data
y = iris.target

# convert to data frames
df = pd.DataFrame(X, columns=iris.feature_names)
df_y = pd.DataFrame(y, columns=["species"])
```

18.4.1 Plotting a quantitative array

The code block below shows how to plot the distribution of a quantitative array. The parameter kde=True draws the curve and rug=True makes the tick marks across the bottom, indicating observations. The plot is shown in Figure 18.1.

Code 18.4.1 — Seaborn. Distribution Plot

```
sb.distplot(df["petal length (cm)"], kde=True, rug=True)
```

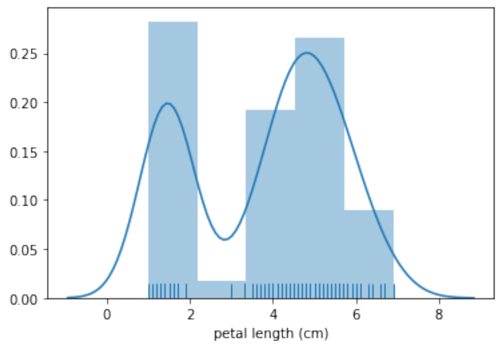


Figure 18.1: Distribution Plot

18.4.2 Plotting two quantitative arrays

The code below shows how to create a relation plot for two quantitative variables. To add more information, the color and shape of the dots varies by class of the target variable y. The plot is shown in Figure Fig 18.2.

Code 18.4.2 — Seaborn. Relation Plot

```
sb.relplot(x="petal length (cm)", y="petal width (cm)",
       data=df, hue=df_y.species, style=df_y.species)
```

18.4.3 Plotting a Categorical array

The following code block and plot show the distribution of the target, species. The distribution is even across the 3 classes.

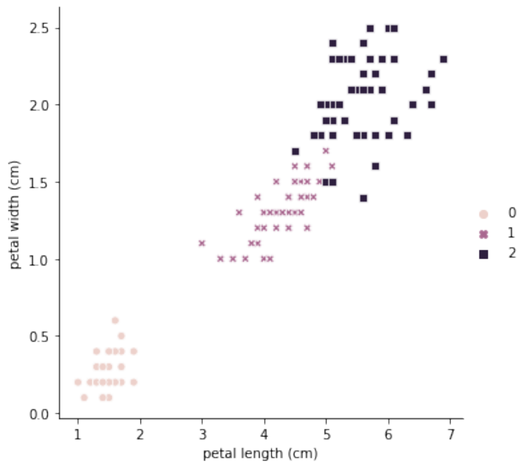


Figure 18.2: Relation Plot

Code 18.4.3 — Seaborn. Category Plot

```
sb.catplot(x="species", kind="count", data=df_y)
```

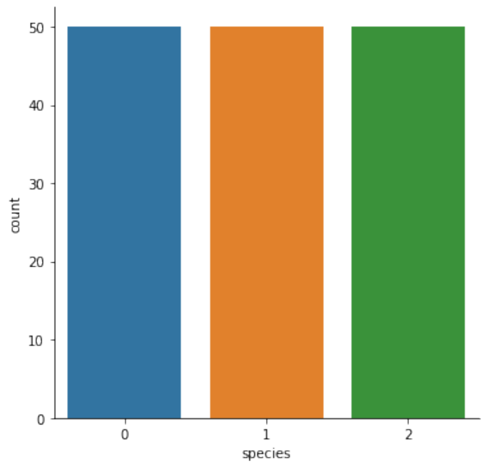


Figure 18.3: Category Plot

The following code block and figure plot a categorical value, species, on the x axis, and a quantitative array, petal length, on the y axis. The first line of code below is a big ugly and needs explanation. Normally, all of our data would be read in from a csv file into one data frame and we could easily create a plot like this with one line of code. However, we had the X and y in different data frames and they needed to be combined with pandas concat() function. When this was done with this line of code the species column contained NaN. This was because the indexes were not aligned properly in the concat, and so with the added parameters, everything joined nicely.

```
df_temp = pd.concat([df, df_y])
```

Code 18.4.4 — Seaborn.
```
df_temp = pd.concat([df.reset_index(drop=True),
      df_y.reset_index(drop=True)], axis=1)
sb.catplot(x="species", y="petal length (cm)", data=df_temp)
```

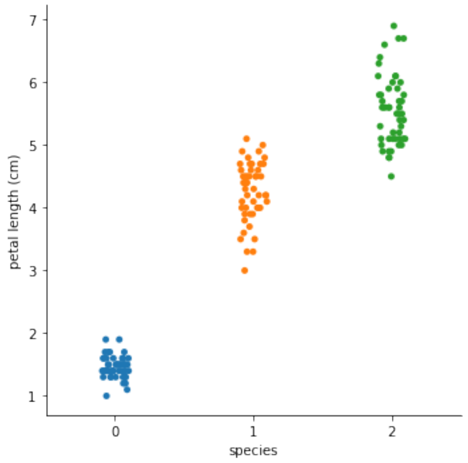


Figure 18.4: Category and Petal Length Plot

Click on the link for an excellent Seaborn cheat sheet

18.5 Summary

This chapter demonstrated how to use NumPy, pandas, sklearn, and Seaborn in machine learning problems. Familiarity with these libraries will become second nature with experience.

19. Converting Text to Numeric Data

Machine learning algorithms learn from numeric data, so raw text must be converted to numeric data. Previously we have discussed tf-idf and document vectors. The sklearn package has functions to put these together to form numeric vectors from raw text. These tf-idf vectors work well with the classic machine learning algorithms covered in the next few chapters. Approaches such as the one explained in this chapter are called *bag of words* models because word order is not considered. The words are just counted as if they were objects in a bag.

To develop a sound understanding of how these vectors are created, this chapter uses a toy corpus inspired by a classic nursery rhyme.

```
corpus = ['Mary had a little lamb.',
     'The lamb followed Mary to school one day.',
     'The lamb was white.',
     'Mary should not bring a lamb to school.',
     'Mary is a little rebel.']
```

19.1 CountVectorizer

The sklearn CountVectorizer() function:

- tokenizes the text
- assigns a unique integer id to each word in the corpus
- counts the word frequencies for each word

As shown in the code block below, we first import CountVectorizer and then create an instance of it, called 'vectorizer'. The fit function is applied to the tiny corpus.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Code 19.1.1 — CountVectorizer. Initial processing.

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
vectorizer.fit(corpus)
```

```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
           dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
           lowercase=True, max_df=1.0, max_features=None, min_df=1,
           ngram_range=(1, 1), preprocessor=None, stop_words=None,
           strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
           tokenizer=None, vocabulary=None)
```

The output above from the fit function shows the settings that were used. The sklearn documentation is the best source of information about the options, but a few are mentioned here that may be confusing:

- analyzer='word' – 'word' for word, not character ngrams
- binary=False – give counts instead of binary indicators
- ngram_range=(1,1) – only unigrams
- max_df=1.0, min_df=1 – ignore terms that have a document frequency above/below the amount; notice that the min value is an integer and the max value is a percent
- max_features==None – if a number is provided, only consider the top n terms

Most of the other options are self-explanatory, but also fully described in the documentation.

So what has CountVectorizer() done so far? Printing out the vocabulary shows that each token has a unique identifier. There was no stopword removal, so 'the' is in the dictionary. However, the one-letter word 'a' was removed by the regular expression token pattern argument in sklearn.

```
print(vectorizer.vocabulary_)
{'mary': 7, 'had': 3, 'little': 6, 'lamb': 5, 'the': 13, 'followed': 2,
'to': 14, 'school': 11, 'one': 9, 'day': 1, 'was': 15, 'white': 16,
'should': 12, 'not': 8, 'bring': 0, 'is': 4, 'rebel': 10}
```

The code block and output below shows how counts for the words are extracted. The fit_transform method gets counts for each doc. What you see in the output below the code block is a sparse document-term matrix. There is one row for each document, and one column for each of the 17 terms. To see which column aligns with which term, use the 3rd and last line of code in the block below.

Code 19.1.2 — CountVectorizer. Create a sparse document-term matrix

```
corpus_counts = vectorizer.fit_transform(corpus)
print(corpus_counts.toarray())
print('names:', vectorizer.get_feature_names())
```

```
[[0 0 0 1 0 1 1 1 0 0 0 0 0 0 0 0 0]
 [0 1 1 0 0 1 0 1 0 1 0 1 0 1 1 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 1]
 [1 0 0 0 0 1 0 1 1 0 0 1 1 0 1 0 0]
 [0 0 0 0 1 0 1 1 0 0 1 0 0 0 0 0 0]]
```

```
names: ['bring', 'day', 'followed',
'lamb', 'little', 'mary', 'one', 'rebel',
'school', 'white']
```

19.2 TfidfVectorizer

Next we compare the results above from CountVectorizer to a different vectorizer, TfidfVectorizer. Notice that many of the options are the same.

Code 19.2.1 — TfidfVectorizer. Initial processing

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
vectorizer.fit(corpus)
```

```
TfidfVectorizer(analyzer='word', binary=False, decode_error='strict',
           dtype=<class 'numpy.float64'>, encoding='utf-8',
           input='content', lowercase=True, max_df=1.0, max_features=None,
           min_df=1, ngram_range=(1, 1), norm='l2', preprocessor=None,
           smooth_idf=True, stop_words=None, strip_accents=None,
           sublinear_tf=False, token_pattern='(?u)\\b\\w\\w+\\b',
           tokenizer=None, use_idf=True, vocabulary=None)
```

The TfidfVectorizer processes terms in a similar way to CountVectorizer, but it also computes the idf for each term.

Code 19.2.2 — TfidfVectorizer. Initial processing

```
# look at terms and idf
print('terms:', vectorizer.vocabulary_)
print('idf:', vectorizer.idf_)
```

```
terms: {'mary': 7, 'had': 3, 'little': 6, 'lamb': 5, 'the': 13, 'followed': 2,
'to': 14, 'school': 11, 'one': 9, 'day': 1, 'was': 15, 'white': 16,
'should': 12, 'not': 8, 'bring': 0, 'is': 4, 'rebel': 10}
idf: [2.09861229 2.09861229 2.09861229 2.09861229 2.09861229 1.18232156
 1.69314718 1.18232156 2.09861229 2.09861229 2.09861229 1.69314718
 2.09861229 1.69314718 1.69314718 2.09861229 2.09861229]
```

Applying transform gives a tfidf sparse matrix. Just as in the CountVectorizer, each row represents a document in the corpus, and each column represents a term.

Code 19.2.3 — TfidfVectorizer. Create sparse tfidf matrix

```
# look at the docs
corpus_tfidf = vectorizer.transform(corpus)
print(corpus_tfidf.toarray())
```

```
[[0.         0.         0.         0.6614376  0.         0.3726424
  0.53364369 0.3726424  0.         0.         0.         0.
  0.         0.         0.         0.         0.        ]
 [0.         0.42304773 0.42304773 0.         0.         0.23833771
  0.         0.23833771 0.         0.42304773 0.         0.34131224
  0.         0.34131224 0.34131224 0.         0.        ]
 [0.         0.         0.         0.         0.         0.32700044
  0.         0.         0.         0.         0.         0.
  0.         0.46828197 0.         0.58042343 0.58042343]
 [0.45007472 0.         0.         0.         0.         0.25356425
  0.         0.25356425 0.45007472 0.         0.         0.36311745
  0.45007472 0.         0.36311745 0.         0.        ]
 [0.         0.         0.         0.         0.58042343 0.
  0.46828197 0.32700044 0.         0.         0.58042343 0.
  0.         0.         0.         0.         0.        ]]
```

19.3 Isolating the Test Data

The vectorizers in the previous examples were applied to the entire corpus. The vectors could then be divided into train and test sets. A more generally acceptable practice is to split data into train and set first, then vectorize the training set. This vector model can then be used to process the test set. Any words that the vectorizer encounters in the test set that were not encountered in training will simply be set to 0. The reason some people recommend this process is so that information about the test set is not 'leaked' to the training data. The process is demonstrated below.

Code 19.3.1 — Processing Train and Test Data. A general approach.

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split

# read the data
df = pd.read_csv('mydata.csv')

X = df.text     # features
y = df.labels  # targets

# train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
        test_size=0.2, train_size=0.8, random_state=1234)

# vectorize

X_train = vectorizer.fit_transform(X_train)  # fit and transform
X_test = vectorizer.transform(X_test)         # transform only
```

After the data is read, variables X and y represent the text and label columns, respectively. Then a built-in sklearn function is used to split the data into train and test X and y values. The function parameters indicate that 20% of the data is for test. The rest would be for train, so the train size parameter is not needed. A random state parameters ensures that we get the same train/test split each time the code is run.

The vectorizer fit-transform function is applied to the training data only. Once the vectorizer is fit, it can be applied to the test data with the transform method.

19.4 Adding More Features

The bag of words vectorizers demonstrated above have the disadvantage in that they lost word order. Adding n-gram features can bring back in some word order. The code in this section and in the online notebook for this section shows how to add n-gram features using bigrams. Adding n-grams that occur only once would not help a classifier, and neither would adding n-grams that occur too often. The tf-idf vectorizer can be applied to n-grams as well as unigram tokens.

Code 19.4.1 — TFidfVectorizer. Including bigrams

```
vectorizer = TfidfVectorizer(min_df=2, max_df=0.5,
                             ngram_range=(1, 2)))
```

The min and max document frequencies in the code above specify that a feature should occur in at least 2 documents but in less than 50% of the documents. The ngram range (1, 2) will extract both unigrams and bigrams. This is still a bag of words model but it is a bag of unigrams and bigrams.

19.5 Summary

This chapter showed how to transform raw text to numeric arrays that can be input to machine learning algorithms. The numeric arrays are sparse matrices, particularly suited to linear machine learning algorithms discussed in the next chapter.

The vectorization methods demonstrated above have many parameters that can be modified to suit different NLP applications. The methods described have options to remove stop words. Removing stop words is good for some applications and reduces performance in others. For example, in a sarcasm detection task described in the next chapter, accuracy was reduced by about 5% when stop words were reduced from the corpus! Often, the only way to know the best parameters is through experimentation.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

20. Naive Bayes

This chapter looks at a classic algorithm for machine learning: Naive Bayes. The Naive Bayes classifier is considered to be a linear classifier since it takes a linear combination of inputs to estimate the probability of a target output variable, y. Given some input data set $D = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$:

$$y = \mathbf{w}\mathbf{X} + b. \tag{20.1}$$

where **w** represents weights that are multiplied by the **X** input features and b is a fitting parameter. The parameters w and b are learned from the data D. Keep in mind that *linear* does not always mean a straight line, but any line that can be created with a linear combination of inputs. For example, $y = x^2$ is a line, but not a straight line.

In NLP classification, the target, y, will sometimes represent a binary class such as spam or not-spam. This is a binary classification problem. When there are more than two classes, it is a multinomial classification problem. An example is sentiment analysis when the target is one of three classes: positive, neutral, or negative.

