Community Experience Distilled

Mastering Natural Language Processing with Python

Maximize your NLP capabilities while creating amazing NLP projects in Python

Deepti Chopra
Iti Mathur
Nisheeth Joshi

[PACKT] PUBLISHING
open source*
community experience distilled

Mastering Natural Language Processing with Python

Maximize your NLP capabilities while creating amazing NLP projects in Python

Deepti Chopra
Nisheeth Joshi
Iti Mathur

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Mastering Natural Language Processing with Python

First published: June 2016

Production reference: 1030616

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-904-1

www.packtpub.com

Credits

Authors
Deepti Chopra
Nisheeth Joshi
Iti Mathur

Reviewer
Arturo Argueta

Commissioning Editor
Pramila Balan

Acquisition Editor
Tushar Gupta

Content Development Editor
Merwyn D'souza

Technical Editor
Gebin George

Copy Editor
Akshata Lobo

Project Coordinator
Nikhil Nair

Proofreader
Safis Editing

Indexer
Hemangini Bari

Graphics
Jason Monteiro

Production Coordinator
Manu Joseph

Cover Work
Manu Joseph

About the Authors

Deepti Chopra is an Assistant Professor at Banasthali University. Her primary area of research is computational linguistics, Natural Language Processing, and artificial intelligence. She is also involved in the development of MT engines for English to Indian languages. She has several publications in various journals and conferences and also serves on the program committees of several conferences and journals.

Nisheeth Joshi works as an Associate Professor at Banasthali University. His areas of interest include computational linguistics, Natural Language Processing, and artificial intelligence. Besides this, he is also very actively involved in the development of MT engines for English to Indian languages. He is one of the experts empaneled with the TDIL program, Department of Information Technology, Govt. of India, a premier organization that oversees Language Technology Funding and Research in India. He has several publications in various journals and conferences and also serves on the program committees and editorial boards of several conferences and journals.

Iti Mathur is an Assistant Professor at Banasthali University. Her areas of interest are computational semantics and ontological engineering. Besides this, she is also involved in the development of MT engines for English to Indian languages. She is one of the experts empaneled with TDIL program, Department of Electronics and Information Technology (DeitY), Govt. of India, a premier organization that oversees Language Technology Funding and Research in India. She has several publications in various journals and conferences and also serves on the program committees and editorial boards of several conferences and journals.

We acknowledge with gratitude and sincerely thank all our friends and relatives for the blessings conveyed to us to achieve the goal to publishing this Natural Language Processing-based book.

About the Reviewer

Arturo Argueta is currently a PhD student who conducts High Performance Computing and NLP research. Arturo has performed some research on clustering algorithms, machine learning algorithms for NLP, and machine translation. He is also fluent in English, German, and Spanish.

www.PacktPub.com

eBooks, discount offers, and more

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `customercare@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Table of Contents

Preface

In this book, we will learn how to implement various tasks of NLP in Python and gain insight to the current and budding research topics of NLP. This book is a comprehensive step-by-step guide to help students and researchers to create their own projects based on real-life applications.

What this book covers

Chapter 1, *Working with Strings*, explains how to perform preprocessing tasks on text, such as tokenization and normalization, and also explains various string matching measures.

Chapter 2, *Statistical Language Modeling*, covers how to calculate word frequencies and perform various language modeling techniques.

Chapter 3, *Morphology – Getting Our Feet Wet*, talks about how to develop a stemmer, morphological analyzer, and morphological generator.

Chapter 4, *Parts-of-Speech Tagging – Identifying Words*, explains Parts-of-Speech tagging and statistical modeling involving the n-gram approach.

Chapter 5, *Parsing – Analyzing Training Data*, provides information on the concepts of Tree bank construction, CFG construction, the CYK algorithm, the Chart Parsing algorithm, and transliteration.

Chapter 6, *Semantic Analysis – Meaning Matters*, talks about the concept and application of Shallow Semantic Analysis (that is, NER) and WSD using Wordnet.

Chapter 7, Sentiment Analysis – I Am Happy, provides information to help you understand and apply the concepts of sentiment analysis.

Chapter 8, Information Retrieval – Accessing Information, will help you understand and apply the concepts of information retrieval and text summarization.

Chapter 9, Discourse Analysis – Knowing Is Believing, develops a discourse analysis system and anaphora resolution-based system.

Chapter 10, Evaluation of NLP Systems – Analyzing Performance, talks about understanding and applying the concepts of evaluating NLP systems.

What you need for this book

For all the chapters, Python 2.7 or 3.2+ is used. NLTK 3.0 must be installed either on a 32-bit machine or 64-bit machine. The operating system that is required is Windows/Mac/Unix.

Who this book is for

This book is for intermediate level developers in NLP with a reasonable knowledge level and understanding of Python.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "For tokenization of French text, we will use the `french.pickle` file."

A block of code is set as follows:

```
>>> import nltk
>>> text=" Welcome readers. I hope you find it interesting. Please do
reply."
>>> from nltk.tokenize import sent_tokenize
```

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Mastering-Natural-Language-Processing-with-Python`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Working with Strings

Natural Language Processing (**NLP**) is concerned with the interaction between natural language and the computer. It is one of the major components of **Artificial Intelligence** (**AI**) and computational linguistics. It provides a seamless interaction between computers and human beings and gives computers the ability to understand human speech with the help of machine learning. The fundamental data type used to represent the contents of a file or a document in programming languages (for example, C, C++, JAVA, Python, and so on) is known as string. In this chapter, we will explore various operations that can be performed on strings that will be useful to accomplish various NLP tasks.