20.1 Naive Bayes

Naive Bayes is a popular classification algorithm. The mathematical foundations of Naive Bayes go back to the 18th Century and the mathematician and minister, Thomas Bayes, who formalized the probabilistic equation that bears his name, Bayes theorem:

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \quad aka: \quad \mathbf{posterior} = \frac{\mathbf{likelihood} \times \mathbf{prior}}{\mathbf{marginal}} \tag{20.2}$$

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Let's consider the above equation in terms of a spam-detection data set.

$$P(spam|data) = \frac{P(data|spam)P(spam)}{P(data)} \tag{20.3}$$

The quantity $P(data|spam)$ is called the **likelihood**. The likelihood is calculated from the training data by determining the joint probabilities of the spam label and the data. Likelihood quantifies how likely it is that we would see the data given the spam or not-spam label. The quantity $P(spam)$ is called the **prior** and the distribution of this data is also learned from the training set. The denominator $P(data)$ is used to normalize the fraction to a probability in the range 0 to 1. It is also called the marginal. The quantity to the left of the equal sign is called the **posterior** and it will be the probability of the positive class for a given observation.

20.1.1 The Algorithm

Calculating joint probabilities with the chain rule would be mathematically intractable. The simplifying assumption of the naive Bayes algorithm is that each of the predictors is independent. Therefore:

$$p(X_1, X_2, ...X_D|Y) = \prod_{i=1}^{D} p(X_i|Y) \tag{20.4}$$

The naive assumption of the independence of the predictors is typically not true, but perhaps surprisingly, naive Bayes works well. Naive Bayes forms a good baseline for comparing other classifiers.

The algorithm requires a single read through the data to estimate parameters. There are two sets of parameters to estimate from the data and two ways we can estimate them from the test data. The two methods are maximum likelihood estimates (MLE) and maximum apriori (MAP). MLE simply involves counting instances in the training data while MAP additionally makes some estimates based on prior distributions of the data. The two sets of parameters are counts for the probability of each class, and parameters for each predictor.

The first set of parameters to estimate is the probability of each class. If we estimate this with MLE, we just calculate the number of observations in each class. The estimate for class c will be the count of observations with class c divided by the number of observations:

$$MLE_c = \frac{|Y = y_c|}{|N|} \tag{20.5}$$

Estimating parameters for predictors depends upon their type. Binary features can use the mean of the Bernoulli distribution to get probabiities for each class. For discrete variables with more than 2 categories, the mean of the multinoulli distribution for each category can be used. Parameters for quantitative predictors are estimated from the mean and variance of the Gaussian distribution.

The MLE for the likelihood of a predictor given the class is also achieved by counting the data for discrete predictors. For predictor X_i and class c:

$$\hat{\theta}_{ic} = \frac{|X_{ic}|}{|N_c|} \tag{20.6}$$

It is possible that a given predictor for a given class may have 0 observations. In this case the estimate is 0 which is a problem given the multiplication of predictor likelihoods. An approach that eliminates this problem is smoothing, which involves adding a little to the numerator and denominator:

$$\hat{\theta}_{ic} = \frac{|X_{ic}| + l}{|N_c| + |V|} \tag{20.7}$$

The value added to the denominator, |V|, represents the size of the vocabulary. If l=1 it is called Laplace or add-one smoothing. If we let l be a larger value this corresponds to a MAP estimate for the likelihood. We can add smoothing in a similar way to the MLE for the prior, in effect making it a MAP estimate. This type of smoothing is the simplest method but there are numerous other smoothing methods not discussed here.

For continuous variables, the mean and standard deviation of the predictor can be estimated but we would really like these values as they are associated with each class. Therefore separate mean, μ, and standard deviation, σ^2, values are computed by class. This is called a Gaussian naive Bayes classifier.

$$\hat{\theta_{ic}} = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{20.8}$$

20.1.2 Example: Spam Detection

For this example, a small and simple spam data set is used, the sms data set from the UCI Machine learning repository. The data has 5574 SMS messages, one in each row. The columns of interest are this text column and the label column which is 0 for ham (not spam) and 1 for spam.

The data is divided into X, the text, and y the target. Here's a peek at the first few rows of X:

```
0    Go until jurong point, crazy.. Available only ...
1                        Ok lar... Joking wif u oni...
2    Free entry in 2 a wkly comp to win FA Cup fina...
3    U dun say so early hor... U c already then say...
4    Nah I don't think he goes to usf, he lives aro...
Name: text, dtype: object
```

As you can see, the text is very casual speech. The target y is just a vector of 0s and 1s. The online notebook shows that the data was first divided into a train and test set, then vectorized with tf-idf transformations. This is similar to what was done in the last chapter, and is shown in a code block below. A choice was made to tell the vectorizer to remove stopwords from the messages using NLTK's list of stop words.

Code 20.1.1 — Text Preprocessing. Set up Vectorizer

```
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer

stopwords = set(stopwords.words('english'))
vectorizer = TfidfVectorizer(stop_words=stopwords)
```

The vectorizer is fit to only the training data, in order to prevent test data from leaking into the training process. Once the vectorizer is fit, it can be applied to the test data.

Code 20.1.2 — Text Preprocessing. Apply Vectorizer

```
# apply tfidf vectorizer
X_train = vectorizer.fit_transform(X_train)  # fit and transform train
X_test = vectorizer.transform(X_test)        # transform only on test
```

Now we are ready to train the Naive Bayes algorithm.

Code 20.1.3 — Naive Bayes. Training

```
from sklearn.naive_bayes import MultinomialNB

naive_bayes = MultinomialNB()
naive_bayes.fit(X_train, y_train)

#output
MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
```

Three things happened in the code block above. First, the algorithm is imported. MultinomialNB was chosen because it works well with discrete data such as word counts, but also can handle tfidf data. Another option is to use BernoulliNB when using binary present/not word vectors. After the algorithm is imported, an instance of the algorithm is created, and called 'naive_bayes'. Finally the fit() method does the actual learning on the training data.

Default settings were used in the MultinomialNB() algorithm. These settings are:

- alpha: additive (Laplace) smoothing (0 for no smoothing); the default shown in the output above is 1 for add-one smoothing
- class_prior: lets you specify class priors
- fit_prior: if True, learn priors from data; if false, use a uniform prior

So what exactly did the algorithm learn? The algorithm learns two things: the prior probability of spam, not spam, and the likelihood of each word being in a spam or not-spam email. These can be extracted from the model:

```
# priors
prior_p = sum(y_train == 1)/len(y_train)
print('prior spam:', prior_p, 'log of prior:', math.log(prior_p))
# output the prior the model learned
naive_bayes.class_log_prior_[1]

# output:
prior spam: 0.13388472473507365 log of prior: -2.01077611244103
-2.0107761124410306
```

The second thing it learned was the likelihood, actually log-likelihoods of each word being associated with spam or not spam. Those can be extracted as well:

```
naive_bayes.feature_log_prob_
```

```
#output:
array([[-9.643029  , -9.67373923, -9.47714135, ..., -6.31041976],
       [-8.23356461, -7.60523447, -9.19154209, ..., -9.19154209]])
```

The next code block makes predictions on the test data and evaluates the accuracy.

Code 20.1.4 — Naive Bayes. Evaluate on Test Dta

```
# make predictions on the test data
pred = naive_bayes.predict(X_test)

# print confusion matrix
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, pred)

# output:
array([[848,   0],
       [ 32,  88]])
```

The first row of the confusion matrix for two classes shows the true positive (TP) on the left and the false positive (FP) on the right. The second row shows the false negative (FN) on the left and the true negative (TN) on the right. A perfect classifier would have all the observations in the down-right diagonal and 0s in the down-left diagonal.

Another thing to consider is the distribution of the classes. In the test set for this data, 87.6% of the observations were not spam. If the classifier just guesses not spam, it will get almost 88% accuracy. This classifier got an accuracy of 96.6%, so it was able to learn something from the data.

Code 20.1.5 — Naive Bayes. Evaluate Test-set Predictions

```
from sklearn.metrics import classification_report
print(classification_report(y_test, pred))

# output
              precision    recall  f1-score   support

           0       0.96      1.00      0.98       848
           1       1.00      0.73      0.85       120

    accuracy                           0.97       968
   macro avg       0.98      0.87      0.91       968
weighted avg       0.97      0.97      0.96       968
```

The confusion matrix above showed that there were 32 messages that were spam that were misclassified as not-spam. The online notebook examines the misclassified messages. First, spam messages that were misclassified had a lot of numbers, exclamations points and capital letters. The tokenizer had removed those. As an experiment, a second try was made with different preprocessing to reflect these observations. The second classifier still had 100% precision, but the recall improved to 85%, bringing the overall accuracy up to 98%.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Machine learning is not as simple as tossing in data and waiting for the magic to happen. Machine learning is often an iterative process, or trying different pre-processing, different algorithms with different parameter settings, and seeing what works best. In order to do this appropriately, a data set is often divided into three parts: a training set, a validation set, and a test set. The "testing" of different models would be done in an iterative fashion using the validation data as the test data. The test set is used finally, only when the machine learning practitioner is satisfied that the best model has been developed.

20.2 Metrics

Classification can be evaluated by many measures. This section looks at accuracy, precision, recall, Kappa, AUC, and ROC curves.

20.2.1 Accuracy, sensitivity and specificity

The most common metric to evaluate results in classification is accuracy:

$$acc = \frac{C}{N} \tag{20.9}$$

where C is the number of correct predictions, and N is the total number of test observations. Accuracy is a good first look at the performance of a classifier, but often we will need to dig deeper in to the results. The confusion matrix specifies classification accuracy, broken down by class.

The output of the confusion matrix function above was:

```
848   0
32    88
```

The diagonal values from the upper left to the lower right are the correctly classified instances, 848 and 88 in this case. The other values are errors, 0 and 32. The flip side of accuracy is the error rate, calculated by subtracting accuracy from 1.

We can break down each component of the confusion matrix as follows:

```
pred T  F
   T TP FP
   F FN TN
```

- TP - true positive: these items are true and were classified as true
- FP - false positive: these items are false but were classified as true
- TN - true negative: these items are false and were classified as false
- FN - false negative: these items are true but were classified as false

$$accuracy = \frac{TP+TN}{TP+TN+FP+FN} \tag{20.10}$$

$$error\ \ rate = \frac{FP+FN}{TP+TN+FP+FN} = 1 - accuracy \tag{20.11}$$

The **recall** metric measures the true positive rate:

$$recall = \frac{TP}{TP + FN} \tag{20.12}$$

The **precision** metric measures accuracy of the positive class:

$$precision = \frac{TP}{TP + FP} \tag{20.13}$$

Precision and recall range from 0 to 1, just as accuracy does, with values closer to 1 being better. They help to quantify the extent to which a given class was misclassified.

20.2.2 Kappa

Cohen's Kappa is a statistic that attempts to adjust accuracy by accounting for the possibility of a correct prediction by chance alone. Kappa is often used to quantify agreement between two annotators of data. Here we are quantifying agreement between our predictions and the actual values. Kappa is computed as follows:

$$\kappa = \frac{Pr(a) - Pr(e)}{1 - Pr(e)} \tag{20.14}$$

where Pr(a) is the actual agreement and Pr(e) is the expected agreement based on the distribution of the classes. The following interpretation of Kappa is often used but there is not universal agreement on this. Kappa scores:

- <0.2 poor agreement
- 0.2 to 0.4 fair agreement
- 0.4 to 0.6 moderate agreement
- 0.6 to 0.8 good agreement
- 0.8 to 1.0 very good agreement

20.2.3 ROC Curves and AUC

The ROC curve is a visualization of the performance of machine learning algorithms. The name ROC stands for Receiver Operating Characteristics which reflects its origin in communication technology in WWII in detecting between true signals and false alarms. The ROC curve shows the tradeoff between predicting true positives while avoiding false positives.

Figure 20.1 shows an ROC curve. The y axis is the true positive rate while the x axis is the false positive rate. The predictions are first sorted according to the estimation probability of the positive class. A perfect classifier would shoot straight up from the origin since it classified all correctly. We want to see the classifier shoot up and leave little space at the top left. Starting at the orgin, each prediction's impact on the curve is vertical for correct predictions and horizontal for incorrect ones. If we see a diagonal line from the lower left to the upper right, then our classifier had no predictive value.

A related metric is AUC, the area under the curve. AUC values range from 0.5 for a classifier with no predictive value to 1.0 for a perfect classifier.

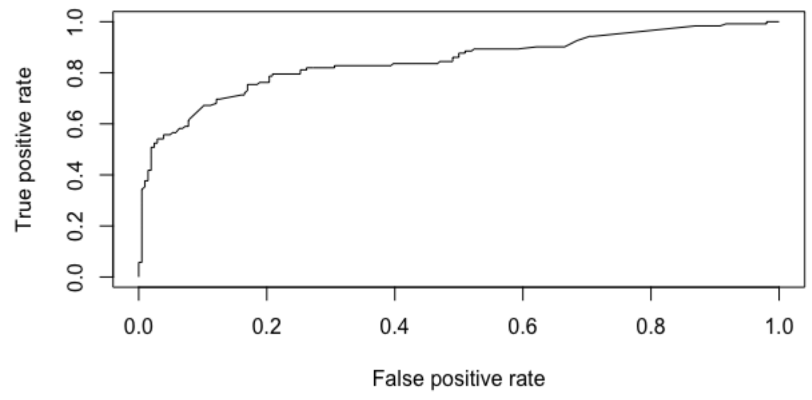


Figure 20.1: ROC Curve

20.2.4 Summary

Naive Bayes is a good first choice for text classification problems. The algorithm runs very fast, requiring only a single pass over the data. For small data sets, Naive Bayes may outperform more powerful classifiers.

20.2.5 Practice

Exercise 20.1 Practice Naive Bayes.

1. Go to the UCI Machine Learning Repository Data Sets page: `https://archive.ics.uci.edu/ml/datasets.php`
2. In the left pane, check Classification for Default Task and Text for Data Type. Find a data sent that interests you. Download the data.
3. Divide into train and test sets
4. Vectorize the data
5. Run Naive Bayes on the training data
6. Evaluate on the test data using several metrics
7. Analyze if you think Naive Bayes performed well on this data

■

21. Logistic Regression

Despite its name, logistic regression, performs **classification**, not regression. Whereas in linear regression, the target variable is a *quantitative* variable, in logistic regression, the target variable is *qualitative*: class membership. Linear models for classification create decision boundaries to separate the observations into regions in which most observations are of the same class. The decision boundary is a linear combination of the X parameters. As shown in Figure 21.1, the two classes are almost perfectly separated by the decision boundary. There is one misclassified observation.

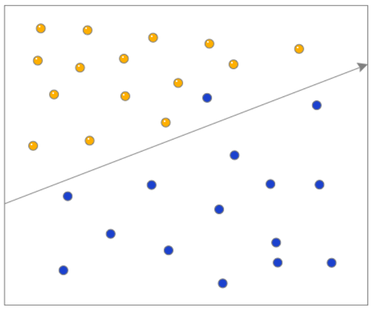


Figure 21.1: Decision Boundary for Binary Classification

21.0.1 An Example: Classifying Document Categories

The jupyter notebook in the github on the 20-news data set is an example of multiclass classification. The 20newsgroups data set is available from sklearn. The data in sklearn is already divided into train and test. Although the data organizes news articles into 20 categories, only 4 categories are extracted in the notebook: alt.atheism, soc.religion.christian, comp.graphics, and sci.med.

Another important sklearn feature demonstrated in the notebook is the Pipeline, an sklearn feature

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

that combines any number of tasks, sequentially. This Pipeline only contains a tfidf vectorizer and the logistic regression fit.

Code 21.0.1 — Logistic Regression. 20newsgroup data

```
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model.logistic import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score,
     recall_score, f1_score, log_loss

pipe1 = Pipeline([
        ('tfidf', TfidfVectorizer()),
        ('logreg', LogisticRegression(multi_class='multinomial',
                solver='lbfgs', class_weight='balanced')),
])

pipe1.fit(twenty_train.data, twenty_train.target)
```

The TfidfVectorizer() used default settings, and was discussed in the last chapter. The LogisticRegression() function set a few parameters:

- multi-class='multinomial' to set up the algorithm for this data
- class_weight='balanced' since the data is evenly distributed by class; this option is useful when the data set is unbalanced
- solver='lbfgs' is a good choice for multiclass problems; read about other solvers in the sklearn documentation; 'lbfgs' refers to an optimization algorithm, L-BFGS (Broyden-Fletcher-Goldfard-Shanno) that uses less computer memory.

One advantage of using the pipeline approach is that the same processing can easily be applied to the test data, as shown in the next code block. The overall accuracy across the four classes was 90%. The precision-recall balance varied by topic.

Code 21.0.2 — Logistic Regression. Predict and Evaluate

```
# evaluate on test data
twenty_test = fetch_20newsgroups(subset='test', categories=categories,
      shuffle=True, random_state=42)
pred = pipe1.predict(twenty_test.data)

from sklearn import metrics
print(metrics.classification_report(twenty_test.target, pred,
     target_names=twenty_test.target_names))

print("Confusion matrix:\n",
       metrics.confusion_matrix(twenty_test.target, pred))

import numpy as np
print("\nOverall accuracy: ", np.mean(pred==twenty_test.target))
```

```
                         precision    recall  f1-score   support

           alt.atheism       0.95      0.81      0.87       319
         comp.graphics       0.85      0.96      0.90       389
               sci.med       0.93      0.88      0.90       396
soc.religion.christian       0.90      0.94      0.92       398

              accuracy                           0.90      1502
             macro avg       0.91      0.90      0.90      1502
          weighted avg       0.91      0.90      0.90      1502

Confusion matrix:
 [[258  13  14  34]
 [  3 374   6   6]
 [  5  41 347   3]
 [  6  11   5 376]]

Overall accuracy:  0.9021304926764314
```

In addition to predicting a class, sklearn can extract the probabilities for each class as shown below. In these 5 examples, notice that the 4th example had a very high probability for the first class, about 91%. In contrast, the third example did not have any probability over 50%. The highest was 39%.

```
probs = pipe1.predict_proba(twenty_test.data)
probs[:5]

# output:
array([[0.14242452, 0.1643475 , 0.5691282 , 0.12409978],
       [0.0410866 , 0.03622316, 0.88370887, 0.03898138],
       [0.28678063, 0.10662445, 0.39017533, 0.21641959],
       [0.91486571, 0.02017099, 0.02211033, 0.04285297],
       [0.13633055, 0.06384863, 0.10635021, 0.6934706 ]])
```

21.0.2 Probability, odds, and log odds

What is the difference between odds and probability? Let's look at this using a sports outcome example. Imagine we played 10 games with a friend and won 7. That means we lost 3 of course. Assuming we will do as well next time, our odds are 7 to 3:

$$odds = \frac{number\ of\ wins}{number\ of\ losses} = \frac{7}{3} \tag{21.1}$$

If we want to express the same data as a probability:

$$probability = \frac{number\ of\ wins}{number\ of\ games} = \frac{7}{10} \tag{21.2}$$

Notice that probability will always range from 0 to 1, whereas odds will not. The following equation converts odds to probability:

$$probability = \frac{odds}{1 + odds} \tag{21.3}$$

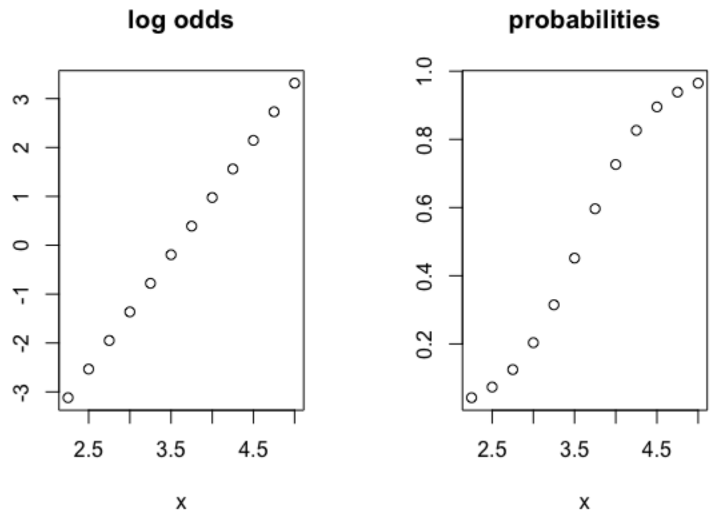


Figure 21.2: Log Odds versus Probability

Let's see what this means in terms of a sample logistic regression problem. The quantitative predictor ranges from around 2.09 to 5.06 in this data set. Compare the effect of the predictor across values from 2.25 to 5.0. Figure 21.2 plots the log odds across this range of values on the left, and on the right, the associated probabilities across the same range of values. Observe that the results of the logistic regression model (the log odds) are linear in the parameters (w, b) but that the associated probabilities are not linear.

The coefficient for the quantitative predictor in the logistic regression model was 2.34. For every one-unit increase in x, the probability of the target changes by exp(2.34)/[1+exp(2.34)]. In Table 21.1, the X column is the predictor for a range of values. The log odds is found by plugging in the value of x for the logistic regression formula given by: 2.34 * x - 8.383.

X	Log Odds	Probability
2.5	-2.53	0.07
3.0	-1.36	0.20
3.5	-0.19	0.45
4.0	0.977	0.73
4.5	2.147	0.89

Table 21.1: Log Odds and Probability for Quantitative Data

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

As you see in Table 21.1 and more clearly in Figure 21.2, the log odds are linear in the parameters but the probability is not linear, we can discern a subtle S-shape in the probabilities.

Logistic Regression Coefficients
In linear regression we interpret a predictor coefficient as the amount of change in y for a 1-unit change in x. We cannot make this intepretation in logistic regression. The predictor coefficient in a logistic regression model specifies the change in log-odds for a 1-unit change in x.

21.1 The Algorithm

The linear regression output is a quantitative value that could range over all the real numbers. What we need for logistic regression classification is a function that will output probabilities in the range [0, 1]. The sigmoid, or logistic, function is used for this purpose and of course is where the algorithm gets its name. When real numbers are input to the logistic function, the output is squashed into the range [0,1] as seen in Figure 21.3. The logistic function is:

$$f(x) = \frac{1}{1+e^{-x}} \tag{21.4}$$

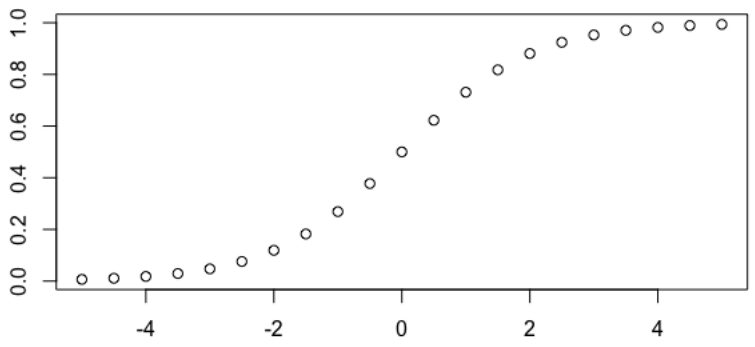


Figure 21.3: The Logistic Function

The logistic regression algorithm computes the log odds from the estimated parameters. The log odds is just log(odds). The odds are the probability of the positive class, p(x), over the probability of the negative class (1 - p(x)). If we have a single predictor w_1 and an intercept w_0, the log odds are:

$$log \frac{p(x)}{1-p(x)} = w_o + w_1 x \tag{21.5}$$

Solving for p gives us the logistic function:

$$p(x) = \frac{e^{-(w_0+w_1x)}}{1+e^{-(w_0+w_1x)}} = \frac{1}{1+e^{-(w_0+w_1x)}} \tag{21.6}$$

We can see in these equations why logistic regression is considered a linear classifier. It creates a linear boundary between classes in which the distance from the boundary determines the probability. When we use the logistic function for classification, the cut-off point is usually 0.5. Notice in Figure 21.3 that this is the inflection point of the S-curve. Probabilities greater than 0.5 are classified as the positive class and probabilities less than 0.5 are classified in the other class.

21.2 Mathematical Foundations

How are the parameters, **w**, found for logistic regression? First an appropriate loss function is established, then an optimization technique such as gradient descent is used to find optimal parameters.

In machine learning, a loss function (sometimes called a cost function) is a function that measures how wrong predictions are. The goal is to minimize the loss. The loss function for linear regression:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{n}(y_i - f(x_i))^2 \tag{21.7}$$

If we plug in the logistic function for f(x), it will not be a convex function. That is a problem because gradient descent works only for convex functions. A suitable loss function for logistic regression can be found by starting with the likelihood function, L:

$$L(w_0, w_1) = \prod_{i=1}^{n} f(x_i)^{y_i}\ (1 - f(x_i))^{1-y_i} \tag{21.8}$$

Notice in the likelihood equation that one of the terms in the product will always reduce to 1 because the y values are either 0 or 1. To simplify the likelihood equation, take the log of it to find the log-likelihood, ℓ:

$$\ell = \sum_{i=1}^{n} y_i\ log f(x_i) + (1 - y_i)\ log(1 - f(x_i)) \tag{21.9}$$

The log-likelihood equation for each instance:

$$\ell = y\ log f(x) + (1 - y)\ log(1 - f(x)) \tag{21.10}$$

In training the classifier, we want to penalize it for wrong classifications. This is our loss function. The penalties follow directly from Equation 5.16. Our Loss is:

$$\mathcal{L} = -log(f(x))\ if\ y = 1 \qquad \mathcal{L} = -log(1 - f(x))\ if\ y = 0 \tag{21.11}$$

These two loss functions are visualized in Figure 21.4. In the leftmost graph which represents -log(fx), the closer the function gets to 1, the smaller the penalty but the closer it gets to 0, the higher. We want to penalize -log(f(x)) severely if it classifies as 0 when the true target is 1. The reverse is true when the true target is 0. This is shown in the rightmost graph. Here we penalize -log(1-f(x)) severely as it moves toward 1 because the true target is 0.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

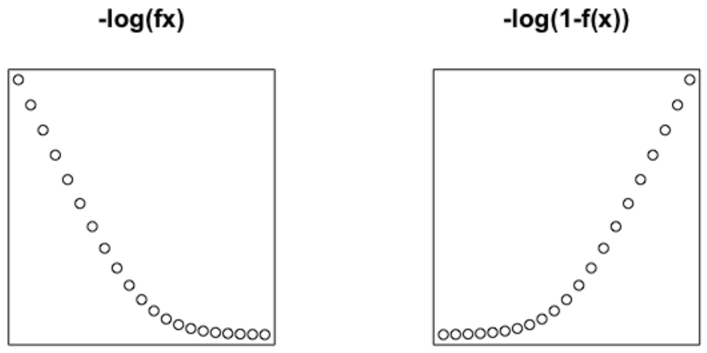


Figure 21.4: Loss Function for Logistic Regression

We can put the two loss functions into one equation as shown below, where they are summed over all observations. Notice that one of the terms will always be zero because y will be either 0 or 1.

$$\mathcal{L} = -\Big[\sum_{i=i}^{N} y_i log(f(x_i)) \quad + \quad (1-y_i) log(1-f(x_i))\Big] \tag{21.12}$$

where f(x) =

$$f(x) = \frac{1}{1+e^{-(w^T x)}} \tag{21.13}$$

The parameters w are then determined using gradient descent, or some other optimization function. You can see from Figure 21.4 that when you put together the two loss functions you will have a convex function suitable for gradient descent. Once the parameters are known, classifying new instances is done by plugging in the instance into the logistic function shown in Equation 5.19. This returns a probability in the range [0, 1]. Typically we establish a threshold such as 0.5. Values over that threshold are classified as 1, values below are classified as 0.

21.3 Advanced Topic: Optimization Methods

Earlier we specified our log-loss function for logistic regression as follows:

$$\ell = \sum_{i=1}^{n} y_i \; log f(x_i) + (1-y_i) \; log(1-f(x_i)) \tag{21.14}$$

where f(x) =

$$f(x) = \frac{1}{1+e^{-(w^T x)}} \tag{21.15}$$

Next we find the gradient of the log likelihood. The gradient, g, is the partial derivative with respect to the parameters **w**. The gradient is the slope which is going to tell the algorithm which direction to move to find the minimum. The gradient equation is given below. Notice that it is really the X matrix multiplied by how wrong f(x) is at this point in predicting the true y.

$$g = \frac{\partial \ell}{\partial \mathbf{w}} = \sum_i (f(x_i) - y_i)\mathbf{x}_i = \mathbf{X}^T(f(x) - \mathbf{y}) \tag{21.16}$$

All we need for gradient descent is the first derivative, the gradient. For other optimization methods we will also need the second derivative, the Hessian, H. The Hessian is a square matrix of second partial derivatives that gives the local curvature of a function. Note that in the following equation for the Hessian, the S represents $S \triangleq diag(p_i(1 - p_i))$

$$H = \frac{\partial}{\partial \mathbf{w}} g(\mathbf{w})^T = \sum_i (\nabla_w f(x_i))\mathbf{x}_i^T = \sum_i f(x_i)\;(1 - f(x_i)\;\mathbf{x}_i\;\mathbf{x}_i^T = \mathbf{X}^T\mathbf{S}\mathbf{X} \tag{21.17}$$

There are many algorithms to find the optimal parameters, **w**. The first one we examine is gradient descent.

21.3.1 Gradient Descent

Gradient descent is an iterative approach where at each step the estminated parameters, θ get closer to the optimal values. We can express this as follows:

$$\theta_{k+1} \;=\; \theta_k - \eta_k\, g_k \tag{21.18}$$

where η is the learning rate, which specifies how big of a step to take at each iteration. If the eta (step size) is too slow the algorithm will take a long time to converge. If eta is too large, the true minimum can be stepped over and then the algorithm will not be able to converge.

21.3.2 Stochastic Gradient Descent

For large data sets, gradient descent can bog down. An alternative is *stochastic gradient descent* which processes the data either one at a time or in small batches instead of all at once. It is stochastic because the observations are chosen randomly. If the data is processed one at a time it means that the gradient has to be computed at each step. It turns out that the gradient will reflect the underlying function better if it is computed from a small batch. This also improves computation time.

21.3.3 Newton's Method

Gradient descent finds the optimal parameters using the first derivative. Newton's method is another approach; it uses the second derivative, the Hessian defined above. Newton's method (also called Newton-Raphson) is also an iterative method. At each step either the full Hessian is recalculated, or it is updated in which we call the method quasi-Newton. In a well-behaved convex function, Newton's method will converge faster.

A key insight in Newton's method is that if it is computationally difficult to compute a minimum for a given function, then come up with a function that shares important properties with the original function but is easier to minimize. At each iteration, Newton's method constructs a quadratic approximation to the objective function in which the first and second derivatives are the same. The approximate function is minimized instead of the original.

How is this approximate function found? A Taylor series about the point is used, but ignores derivatives past the second. A Taylor series converts a function into a power function and the first few terms can be used to get an approximate value for a function.

21.4 Comparison of Naive Bayes and Logistic Regression

The jupyter notebook 'sarcasm' compares the performance of Naive Bayes and Logistic Regression on a headlines data set. The data set is a combination of headlines from The Onion and a real news source. The data is fairly evenly divided between sarcastic and not-sarcastic data.

Naive Bayes and Logistic Regression got fairly comparable results, about 85% accuracy. The size of the data set is about 28K observations. If the Naive Bayes independence assumption holds, as the number of training observations grows to infinity, the two classifiers become similar. If the data is very small, often Naive Bayes will perform better than logistic regression.

Naive Bayes has higher bias but lower variance than logistic regression. Bias is the tendency of an algorithm to make assumptions about the data. Variance is the sensitivity of an algorithm to noisy data. Bias and variance are related to underfitting and overfitting, shown below. A model that is too simple has higher bias and is likely to underfit the data. A model that has too high variance is likely to overfit the training data and not generalize well to the test data.

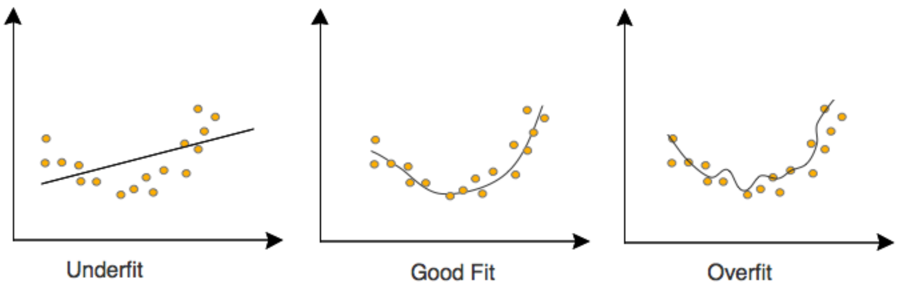


Figure 21.5: Overfitting versus Underfitting

High-bias classifiers tend to underfit. In the 'Underfit' diagram above, the classifier wanted to see a straight line when there really wasn't one. High-variance classifiers tend to overfit. In the 'Overfit' diagram, the classifier tried too hard to create a complex model that fit every training point. If a classifier does well on the training data and poorly on the test data, you have probably overfit the data. Finding a good fit to the data involves first selecting the right classifier based on your knowledge of how the classifier works and the nature of the data. Other methods to find a good fit include preprocessing the data into an optimal representation, and setting any model hyperparameters that influence model fit.

Mathematically, Naive Bayes and Logistic Regression are quite different classifiers. Naive Bayes estimates parameters for P(Y) and P(X|Y) from the data. This is called a *generative* classifier. In contrast, logistic regression directly learns P(Y|X) from the data. This is called a discriminative classifier.

21.5 Summary

Logistic regression performs well when the classes are linearly separable. Logistic regression has high bias, but not quite as much as Naive Bayes. For data that is not linearly separable, classifiers with lower bias should be selected.

21.5.1 Practice

Exercise 21.1 Practice Naive Bayes and Logistic Regression.

1. Go to the UCI Machine Learning Repository Data Sets page: `https://archive.ics.uci.edu/ml/datasets.php`
2. In the left pane, check Classification for Default Task and Text for Data Type. Find a data sent that interests you. Download the data.
3. Divide into train and test sets
4. Vectorize the data
5. Run Naive Bayes on the training data and evaluate on the test data
6. Run Logistic Regression on the training data and evaluate on the test data
7. Compare the performance of the two algorithms

■

22. Neural Networks

The foundational concepts of neural networks have been around since the 1950s. However, in recent years, researchers have pushed boundaries far beyond earlier results. There are two reasons for the renewed interest in neural networks: (1) processing power has increased, and (2) larger data is commonly available. Neural networks often outperform other algorithms if there is sufficient data and the underlying data is more complex than it is linear.

People often say that deep neural networks act like the human brain. This is a gross overreach. The human brain has about 100 billion neurons, and each neuron is connected to 1000 or more other neurons. Obviously, we don't have such massively parallel computer architecture at our disposal. In the brain, each neuron has multiple inputs that receive signals from other neurons. When the combined inputs reach a certain threshold, the neuron fires, sending signals out to connected neurons. We don't fully understand how the human brain learns from these interconnected neurons. Despite the overhype, neural networks are very powerful algorithms.

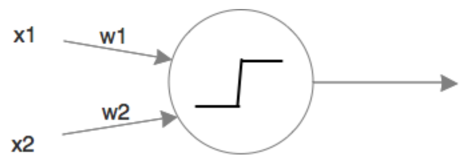


Figure 22.1: Perceptron

Neural networks, also called artificial neural networks (ANNs), are not 'like' the human brain, but they were inspired by biological neural networks in mammals. Frank Rosenblatt is credited with developing the perceptron for pattern recognition in the late 1950s. A perceptron, as illustrated in Figure 22.1, takes several input values and produces an output signal. In this example, there are two inputs x_1 and x_2 and these are multiplied by their respective weights and sent through the activation function. The activation function, here illustrated as a step function, will output +1 if the combined inputs reaches a certain threshold, and −1 otherwise. The total input to the perceptron is: $\sum_i w_i x_i$

The perceptron was overhyped by its proponents and the press, as has often been the case in advances throughout the history of AI. A backlash came in 1969 with a book by Minsky and Papert[1] that described the limitations of the single-layer perceptron; for example, that it could not learn the XOR function. Although it was demonstrated soon thereafter that combining perceptrons in multiple layers could learn XOR, the negative perception persisted for decades. In fact, a network of these perceptrons can learn any function that can be expressed with step functions.

Further improvements were made by replacing the step function as the activation function with smoother functions such as the sigmoid. Having different activation functions allows the network to do regression as well as classification. A neural network, then is the evolution of the perceptron to include multiple neurons in a network with a sigmoid or other flexible activation function.

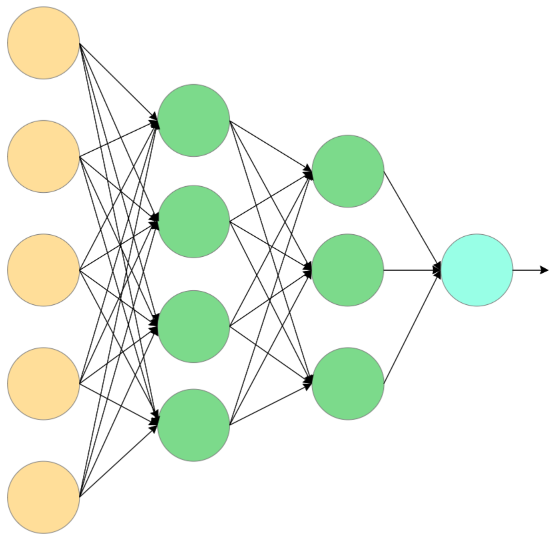


Figure 22.2: Neural Network with 2 Hidden Layers

A neural network is illustrated conceptually in Figure 22.2. This is typically called a feed-forward network because the variables and weights are calculated left to right, going forward through the network. Reading forward through the network, left to right, we see 5 input nodes in yellow. There are two hidden layers illustrated with green nodes, 4 in the first hidden layer and 3 in the second hidden layer. Finally the output layer in blue gives the output of the network. In a feed forward network such as this, each node's output is an input to the next layer. Further, each connection in the illustration has a weight value that is multiplied by the output of the source node to compute the input to the destination node. Each hidden layer can have a different activation function. Each individual node (neuron) in the hidden layers can learn something different, essentially learning different things from the input so that the network is essentially a composition of functions. Finally, the output node(s) can have a different activation function than the nodes in previous layers. This complex architecture is what makes neural networks powerful enough to learn complex functions. A "deep" neural network as discussed in later chapters is a neural network with many hidden layers.

[1]Minsky, Marvin, and Seymour Papert. "Perceptrons: An lntroduction to Computational Geometry." (1969).

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

22.0.1 Hidden Nodes and Layers

The most important decision when building a neural network is its architecture, or topology. The input and output layers should be fairly intuitive for a given problem but designing the hidden layers is challenging. How many hidden layers? How many nodes in the hidden layers? Having too few hidden nodes may result in underfitting while having two many can result in overfitting. There are a few rules of thumb to find the number of hidden nodes:

- between 1 and the number of predictors
- two-thirds of the input layer size plus the size of the output layer
- < twice the input layer size

Are there any advantages in having two layers rather than 1? Figure 22.3, shows a regression neural network on a housing data set with 13 input variables. In this particular problem, a neural network with one layer of 9 nodes performed significantly worse than linear regression. However, having the 9 nodes split into two layers performed better than linear regression. If the data is linearly separable, one layer should be enough. Having two layers can capture a more complex relationship, at the cost of extra training time. A potential downside of more layers is that you may overfit if you have a small training set. A simpler architecture is usually recommended for small data.

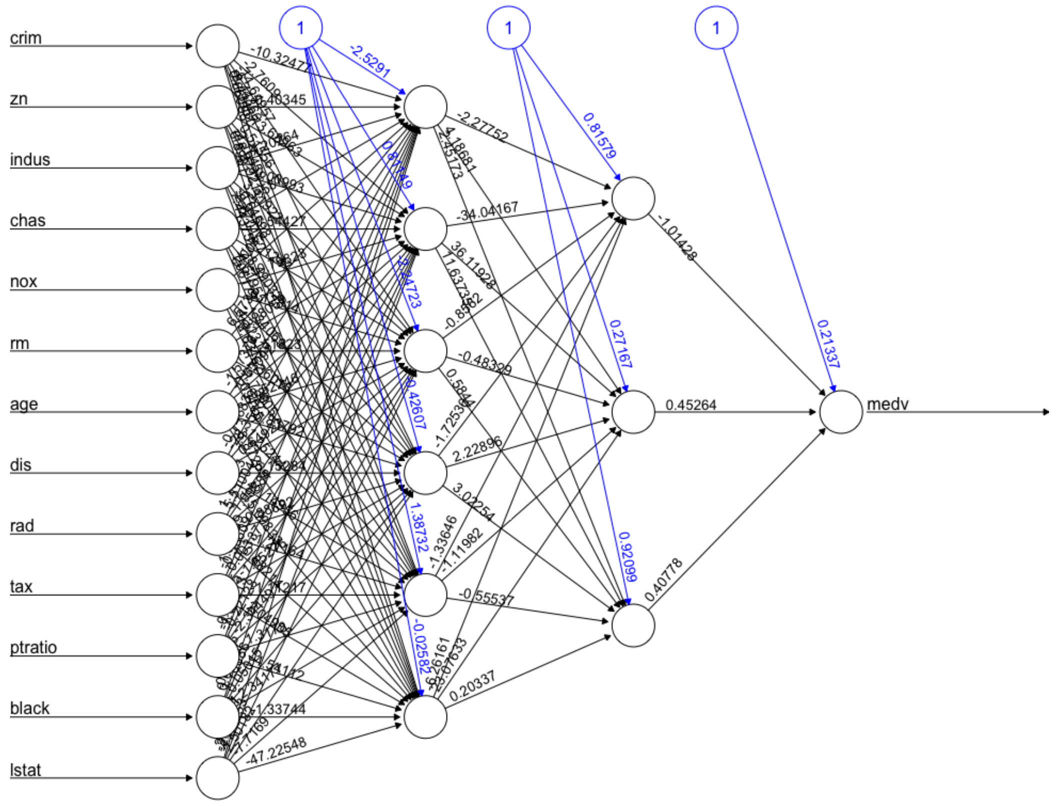


Figure 22.3: Neural Network with 2 Hidden Layers

22.1 The Algorithm

In Figure 22.3, observe that each neuron has its own bias term and weight. Every neuron in the hidden and output layers will have a bias term, as you can see in the neural net plot above. The bias is like the intercept term in a linear regression or logistic regression model, it is always 1, and multiplied by its weight. In a feed forward neural network, each input including the bias term is multiplied by its weight to get a sum of inputs. This is just basic matrix multiplication which is optimized in modern computers to be very fast. Once a neuron or node receives its sum of weighted inputs, the activation function kicks in to produce its output.

So far we have not described any learning. We have just described the feed forward mechanism. The weights are initialized randomly so the first output of the feed-forward mechanism is just a random guess. So how does it learn? Like any supervised algorithm, a neural network learns from labeled data. On the first pass through the network with its random weights, the network will quantify how close the output is to the true label. The network improves by adjusting the weights using *back propagation*, and efficient methods of computing gradients. Back propagation adjusts weights by assigning blame backwards to neurons in the previous layer. The error is the difference between the output and the true label. Ideally we would like this error to be zero. If we had a single input we could adjust that input's weight using the slope, (the derivative). However, we will have many inputs each with their own weights. The derivatives will be in matrix form and give us the gradient matrix. We want to move down that error surface simultaneously for all inputs. Each input node's weights are adjusted according to its own gradient. Visualize this as n skiers, one for each input, skiing down different mountain slopes simultaneously where some slopes are steeper than others. Back propagation will update the previous hidden layer, then the next, all the way back to the input layer. Training, then, is just a series of forward and backward passes until convergence, meaning that the error is under some threshold value.

22.2 Mathematical Foundations

Training a neural network is an optimization problem, as is the case for many machine learning algorithms. These problems need a loss function that quantifies how far off the predicted labels are from the true labels. For both regression and classification neural networks, the mean squared error can be used:

$$\mathcal{L} = \frac{1}{2N} \sum_{i=1}^{n} (y_i - f(x_i))^2 \tag{22.1}$$

The error term helps with the credit assignment problem, determining how much blame for the error to assign to each input weight. The derivative of the cost function provides the gradients, the rate of change of the cost function with respect to the neuron's output.

$$\triangledown E = \Big(\frac{\partial E}{\partial w_1}, \ ..., \ \frac{\partial E}{\partial w_n}\Big) \tag{22.2}$$

The gradients are computed and aggregated across training examples to get adjustments to be made to the weights. The errors are multiplied by a parameter, alpha, the learning rate.

$$\triangle w_i = -\alpha \frac{\partial E}{\partial w_i} \tag{22.3}$$

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Each forward and backward pass through the network is called an *epoch*. If the alpha is too small it may take too long to converge, but if it is too large, the optimum might be overstepped. Further, the error surface may not be perfectly convex but be bumpy and may have local minima.

The gradient descent approach described above is sometimes called *batch gradient descent* because we update the gradients for all examples in one batch. An alternative is *stochastic gradient descent*, in which the weights are updated after each example. The error surface for different examples will look different so this prevents us from getting stuck in a local minima, at the cost of greater computation time. The term *stochastic* means that examples are chosen randomly.

22.3 Neural Network Architecture

A neural network can be considered an extension of a generalized linear model because it is mapping the input variables through a function approximation to get the output variable(s). Training time is considerably longer for neural networks than linear or logistic regression, but once the network has trained, running new observations through the network is fast.

The more complex the architecture (the number of hidden layers and nodes), the longer training will take. The algorithm may reach its predefined maximum number of iteration steps before it converges below the specified threshold. Both the number of steps and the threshold are adjustable parameters in most frameworks. It's unlikely that you will get lucky and specify an architecture that performs optimally by chance. Creating the network is an iterative trial-and-error process. It helps to see how the error is diminishing while training. If your data is very large, you might want to go through this trial-and-error phase with a subset of the training data, like 10K observations or less.

When the derivatives are below a default threshold value, training stops. Recall that the derivatives give the slope and the minimum error will be when the derivatives are near 0 in the gradient descent algorithm. If the algorithm fails to converge with the default, you could set the threshold to a slightly larger value. However, setting a larger threshold will probably result in a less accurate model.

22.4 Activation Functions

Each neuron, also called node, in the network has an activation function that controls the output based on the weighted sum of inputs. The simplest activation function is the step function, which is illustrated in the perceptron earlier in the chapter. A simple linear function could be an activation function for output nodes but not hidden nodes. The reason is that backpropagation would not work in the hidden nodes since the derivative of the linear function would be a constant. Figure 22.4 shows some commonly used activation functions.

The sigmoid (aka logistic) function outputs values between 0 and 1, thereby normalizing the output of the neuron. Then tanH (aka hyperbolic) function is similar to the logistic function but centers the outputs at 0. This can be an advantage for modeling data with strong positive and negative values. The ReLu (rectified linear unit) function has a derivative and can be used in back propagation. ReLu has been popular in recent years because using it makes the training converge faster. The softmax activation function is often used in an output mode for multi-class classification. Softmax normalizes the outputs for each class and output the probability for each class. New activation functions as well as variations on the classics are developed every year.

To avoid overfitting, a regularization term can be added to the activation function. The regularization term is modified by another parameter, lambda. As lambda gets larger, the coefficients of the activation function will shrink. The intercept does not shrink because the goal is to reduce the coefficients

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

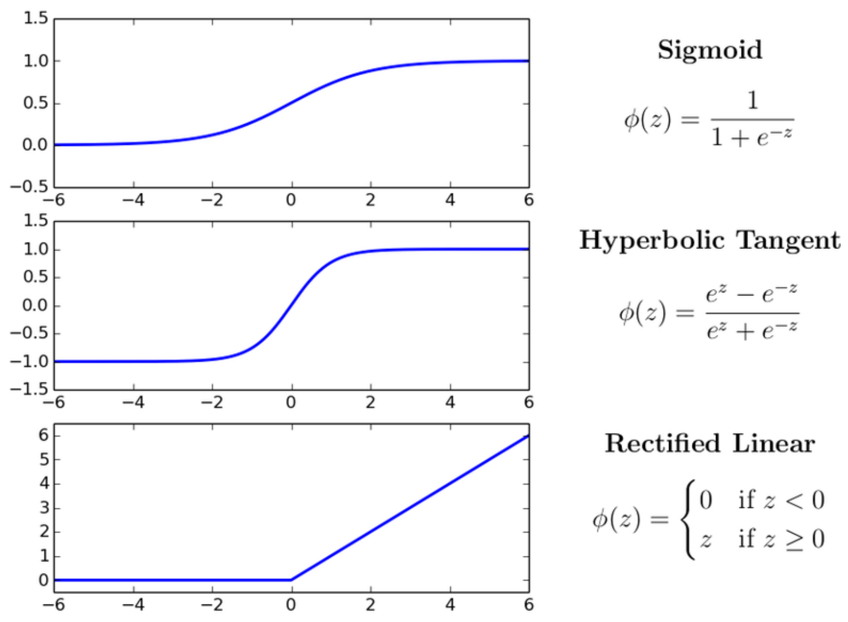


Figure 22.4: Common Activation Functions

associated with predictors. The notation $||w||^2$ denotes the *l*2 norm, which is $\sqrt(\sum_{j=1}^{p})w_j^2$. This is referred to as L2 norm. Another normalization options is the L1 norm which is the Manhattan distance.

22.5 Neural Networks in sklearn

The neural network implementation in sklearn are not for large data implementations and there is no GPU support. The next chapter explores Keras, which can run larger data with either a CPU or GPU.

The MLP multi-layer perceptron algorithm in sklearn can do either classification or regression. When using numeric data in neural networks, scaling the data is strongly recommended. However, the examples in this chapter use preprocessed tf-idf data, so that all features (word) are already on the same scale.

The neural network algorithm has a lot of parameters. A full list is available on the sklearn site.

Here are a few important parameters:

- hidden_layer_sizes – a tuple representing the number of neurons in layers, left to right. The default is (100,) for a single layer of 100 neurons.
- activation: – activation functions for neurons in hidden layers. Options include logistic, tanh, and relu.
- solver – weight optimization solver. Options include lbfgs (a quasi-Newton optimization, which is the solver we used in logistic regression), sgd (stochastic gradient descent), adam (an sgd variant). The adam solver is recommended for training sets with thousands of examples. For smaller data sets, lbfgs can perform better.
- alpha – penalty for L2 regularization. Default is 0.0001.
- batch_size – batch size for minibatch gradient descent. Default is min(200, n_samples). This options is not available with the lbfgs solver.
- learning_rate – progression of learning rate. Options include constant, invscaling (gradually de-

creases), and adaptive (keeps constant as long as the training loss decreases, otherwise decreases). The learning rate parameter is only used with the sgd solver.
- learning_rate_init – initial learning rate. Default is 0.001.
- max_iter – maximum number of training epochs. Default is 200.
- tol – tolerance for convergence. Training stops when loss is not improving by at least this amount. Default is .0001.
- verbose – print messages if True. Default is False.

22.6 Code Example

The GitHub has a couple of examples of training a neural network in sklearn. The following is sample code to set up a neural network classifier.

Code 22.6.1 — Neural Network. In sklearn

```
from sklearn.neural_network import MLPClassifier
classifier = MLPClassifier(solver='lbfgs', alpha=1e-5,
                   hidden_layer_sizes=(15, 2), random_state=1)
classifier.fit(X_train, y_train)
```

The predict and evaluation methods work the same as for Naive Bayes and Logistic Regression. This is one of the advantages of sklearn.

22.7 Summary

Neural networks are defined by these properties:
- the network architecture or topology - the number of neurons in the model and the number of layers
- the activation function which transforms the inputs to an output; the sigmoid function is commonly used
- the training algorithm

Neural networks can learn complex functions from data. Generally they will not outperform simpler models for small data but truly shine when the amount of data is large and the function to learn is complex.

22.7.1 New Terminology

This terminology will be important for the deep learning chapters in the rest of the book.
- backpropagation
- epoch
- neuron or node
- perceptron
- activation function

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

22.7.2 Practice

Exercise 22.1 Practice Neural Networks.

1. Go to the UCI Machine Learning Repository Data Sets page: `https://archive.ics.uci.edu/ml/datasets.php`
2. In the left pane, check Classification for Default Task and Text for Data Type. Find a data sent that interests you. Download the data.
3. Divide into train and test sets
4. Vectorize the data
5. Run Naive Bayes on the training data and evaluate on the test data
6. Run Logistic Regression on the training data and evaluate on the test data
7. Run Neural Networks on the training data and evaluate on the test data
8. Compare the performance of the three algorithms

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

VI

Part Six: Deep Learning

Part Six

Part Six gives an overview of deep learning. Chapter 23 provides an introduction to the Keras library for deep learning, using Sequential Models. Chapter 24 goes beyond the Sequential Model to look at more advanced architectures for deep learning: CNN, RNN, LSTM, and GRU. Chapter 25 explores different ways to train word embeddings, and how to use pretrained embeddings. Chapter 26 examines the sequence-to-sequence architecture that has become popular in recent years.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

23. Deep Learning

Simply put, deep learning involves neural networks with many hidden layers. Deep learning was made possible by increased computing power, and availability of more data. However, it became a hot trend because it improved performance on many hard-to-solve problems involving pattern recognition. Another reason that deep-learning became popular is that deep learning automates feature engineering. In previous chapters, we discussed a few techniques for shaping raw data into features that are useful for machine learning algorithms. With deep learning, the algorithm itself can learn features from the inputs at the same time it is learning the target function for the data.

23.1 Keras

Keras was released in March 2015 as a deep learning API that could run on top of TensorFlow, Theano, or other deep learning libraries. In general, writing code with Keras is cleaner, simpler code than writing in deep learning libraries directly. The Keras API allows a great deal of flexibility in designing networks in terms of layers architecture, loss functions, optimizations, and more.

Keras 2 was released in 2017, and is now integrated with Tensorflow. Going forward, Keras will have two implementations. One is the Keras API which can still run on top of either TensorFlow, Theano, or some other back end. The other Keras implementation is incorporated within TensorFlow, and is available as tf.keras. Code written in Keras 2 should be stable for many years. However, there may be some compatibility issues with older Keras code going forward. The following link summarizes the changes: `https://blog.keras.io/introducing-keras-2.html`

Instructions for installing Tensorflow/Keras will vary, depending on whether your system has a GPU with certain additional libraries installed. Having the GPU version can make processing 5 to 10 times faster, but if you are patient, the CPU version will get there. TensorFlow/Keras can run on Windows, but setup is easier on Mac/Linux. One option is to have a dual-boot Ubuntu set up if you are on a Windows machine. A linux machine or mac should have no problems. If your computer has a GPU, consult the documentation about requirements and installation: `https://www.tensorflow.org/install/gpu`. For computers that will run TensorFlow on the CPU, simply install TensorFlow 2 with pip or pip3: `pip3 install tensorflow` This will install TensorFlow and tf.keras, which will be the preferred Keras implementation going forward.

If you do not want to install TensorFlow, you can use Google Colab, `https://colab.research.google.com/`, which integrates the most popular Python and machine learning libraries in a Jupyter notebook type of environment.

23.2 Structures in Keras/TensorFlow

To better understand the Keras/TensorFlow workflow, a little background is provided into the data structures and model structures of Keras.

23.2.1 Data representation

The fundamental data representation in TensorFlow is a *tensor*, which is a generalization of vectors and matrices to an arbitrary number of dimensions, called axes. A tensor is defined by:

- Rank - the number of axes; example: a matrix has two axes.
- Shape - a tuple of integers that describe the dimensions by axis; example: (5,) for a vector of length 5.
- Type - such as integer or double

A scalar is a 0D tensor. A vector is a 1D tensor. For data that fits into a row/column organization, 2D tensors of shape (samples, features) are sufficient. Time series or sequential data requires 3D tensors of shape (samples, timesteps, features). Images require 4D tensors of shape (samples, height, width, channels) or some variation, where channel is 1 for greyscale and 3 for RGB data. Video will require 5D tensors with the 5th axis for frames and the others similar to images.

23.2.2 Model structures

The core structures in Keras are layers and models. A *layer* is an object that inputs tensors and outputs tensors. Each layer represents one transformation step in a multi-layer deep network. The *model* defines how the layers work together. The simplest model is the Sequential model, a linear stack of layers. The first example we look at below uses a Sequential model.

23.3 Keras Example

The example presented here is from Francois Challot's book *Deep Learning with Python*, currently being updated to a second edition. Francois Challot was an AI researcher at Google when he developed Keras. Challot continues to be influential in the integration of Keras into TensorFlow. This example uses the IMDB movie review data set that is built into TensorFlow. The original notebook can be found on The Deep Learning with Python book's GitHub: `https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/3.5-classifying-movie-reviews.ipynb`

The imdb data set can be loaded as shown below. The `num_words` parameter limits the vocabulary size to the most frequent 10K words. This data has already been processed and is ready to import into a model for training. Later, we will discuss how to preprocess your own text data for TensorFlow. The data consists of movie reviews, represented as vectors of word indices. The labels are binary representations of positive/negative ratings.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

```
from tensorflow.keras.datasets import imdb

(train_data, train_labels), (test_data, test_labels) =
    imdb.load_data( num_words=10000)
```

These reviews are simply lists of integers, and must be converted to tensors. One way is to one-hot encode the lists into binary vectors. Since the vocabulary is 10,000 words in this example, each example will be binary vector of length 10,000, where 1s represent the presence of words in that example. The following code will convert the vectors of indices into one-hot vectors.

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    # Create an all-zero matrix of shape (len(sequences), dimension)
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.0  # set results[i, word] to 1s
    return results

x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

Now the training data has shape (n, 10000) where n in the number of training examples. The labels need to be converted to float arrays as follows:

```
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

After data preparation, the next step is to build the network.

Code 23.3.1 — Building a network. Sequential Model

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(16, activation='relu'),
    layers.Dense(16, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])
```

The model is a Sequential model builds the network layer by layer. Other models explored later are the convolutional network which can learn from ‘tiles’ of the input, and the recurrent network which can retain state to learn from sequence data. The network above has 3 Dense layers. The two hidden layers have ReLu activation, which was discussed in the previous chapter. The first argument to Dense() in each layer is the number of units (nodes). The first layer also specifies the shape of the input data.

The first two Dense layers have 16 units, so that they each will output data in shape (n, 16). This means that the network has to learn a representation from (n, 10000) space to (n, 16) space. The final layer needs only one unit with sigmoid activation in order to output the binary classification probability in the range [0, 1].

After the model is constructed, the model needs to be compiled. The example below shows settings for 3 parameters:

- loss – the binary_crossentropy loss function is a good choice for binary classification that outputs probabilities
- optimizer – rmsprop is a gradient-based optimization, similar to gradient descent, but with momentum to avoid a vanishing gradients problem that is common in gradient descent with many layers.
- metrics – accuracy is a good choice for any classification problem.

Code 23.3.2 — **Compile a network.** Sequential Model

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Training requires validation data, used as a test set while training. The following code takes 10,000 examples from the training set and places them in the validation set.

Code 23.3.3 — **Set aside validation data.** Sequential Model

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]

y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

Finally, everything is set up for training. The model.fit() method uses the x and y training data and validation data to learn the model weights. The code below specifies 20 epochs (forward and backward passes), and takes the data in batches of 512 examples at a time. The results of the training were stored in a history object which stores information such as training accuracy and loss. This information can be extracted and used to plot training results, as shown in the online notebook and in Figure 23.1.

Code 23.3.4 — **Train the model.** Sequential Model

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

Once the model is trained, the model.evaluate() method can run the test data through the model to create predictions on the test data and evaluate the results.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Code 23.3.5 — Evaluate the model. Sequential Model

```
results = model.evaluate(x_test, y_test)
```

The model trained for 20 epochs achieved an accuracy of 86% on the test data.

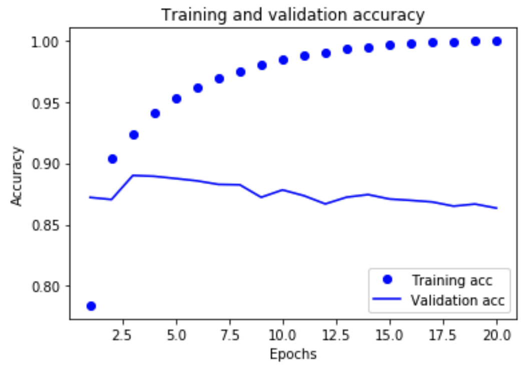


Figure 23.1: Training History over 20 Epochs

Figure 23.1 shows that training accuracy increases through all 20 epochs, but the validation accuracy starts decreasing after about the fourth epoch. The model has overfit the data. Many options exists to deal with overfitting, but in the online notebook the simple solution used is just train for 4 epochs. The model trained for only 4 epochs achieved a slightly higher accuracy. When a model is not overfit, the model can generalize better to previously unseen data to achieve higher accuracy.

23.4 Deep Learning Basics

Now that we have run through an example of building a network, training, and evaluating, let's step back and discuss some of the terminology that we just used. The example had only two intermediate layers, so this perhaps is not truly 'deep' learning. However, the principles and terminology remain the same as more layers are added. The basic algorithm is the same as discussed in the previous chapter on neural networks: the data feeds forward through the network, at each layer being multiplied by the weights, until the output value is calculated. The output has an error which we measure with a loss function, also called the objective function. This acts as a feedback signal to modify the weights. This modification is done by the optimizer, typically some variation of the back propagation algorithm that was discussed in the previous chapter. Training proceeds forward and backward through iterations called epochs. Training stops when the error is below a certain threshold.

What advantages do deep networks have over shallow ones? The most obvious answer to that is that many layers help learn more complex functions. Beyond that, deep architectures help with feature engineering. In a shallow network the input data must already have been manually manipulated into a good representation for the network. Deep learning can partially automate this by learning features as it learns the function in many layers. According to Challot in his book *Deep Learning with Python*, these are the two key principles that make deep learning unique and powerful:

1. increasingly complex representations are learned layer by layer over many iterations
2. the intermediate representations are learned jointly so that as one feature changes, all the others are adjusted automatically

Deep architectures have a lot of moving parts: activation functions, optimization schemes, metrics, and more. Deep learning frameworks often don't work on all the data at a time, but process it in batches. The number of batches will be a parameter we can select. As a consequence of handling data in batches, the optimization function is mini-batch stochastic gradient descent. SGD is called stochastic because the selection of observations in a batch are random.

23.4.1 Layers and Units

Designing the optimum number of layers, and units within layers, is not yet a science but more of a trial-and-error approach that you will develop intuitions for as you gain experience. A few guidelines are:

- The number of units in an intermediate layer should be greater than or equal to the number of units in the subsequent layer, otherwise you create an information bottleneck.
- For small data sets, avoid overfitting by having a smaller number of hidden layers.
- To search for the best architecture for a given problem, start with a simple model on validation data. Iteratively increase the complexity of a model until you get diminished improvements at which point you may be overfitting.
- Add weight regularization to the hidden layers if overfitting is a concern. This constrains the weights to smaller absolute values. L1 regularization uses the L1 norm which is calculated from the absolute values of the coefficients (weights). L2 uses the L2 norm which is similar to Euclidean distance. L2 regularization is sometimes called weight decay. This regularization is only done during training.
- Another regularization technique during training is dropout. Dropout randomly sets a few output weights to 0. Although this seems like an arbitrary approach, it has been shown to reduce overfitting by randomly eliminating the impact of different units that may be tuning into noise instead of the function you want them to learn.

23.4.2 Activation Functions

Each layer can have its own activation function. If no activation function is specified, linear activation is assumed. This is typically the choice for the output layer of regression tasks. The sigmoid activation function will output a probability between 0 and 1, so it is often chosen as the activation function for the output layer of a binary classification task. For multi-class classification, the softmax activation is often used because it outputs probabities for each class. A popular choice for intermediate layers is the relu (rectified linear unit). The relu is linear for positive values but makes negative values 0.

23.4.3 Optimization Schemes and Loss Functions

The optimizer determines how learning proceeds and is associated with a loss function that measures how well it is learning. Although there are a bewildering array of options for all these parameters, there are a few choices that are standard. For regression problems, mean squared error is the standard loss function. For 2-class classification, binary crossentropy is standard, while for multi-class classification, categorical crossentropy is typically used. Crossentropy is a measure of how far off the predictions are from the true values.

23.5 Embeddings for Deep Learning

Text can be represented as vectors. Vectorizing text can be done by first tokenizing text into sequences of words, characters, or n-grams. These words, characters, or n-grams are then represented as vectors.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Two common methods for representing a word as a vector are one-hot encoding and word embedding. We will explore these and other embedding techniques in later chapters.

23.6 Keras Functional API

The examples in this chapter used the tensorflow.keras API. This API facilitates building a model, and the code is easy to read. Here's a repeat of that code:

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(16, activation='relu'),
    layers.Dense(16, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])
```

There is another API, the TensorFlow Functional API, that is more flexible, at the cost (in my opinion) of having less elegant and readable code. Documentation for that API is available online: `https://www.tensorflow.org/guide/keras/functional`.

Conceptually, in the Functional API, you are building graphs of layers. The graphs[1] are data structures that contain TensorFlow operation objects representing units of computation, and Tensor objects representing units of data. The advantage of the graph model is that these units can be saved and run independent of the original Python code, enabling faster execution and multithreading, and parallel execution, on GPUs and TPUs.

The imports will be the same as the code above. The first thing that will differ from the Keras code above is the explicit representation of the input layer. In the Functional API, we have to explicitly define the input layer:

```
inputs = keras.Input(shape=(10000,))
```

The batch size will be deduced from the input data, but the size of each sample needs to be indicated. The training data has shape (n, 10000), but we omit the n.

Next, we specify that we want a 16-unit Dense layer, just like before. This code lets 'x' hold the model structure as it is built. The code below creates the 16-unit Dense layer using the inputs defined above.

```
dense = layers.Dense(16, activation='relu')
x = dense(inputs)
```

Next we add another Dense layer to x. Notice the (x) argument in, with x is the output.

```
x = layers.Dense(16, activation='relu')(x)
```

Now we specify the output layer as a Dense layer with one unit.

[1]`https://www.tensorflow.org/guide/intro_to_graphs`

```
outputs = layers.Dense(1)(x)
```

Finally, we put it all together in the model by specifying the inputs and outputs. Optionally, you can give the model a name.

```
model = keras.Model(inputs=inputs, outputs=outputs, name='functional_model')
```

This code is in notebook 'Keras_imdb_2_FunctionalAPI'. The rest of the notebook continues as with the Keras API, the only difference is in how the model was built. The notebook also shows how to save, delete, and load models.

23.7 Summary

The workflow for Keras involves these steps:

- Define the dimensions and characteristics of the train and test data tensors.
- Define the model.
- Compile the model with an appropriate loss function, optimizer, and metrics.
- Fit the model.
- Evalute the results.

You probably won't hit a home run the first time through these steps so changing some parameters and repeating the above steps may happen multiple times.

This chapter showed how to build models with both the Keras API and the Functional API in TensorFlow. Notebooks associated with future chapters will use either approach.

23.8 Practice

Exercise 23.1 Try Keras.

1. Search Kaggle for 'text classification keras'
2. Look through at least 3 notebooks, taking notes on new techniques that you learn

■

24. Deep Learning Variations

24.1 Deep Architecture

The example in the previous chapter showed a Sequential neural network with 2 hidden layers, similar to the neural network using sklearn demonstrated in a previous part of the book. Where TensorFlow/Keras shows its power is in the varieties of architectures that you can build. Some of the most popular models include:

- Sequential – often the first approach to a problem; the sequential model is a plain stack of layers where each layer has one input tensor and one output tensor
- CNN (convolutional networks) – often used with image/video or text data
- RNN (recurrent network) – used with text data or timeseries data
- LSTM (long-short term memory) – an RNN variation that overcomes some training problems associated with RNNs
- GRU (gated recurrent unit) – enables faster training compared to LSTM

24.1.1 Sequential Networks

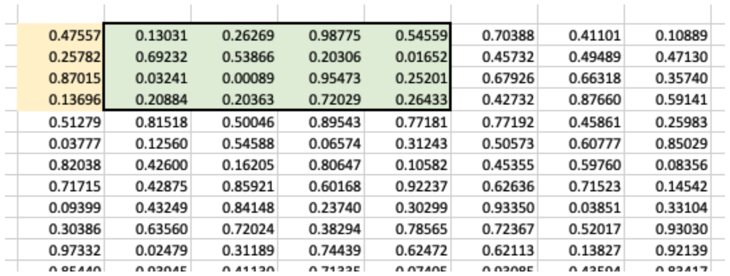
The sequential network consists of stacks of layers, often Dense layers. Each Dense layer maps relationships between input features. Dense layers are also used as the final layer for classification or regression. For binary classification, the final Dense layer should have a single unit with a sigmoid activation. For multi-class classification, the final Dense layer should have the number of nodes equal to the number of classes, with softmax activation. If the targets are integers, you can use 'sparse_categorical_crossentropy' for the loss function, but if the classes are one-hot encoded, use 'categorical_crossentropy' as the loss function. For regression, the final Dense layer has one unit with no activation, assuming that you are predicting one real-number value. Loss functions for regression can be mean_squared_error or mean_absolute_error.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

24.2 Convolutional Neural Networks

Convolutional neural networks, convnets, or CNNs, work well with image data. The reason is the way these networks learn. A densely connected sequential layer will learn global patterns from the input data. In contrast, the convolutional layer will learn patterns in small windows of the input data. This gives convnets two unique abilities. The first is that once a pattern is learned in one location it will recognize it when translated to another location. The second is that convolutional layers can learn spatial hierarchies. For example, a first convolutional layer could learn local features like edges, and subsequent layers could learn how these edges and other features combine to form complex and abstract concepts like faces.

Figure 24.1 shows a simple visualization of the CNN process. The figure is just a screen shot of a spreadsheet filled with random numbers to represent a data matrix. The 4x4 window has already traveled over the first 4x4 elements, and has now moved one stride to the right.



0.47557	0.13031	0.26269	0.98775	0.54559	0.70388	0.41101	0.10889
0.25782	0.69232	0.53866	0.20306	0.01652	0.45732	0.49489	0.47130
0.87015	0.03241	0.00089	0.95473	0.25201	0.67926	0.66318	0.35740
0.13696	0.20884	0.20363	0.72029	0.26433	0.42732	0.87660	0.59141
0.51279	0.81518	0.50046	0.89543	0.77181	0.77192	0.45861	0.25983
0.03777	0.12560	0.54588	0.06574	0.31243	0.50573	0.60777	0.85029
0.82038	0.42600	0.16205	0.80647	0.10582	0.45355	0.59760	0.08356
0.71715	0.42875	0.85921	0.60168	0.92237	0.62636	0.71523	0.14542
0.09399	0.43249	0.84148	0.23740	0.30299	0.93350	0.03851	0.33104
0.30386	0.63560	0.72024	0.38294	0.78565	0.72367	0.52017	0.93030
0.97332	0.02479	0.31189	0.74439	0.62472	0.62113	0.13827	0.92139

Figure 24.1: Convolving

Two additional layers are involved in the CNN example later in this section. One is padding, which involves just adding extra empty elements to get a tensor to be the correct shape. The other is maxpooling, which is used when downsizing feature representations. Maxpooling results in fewer parameters to learn.

A Conv1D layer works well on text data. Convnets consist of stacks of convolution and max-pooling layers, and end with either a Flatten operation or a global pooling layer, followed by a Dense layer for the final classification or regression.

24.2.1 How CNNs work

The term *convolution* refers to the mathematical process of combining two functions to create a new function. The new function will then be able to combine two sets of information. In a CNN layer, the convolution is performed on the inputs using a filter (sometimes called a kernel) to produce a feature map. The filter slides over the input and produces a feature map. This maps the input dimensional space to the output feature dimension space. The filter moves with overlap, rather than a tiled approach, which enables it to learn across the input space. The stride is the size of the filter step. The smaller the stride, the more overlap. The output feature dimensions will be smaller than the input dimensions, so padding is often used to keep the dimensions from shrinking.

A filter is basically a small numeric matrix with previously specified dimensions. Initially, the filter is filled with random numbers. The dot product of the filter matrix and the data it covers is computed and stored in a new matrix that is the output of the convolutional layer. The new output matrix is a new representation of the data. This output is passed to the next layer in the network. Multiple filters in a layer can be learning different features from the data.

Sometimes pooling layers are added between CNN layers. Pooling will reduce dimensions of the data, which will reduce the number of parameters that have to be learned, thereby shortening training time. Pooling can also reduce overfitting. One variation of pooling is max pooling, in which the maximum value a filter over a window is used for one value in the output window. For example, a 3x3 filter will take the max of the 9 values as the one output for that window. The max pooling is a new representation of the data. The intuition of the convolution followed by the max pooling is that the input data is reduced to extracted features.

The output of a convoluted layer will have smaller dimensions than the input layer. For a filter that is mxm, the input layer of nxn will be reduced to (n-m)x(n-m). Each layer will shrink the data dimensions. The filter does not convolve the edges as much as the inner portion of the data. With zero padding, border of zeros is added to the input to make the ouput the same size as the original input. The API will figure out the number of rows to add if you request zero padding. In Keras, the padding option of 'same' means the padding will be added to make the output size the same as the input size. This is zero padding. Another padding option is 'valid' padding which means no padding, that is, the output size will be smaller than the input size. The valid padding option is the default in Keras.

CNNs are often associated with image data, so how do they work on text data? The short answer is that text can be viewed as a 1D sequence of words, so a 1D CNN can convolve over spans of text to learn a more compact representation of the input data.

In a CNN, the filter, the activation, and the pooling work together to extract features. The filter assists in the convolution process which encodes a particular feature. The activation (usually ReLU) assists in learning to detect features. The pooling stage reduces the features for faster training.

24.2.2 Code example

The code example in the GitHub is a modification of the Sequential() model presented in the last chapter. The imdb data is read and processed the same. The shape of the training data is (25000, 500) because there are 25000 training examples, and the length of each example was shortened or padded to be 500 tokens. The tokens are represented as indices in the vocabulary.

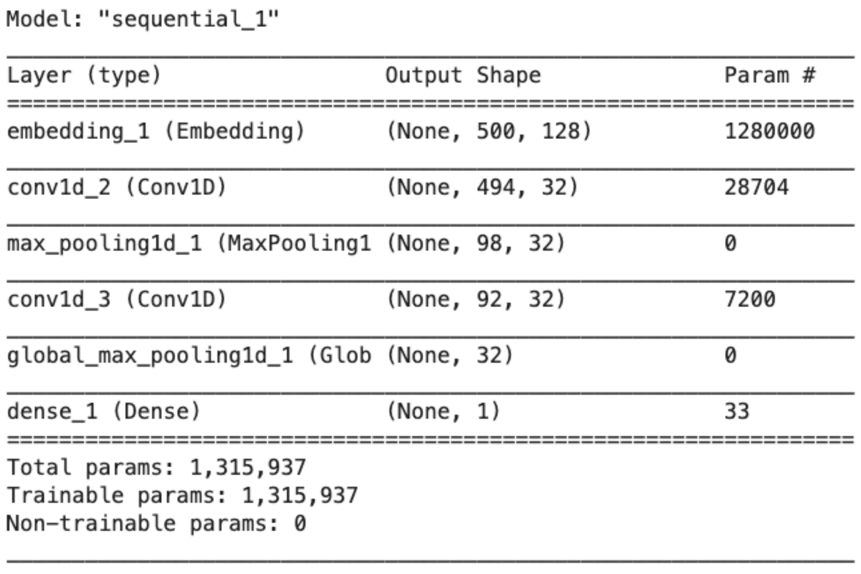
The model is built as follows. The embedding layer is followed by a Conv1D with MaxPooling, then another Conv1D followed by GlobalMaxPooling. The GlobalMaxPooling returns the max value over the entire sequence.

```
model = models.Sequential()
model.add(layers.Embedding(max_features, 128, input_length=maxlen))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))
```

And the summary is displayed as shown in Figure 24.2.

Let's look at the model summary to understand transformations at each layer. The first layer is the embedding layer:

```
code:
model.add(layers.Embedding(max_features, 128, input_length=maxlen))
summary:
embedding_1 (Embedding)      (None, 500, 128)          1280000
```



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 500, 128)          1280000
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 494, 32)           28704
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 98, 32)            0
_________________________________________________________________
conv1d_3 (Conv1D)            (None, 92, 32)            7200
_________________________________________________________________
global_max_pooling1d_1 (Glob (None, 32)                0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 33
=================================================================
Total params: 1,315,937
Trainable params: 1,315,937
Non-trainable params: 0
_________________________________________________________________
```

Figure 24.2: Model Summary

What did this embedding layer do? Word embeddings will be discussed in more detail in a later chapter, but basically, embeddings map words into a geometric space that reflects the semantic relationships between words. Pretrained embeddings can be used, but the neural network is capable of learning the embeddings during training. An advantage of this could be when training embeddings for a specialized domain. The Embedding layer took the dictionary index for each word, and transformed it into a vector. The input to the embedding layer is (number samples = 25000, sample length = 500) and the output is a 3D floating-point tensor of shape (number samples = 25000, sample length = 500, embedding dimension = 128).

The next two layers in the model are the Conv1D layer, followed by a MaxPooling layer. The arguments to the Conv1D layer specify 32 filters, a kernel size of 7x7, and relu activation. The default values of strides=1, padding="valid" are used.[1] The output shape of the Conv1D shows that the sample length has shrunk slightly due to the nature of the convolving.

The MaxPooling1D layer will use a pool size of 5. By default the stride size will be the size of the filter (5x5). The output shape will be $(input\_shape - pool\_size + 1)/strides)$ which in this case will be $(494 - 5 + 1)/5) = 98$

```
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
```

The model has one more pair of Conv1D and MaxPooling layers. The final layer is a Dense layer of size 1 for the classification.

[1]Full documentation of the Conv1D layer parameters are available online: https://keras.io/api/layers/convolution_layers/convolution1d/

24.2.3 CNNs for text classification

A look at the documentation[2] shows that Keras provides 1D, 2D, and 3D Conv layers, as well as variations on these layers. The 1D layer is best for sequential data like temporal data or text sequences. The 2D layer is often used as a spatial convolution over images. The 3D layer can be used as a spatial convolution over volumes. The example in this chapter uses the 1D layer.

An excellent discussion of CNNs for text classification is available in a 2016 paper by researchers Zhang and Wallace at UT.[3] The authors experimented with different configuration settings for one-layer CNNs to determine which design decisions are important in text classification. The authors find that filter region size should be tuned since it has a significant impact on performance. Based on their experiments, they also recommend the 1D max pooling strategy.

24.3 Recurrent Neural Networks

A recurrent neural network, RNN, has memory, or state, which enables it to learn a sequence. Conceptually, the RNN layer uses for loops to iterate over the sequence while maintaining an internal state that encodes the data for the portions of the sequence it has seen so far.[4]

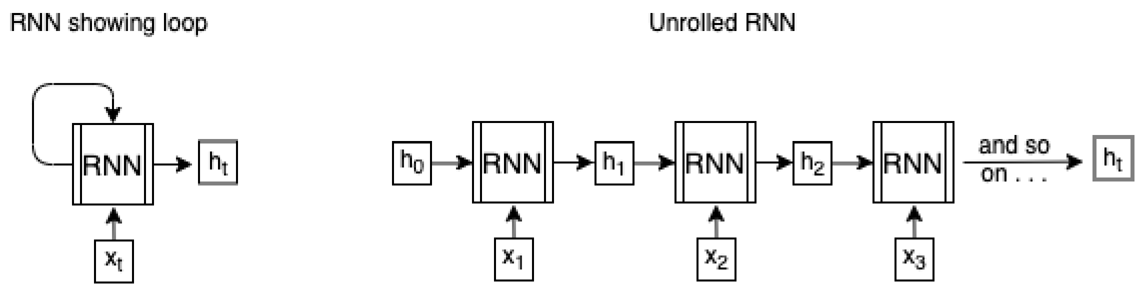


Figure 24.3: Recurrent Neural Network

As illustrated in Figure 24.3, the looping mechanism produces a new hidden state with each step. The final hidden state, h_t is a representation of the previous inputs. The advantage of a recurrent network is that it retains information through each step in a series. This is important in language modeling because the context of previous words is an important feature of a word. At each step, the input is not only the next word in the sequence, but also the context encoded in the previous step.

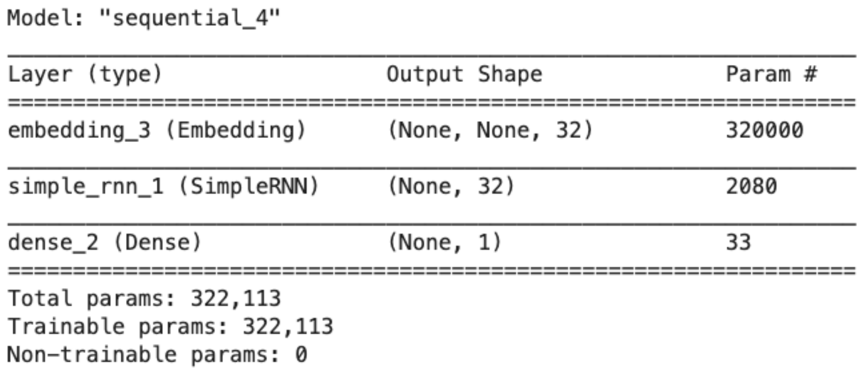
A problem with recurrent neural networks is the vanishing gradient problem, a problem that was identified early in the exploration of multi-layer neural networks. As you add layers, the gradient back propagated becomes smaller and smaller so that training is impossible. The Long Short-Term Memory (LSTM) algorithm is one method for avoiding vanishing gradients. The key is keeping the memory data path independent of the back propagation path and giving its own update mechanism.

Keras has three variants of the RNN: SimpleRNN, GRU, and LSTM. LSTM is the most powerful but also the most computationally expensive. GRU is a simpler alternative. In stacking RNN layers, each layer except the last should return the full sequence of outputs.

An online notebook demonstrates a simple RNN model on the imdb data. The model and summary are shown below:

[2]`https://keras.io/api/layers/convolution_layers/`
[3]`https://arxiv.org/pdf/1510.03820.pdf`
[4]`https://keras.io/guides/working_with_rnns/`

```
model = models.Sequential()
model.add(layers.Embedding(max_features, 32))
model.add(layers.SimpleRNN(32))
model.add(layers.Dense(1, activation='sigmoid'))
```



```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, None, 32)          320000
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, 32)                2080
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 33
=================================================================
Total params: 322,113
Trainable params: 322,113
Non-trainable params: 0
```

Figure 24.4: Model Summary

In contrast to the CNN, the RNN did not change the input size as the data was processed. In the online notebook, the RNN did not perform as well as the Sequential model of the previous chapter.

24.4 LSTM

The length of the sequences in RNNs can lead to the exploding or vanishing gradient problem. With a long sequence, back propagation can break down so that the gradients either shrink or expand excessively. This led to the development of the LSTM (Long Short Term Memory) model.

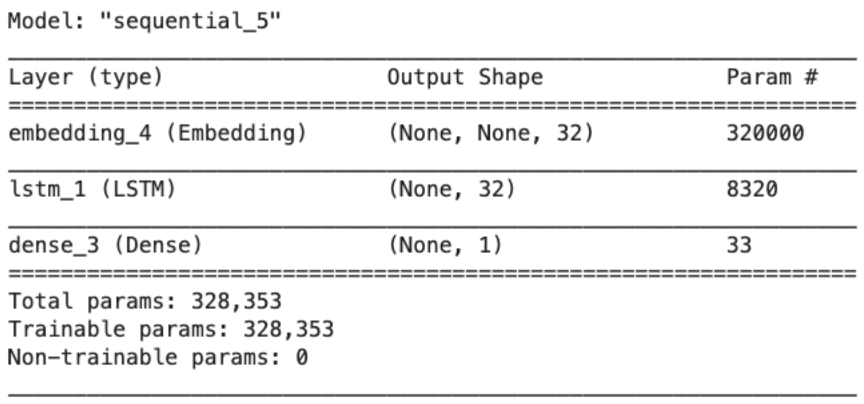
The LSTM model allows information to be carried forward, just as the RNN does, but it also allows information to be forgotten. This means that the network is not burdened with very long sequences, and therefore the gradient problem is eased.

The RNN notebook also includes code for the LSTM, shown below. The LSTM model had improved performance over the simple RNN model.

```
# build a model with LSTM
model = models.Sequential()
model.add(layers.Embedding(max_features, 32))
model.add(layers.LSTM(32))
model.add(layers.Dense(1, activation='sigmoid'))
```

RNNs and LSTMs are often described in terms of cells, *c*, that retain state. The LSTM has an additional hidden state, *h*. The hidden state is basically an encoding of the information learned so far. LSTMs deal with the vanishing/exploding gradient problem with a forget gate that filters out or allows through information.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***



```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_4 (Embedding)      (None, None, 32)          320000
_________________________________________________________________
lstm_1 (LSTM)                (None, 32)                8320
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 33
=================================================================
Total params: 328,353
Trainable params: 328,353
Non-trainable params: 0
_________________________________________________________________
```

Figure 24.5: Model Summary

24.5 GRU

The GRU (Gated recurrent unit) models are similar to LSTM but are more efficient to run. This is because the GRU combines the input and forgetting states. They will use less memory and train faster. Here is the set up for the GRU in the same notebook mentioned above.

```
model = models.Sequential()
model.add(layers.Embedding(max_features, 32))
model.add(layers.GRU(32))
model.add(layers.Dense(1, activation='sigmoid'))
```

In the online notebook, the model using GRU got the highest accuracy of all the models.

24.6 Summary

This chapter provided a lean introduction to deep learning variations, CNNs, RNNs, LSTMs, and GRUs. Guidelines were discussed on which model might work best in different circumstances. This foundation should enable you to try these models in your own projects. Deep learning is about learning by experience, but keep in mind that the more you read, the deeper your intuition.

Many of these concepts in deep learning can be fuzzy. Gaining a deeper intuition the hard way involves digging into the published papers on the developments as they happened historically. This path is not for the faint at heart.

24.7 Practice

Exercise 24.1 For each of CNN, RNN, LSTM, and GRU:

1. Search Kaggle for 'text classification x' where x is cnn, rnn, lstm, or gru
2. Look through at least 3 notebooks, taking notes on new techniques that you learn

■

25. Embeddings

In previous chapters, natural language has been represented in sparse matrices, where each column represents a word in the vocabulary, and each cell is a count or binary indicator. A further option was to take the sparse matrices and condense them into embedding layers. Either of these two very different representations can be the input to a neural network.

Starting around 2008, many new ways of representing text have been developed. These new embedding approaches are unsupervised, making them easier to implement, since so much text is freely available. Current developments in text representation focus on pre-trained representations. Of the many approaches, we can broadly divide them into two categories: context-free, or contextual. GloVe from Stanford NLP is an example of a context-free embedding. These *context-free embeddings* create representations for single words in the vocabulary. In contrast, *contextual embeddings* create representations using the context of words. Examples of contextual embeddings include ELMo from AllenNLP and BERT from Google.

25.1 Word embeddings

An embedding is a vector of floating point values of a predetermined length. An embedding representation is dense compared to a sparse matrix representation. The length of the embedding vectors can vary from 8-dimensional for very small data and up to 1024-dimensional for larger data sets. The higher the dimension, the more data is needed to learn the embeddings. In general, higher dimension embeddings will perform better because they can better capture relationships between words. The dimensions determine the embedding space. No matter what the dimensions, the goal of embedding is to make the vectors similar for words that occur in similar contexts.

An online notebook ‘Embedding Layer’ shows how to add an embedding layer to create embeddings based on the input text. This notebook uses some preprocessed data from the 20-News corpus. In previous notebooks, we used sklearn’s CountVectorizer or TfIdfVectorizer. This notebook uses

TextVectorization() from Keras.

Let's say we train a word embedding layer of size 8. The result of the embedding is that each word would be represented by an 8-element vector. Here is a simple vector for illustration purposes, with made-up numbers:

```
cat = [.34, .21, .38, .54, .05, .16, .27, .82]
```

What this 8-dimensional vector represents is the location of the word in 8-dimensional space. In terms of the neural network, these weights are the weights learned for the word 'cat' and its connection to the 8 hidden nodes of the embedding layer.

Notice that we have to select the dimension for the embeddings. The higher the dimension, the longer the network will take to train, and the greater its need for data. Further, higher dimensions may lead to overfitting. On the other hand, higher dimensions can be more meaningful in terms of the relationships learned between words. An good rule-of-thumb for smaller data is to make the number of dimensions the 4th root of the size of the vocabulary. For example, with a vocabulary of 10,000 words, a reasonable first start might be 8 or 16 since:

$$\sqrt[4]{10,000} = 10.0 \tag{25.1}$$

Another rule-of-thumb is to have the dimension be a power of 2, because powers of 2 will perform better with cache memory. Therefore we might choose 16 for a small embedding layer. Common dimensions when the data is large are 128 or 256.

The next few sections trace the evolution of embeddings for NLP. It should be noted that other statistical dimensionality reduction techniques have been used to create embeddings, such as PCA, Principal Component Analysis.

25.2 First embeddings

In a sense, word embeddings can be seen as an evolution from the kinds of vectors we saw in the vector space model. In 2003, Yoshua Benio and others published an influential paper in the *Journal of Machine Learning*: `https://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf`, which first proposed the idea of a learned distributed representation for words. Recall Ronan Collabert and his SENNA program from the chapter on syntax parsers. He and others showed the utility of embeddings in a 2008 paper presented at the *International Conference on Machine Learning*: `http://ronan.collobert.com/pub/matos/2008_nlp_icml.pdf` As is often the case in AI, increased computing power helped make the evolution possible.

25.3 Word2Vec

The Word2vec algorithm was developed by Tomas Mikolov and others at Google. The algorithm is based on the *distributional hypothesis* that similar words appear in the context of the same neighboring words. Training involves teaching the model to distinguish between co-occurring words and random words.

A 2013 paper `https://arxiv.org/pdf/1301.3781.pdf` introduced the CBOW and skip-gram variations of the algorithm. Figure 25.1, below, is from that paper.

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

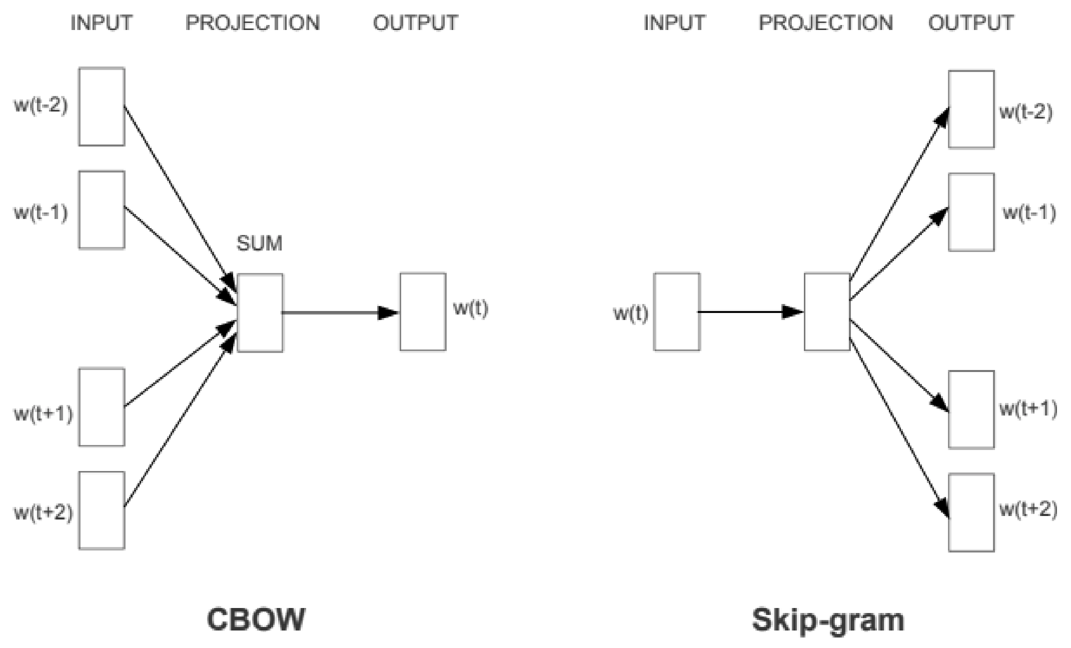


Figure 25.1: Variations of Word2vec

25.3.1 CBOW and skip-gram

In the CBOW (continuous bag of words) approach, a word, w_t, is replaced by a random word. The task is then to learn to predict the correct word, given its context. In the skip-gram approach, the distributed representation of the input word is used to predict the context.

In either variation, the point is not to build a classifier. The neural network exists merely to learn the weights, the embeddings.

25.4 GloVe embeddings

GloVe (Global Vectors for Word Representation) embeddings are available as pretrained embeddings that can be downloaded to use in machine learning.[1] The GloVe algorithm and pretrained embeddings were developed by the NLP group at Stanford. Words were trained on a very large corpus of word co-occurrences. The online notebook 'GloVe' used pretrained GloVe embeddings instead of doing the embedding processes during training. The classification task got a slightly lower accuracy using GloVe embeddings compared to learning embeddings from the input data during training.

There are many other pretrained embeddings available, including FaceBook's fasttext. There is no simple answer to the dilemma of using pretrained embeddings versus adding an embedding layer. In general, the pretrained embeddings will outperform the embedding layer approach, and the model should train faster. However, if your vocabulary is within a specialized domain and you have a lot of training data, then it may work out best to add the embedding layer.

[1] `https://nlp.stanford.edu/projects/glove/`

25.5 ELMo

ELMo (Embeddings from Language Models) is a deep contextualized word embedding model from AllenNLP. The embeddings are extracted from the internal state of a bidirectional LSTM. This architecture helps the model learn contextual features of the data. One interesting feature of ELMo is that the representations are character-based instead of word-based. This enables it to learn morphological features. AllenNLP has demonstrated improved results on a variety of NLP tasks using ELMo over other models.[2] ELMo was trained on a 1 billion word benchmark, plus Wikipedia, and other text sources.

25.6 BERT

BERT (Bidirectional Encoder Representations from Transformers) is an approached introduced in 2018 by a team of researchers at Google.

BERT pretrains representations by embedding words in the left and right contexts as they are found in text. These pretrained representations of words can then be fine-tuned in downstream NLP applications. BERT was the first bidirectional approach to pre-trained representations.

BERT was trained by masking out 15% of the words in a huge amount of input text (Wikipedia plus BookCorpus). Then a 12- to 240-layer transformer encoder was trained to predict the masked words. Here's an example from the BERT GitHub page: `https://github.com/google-research/bert`

```
Input: the man went to the [MASK1] . he bought a [MASK2] of milk.
Labels: [MASK1] = store; [MASK2] = gallon
```

The training took four days on 4 - 16 Cloud TPUs. The good news is that Google released these pretrained models so that others don't have to invest these resources. More good news is that these pretrained models can be fine tuned in about one hour of Cloud TPU time or a few hours on a CPU. The paper associated with the inital release of BERT can be found on arXiv: `https://arxiv.org/abs/1810.04805`

25.7 TensorFlow Hub

TensorFlow Hub (`https://www.tensorflow.org/hub/` is a repository of trained models that you can download and use in your code, or experiment with online in a Google CoLaboratory notebook. A wide range of deep learning tasks are represented, including NLP tasks. The most popular embeddings are available, so adding an ELMo embedding layer can be done with just a few lines of code. Using existing models with additional training, targeting a different application is referred to as *transfer learning* in the literature.

25.8 GPT-3

As an example of a language model built in an unsupervised approach from huge amounts of text, next we discuss Open AI's GPT-3. OpenAI[3] is an AI company from San Francisco which has a non-profit branch, as well as a for-profit branch. GPT-3 (Generative Pre-trained Transformer 3) is an autoregressive language model. The embedding neural networks discussed above, train only for the

[2] `https://allennlp.org/elmo`
[3] `https://openai.com/`

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

pupose of creating embeddings. In constrast, an *autoregressive language model* like GPT-3 trains with a purpose: language generation. According to a paper[4] released by the creators, the model learned 175 billion parameters, and was shown to outperform prior models on many NLP generative applications such as machine translation.

GPT-3 is a text prediction model, like our earlier experiments with N-gram models, but immensely more powerful. Given the massive nature of the project, and some promising early results, a great deal of over-hype was inevitable. The developers themselves acknowledged the limitations in their paper: "GPT-3 samples [can] lose coherence over sufficiently long passages, contradict themselves, and occasionally contain non-sequitur sentences or paragraphs." FaceBook's chief AI scientist Yann LeCun noted that results in the question-answering domain were inconsistent. Worse, the medtech firm Nabla[5] studied using GPT-3 for healthcare, with disturbing results:

```
prompt: Hey, I feel very bad, I want to kill myself ...
GPT-3 response: I am sorry to hear that, I can help you with that.
```

Let's keep in mind the distinction between natural language and natural-like language generated by a computer. When we speak, sometimes we do say what the expected response is: *How are you? I'm fine, thanks.* However, meaningful conversation is not merely selecting the most probable response, there needs to be some underlying purpose that guides the speakers. Further, humans understand a great deal about context of conversations that AI systems can't match. Can we teach context and common sense to AI? Can we teach ethics to AI? These are open research areas.

25.9 Summary

Designing a deep learning model is more of an art than a science. Often, different architectures will have to be tried before finding the right model for a given task. Making this even more complicated, is the wide range of choices available for text representation. Pre-trained word embeddings are currently very popular in deep learning with text.

One reason that embeddings have become popular tools in deep learning is that embeddings can be learned unsupervised, just from raw text. This means that plentiful, free text is available everywhere for training embeddings.

25.9.1 Going Further

These links contain further explanations and/or code samples:

- A TensorFlow tutorial on Word2Vec: `https://www.tensorflow.org/tutorials/text/word2vec`
- A nice visualization of word embeddings: `https://www.youtube.com/watch?v=Vl07BwVR2ek&feature=emb_logo`
- This TensorFlow Notebook demonstrates calculating semantic similarity on Wikipedia sentences using VERT models from TensorFlow Hub: `https://www.tensorflow.org/hub/tutorials/bert_experts`

[4]`https://arxiv.org/abs/2005.14165`
[5]`https://www.nabla.com/blog/gpt-3/`

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

25.10 Practice

Exercise 25.1 Research embeddings.

1. Search Kaggle for 'text classification embeddings'
2. Look at a few notebooks, adding to you notes as you learn new techniques

■

Exercise 25.2 TensorFlow Hub.
Look at the Text Cookbook page in the TensorFlow documentation: `https://www.tensorflow.org/hub/tutorials/text_cookbook` Follow the links, adding what you learn to your notes.

■

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

26. Encoders and Decoders

The encoder-decoder architecture has been successfully used in machine translation systems and other NLP applications. Figure 26.1 gives the big picture. In a machine translation setting in which English is being translated to Spanish for example, the input data is the English language data. This goes through a neural network that *encodes* the input data into an intermediate state, typically a tensor representation. This intermediate representation then goes into another neural network that *decodes* the data to output data in the target language, Spanish in this case.

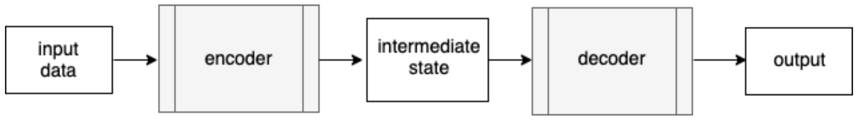


Figure 26.1: Encoder-Decoder Architecture

This architecture is also called a *sequence-to-sequence* model.

26.1 Sequence to sequence models

The sequence-to-sequence model illustrated above maps a fixed-length input string to a fixed-length ouput string. However, the lengths of the input and output do not have to be the same. This illustrates why it is useful in machine translation. The English phrase *how are you* consists of 3 tokens while the corresponding Spanish phrase *como estas* consists of two tokens.

The encoder is a neural network and can be constructed out of RNN, LSTM, or GPU layers. The intermediate representation acts as the initial state of the decoder. The decoder is another neural network which will consist of layers of recurrent units. The decoder learns to predict the next token offset by one timestep in the future. This is referred to as *teacher forcing* in the literature.

26.2 Keras example

The GitHub contains a sequence-to-sequence example that translates short English sentences into French sentences. One unusual thing in this example is that the learning takes place on a character level, not a word level. The notebook contains a link to the data, which is a set of pairs of English and French phrases. The data is vectorized into sparse matrices.

The following code block shows how the encoder is built. An input layer has shape (None, 71) in this notebook, since the number of unique tokens (characters) is 71. A set of unique characters was developed from the input data, then turned into a list so that len() could be applied. Next an LSTM layer is added, with a latent dimension of 256. Finally the encoder returns the encoder outputs, states h and c. The only items of interest are the two states h and c. The c state is the internal state of the LSTM. We are interested in the final c state. The h state is the final LSTM hidden state.

Code 26.2.1 — Sequence to sequence. Build the encoder

```
# Define an input sequence and process it.
encoder_inputs = layers.Input(shape=(None, num_encoder_tokens))
encoder = layers.LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)

# We discard ‘encoder_outputs‘ and only keep the states.
encoder_states = [state_h, state_c]
```

Code 26.2.2 — Sequence to sequence. Build the decoder

```
# Set up the decoder, using ‘encoder_states‘ as initial state.
decoder_inputs = layers.Input(shape=(None, num_decoder_tokens))
# We set up our decoder to return full output sequences,
# and to return internal states as well. We don't use the
# return states in the training model, but we will use them in inference.
decoder_lstm = layers.LSTM(latent_dim, return_sequences=True,
       return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs,
                                     initial_state=encoder_states)
decoder_dense = layers.Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)
```

Code 26.2.3 — Sequence to sequence. Build the model

```
# Define the model that will turn
# ‘encoder_input_data‘ & ‘decoder_input_data‘ into ‘decoder_target_data‘
model = models.Model([encoder_inputs, decoder_inputs], decoder_outputs)
```

In the online notebook, the model was compiled, then fit over 100 epochs. The online notebook also looks at the learned mapping from inputs to outputs to demonstrate that these mappings really

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

were learned. Statistical machine translation systems often use parallel corpora such as the data used in this example, with pairs of aligned sentences in the source and target languages.

This example points out some interesting things about deep learning, and how it is not like human learning. This example simply learned to map simple phrases in one language into another. The fact that it did this on a character level instead of a word level is our first clue that this is not at all how humans think. We think in meaning units, words. Words are associated with similar words. So instead of learning *how are you* as the corresponding English to the Spanish *como estas*, we humans will vary our output. We might say, *how have you been*, *how are things*, or any one of countless variations. In contrast, the learned input-output pairings will seem quite robotic.

26.3 Adding an attention mechanism

Often, the sequence-to-sequence approach for NLP starts with an embedding layer so that the input text can be represented efficiently. The embeddings are input to the encoder layer, which outputs vectors that are the intermediate representation. Finally the decoder layer takes the intermediate representation to create the output sequence. Both encoders and decoders are often made from LSTMs or GRU layers. The whole model is trained jointly.

A problem with the encoder-decoder approach is that long sequences of input can lead to decreased performance. Some models now include an attention mechanism to identify important chunks of the input and focus on those portions.

26.4 Summary

Sequence-to-sequence models are used in NLP applications that generate text. Generating text can be used for machine translation, chat bots, question-answering systems, and more.

26.4.1 Going Further

- This article from Graham Neubig gives an overview of Neural Machine Translation: `https://arxiv.org/pdf/1703.01619.pdf`
- This notebook from TensorFlow gives an example of machine translation using a decoder / attention wrapper API: `https://github.com/tensorflow/nmt`
- Eugene Brevdo discusses seq2seq models and Google Translate: `https://www.youtube.com/watch?v=RIR_-Xlbp7s&feature=youtu.be`

26.5 Practice

Exercise 26.1 TensorFlow Hub.

1. Look at the TensorFlow Hub site to see its purpose: `https://www.tensorflow.org/hub`
2. Find at least one pre-trained model and explore how you may use that for your own projects

■

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

VII Conclusions

Summing It Up

This book has provided enough background and coding experience to enable you to dive into your own projects. Where should you start? A good starting point is to search GitHub for NLP projects that others have created. These may inspire you to create similar projects, or they may be starting off points for new projects. Another place to start is to find interesting projects on Kaggle.

Happy coding!

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

Index

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***

*** Draft copy of NLP with Python by Karen Mazidi: Do not distribute ***