This chapter will include the following topics:

- Tokenization of text
- Normalization of text
- Substituting and correcting tokens
- Applying Zipf's law to text
- Applying similarity measures using the Edit Distance Algorithm
- Applying similarity measures using Jaccard's Coefficient
- Applying similarity measures using Smith Waterman

Tokenization

Tokenization may be defined as the process of splitting the text into smaller parts called tokens, and is considered a crucial step in NLP.

When NLTK is installed and Python IDLE is running, we can perform the tokenization of text or paragraphs into individual sentences. To perform tokenization, we can import the sentence tokenization function. The argument of this function will be text that needs to be tokenized. The `sent_tokenize` function uses an instance of NLTK known as `PunktSentenceTokenizer`. This instance of NLTK has already been trained to perform tokenization on different European languages on the basis of letters or punctuation that mark the beginning and end of sentences.

Tokenization of text into sentences

Now, let's see how a given text is tokenized into individual sentences:

```
>>> import nltk
>>> text=" Welcome readers. I hope you find it interesting. Please do
reply."
>>> from nltk.tokenize import sent_tokenize
>>> sent_tokenize(text)
[' Welcome readers.', 'I hope you find it interesting.', 'Please do
reply.']
```

So, a given text is split into individual sentences. Further, we can perform processing on the individual sentences.

To tokenize a large number of sentences, we can load `PunktSentenceTokenizer` and use the `tokenize()` function to perform tokenization. This can be seen in the following code:

```
>>> import nltk
>>> tokenizer=nltk.data.load('tokenizers/punkt/english.pickle')
>>> text=" Hello everyone. Hope all are fine and doing well. Hope you
find the book interesting"
>>> tokenizer.tokenize(text)
[' Hello everyone.', 'Hope all are fine and doing well.', 'Hope you
find the book interesting']
```

Tokenization of text in other languages

For performing tokenization in languages other than English, we can load the respective language pickle file found in `tokenizers/punkt` and then tokenize the text in another language, which is an argument of the `tokenize()` function. For the tokenization of French text, we will use the `french.pickle` file as follows:

```
>> import nltk
>>> french_tokenizer=nltk.data.load('tokenizers/punkt/french.pickle')
```

```
>>> french_tokenizer.tokenize('Deux agressions en quelques jours,
voilà ce qui a motivé hier matin le débrayage  collège franco-
britanniquedeLevallois-Perret. Deux agressions en quelques jours,
voilà ce qui a motivé hier matin le débrayage  Levallois. L'équipe
pédagogique de ce collège de 750 élèves avait déjà été choquée
par l'agression, janvier , d'un professeur d'histoire. L'équipe
pédagogique de ce collège de 750 élèves avait déjà été choquée par
l'agression, mercredi , d'un professeur d'histoire')
['Deux agressions en quelques jours, voilà ce qui a motivé hier
matin le débrayage  collège franco-britanniquedeLevallois-Perret.',
'Deux agressions en quelques jours, voilà ce qui a motivé hier matin
le débrayage  Levallois.', 'L'équipe pédagogique de ce collège de
750 élèves avait déjà été choquée par l'agression, janvier , d'un
professeur d'histoire.', 'L'équipe pédagogique de ce collège de
750 élèves avait déjà été choquée par l'agression, mercredi , d'un
professeur d'histoire']
```

Tokenization of sentences into words

Now, we'll perform processing on individual sentences. Individual sentences are tokenized into words. Word tokenization is performed using a `word_tokenize()` function. The `word_tokenize` function uses an instance of NLTK known as `TreebankWordTokenizer` to perform word tokenization.

The tokenization of English text using `word_tokenize` is shown here:

```
>>> import nltk
>>> text=nltk.word_tokenize("PierreVinken , 59 years old , will join
as a nonexecutive director on Nov. 29 .»)
>>> print(text)
[' PierreVinken', ',', '59', ' years', ' old', ',', 'will', 'join',
'as', 'a', 'nonexecutive', 'director' , 'on', 'Nov.', '29', '.']
```

Tokenization of words can also be done by loading `TreebankWordTokenizer` and then calling the `tokenize()` function, whose argument is a sentence that needs to be tokenized into words. This instance of NLTK has already been trained to perform the tokenization of sentence into words on the basis of spaces and punctuation.

The following code will help us obtain user input, tokenize it, and evaluate its length:

```
>>> import nltk
>>> from nltk import word_tokenize
>>> r=input("Please write a text")
Please write a textToday is a pleasant day
>>> print("The length of text is",len(word_tokenize(r)),"words")
The length of text is 5 words
```

Tokenization using TreebankWordTokenizer

Let's have a look at the code that performs tokenization using `TreebankWordTokenizer`:

```
>>> import nltk
>>> from nltk.tokenize import TreebankWordTokenizer
>>> tokenizer = TreebankWordTokenizer()
>>> tokenizer.tokenize("Have a nice day. I hope you find the book
interesting")
['Have', 'a', 'nice', 'day.', 'I', 'hope', 'you', 'find', 'the',
'book', 'interesting']
```

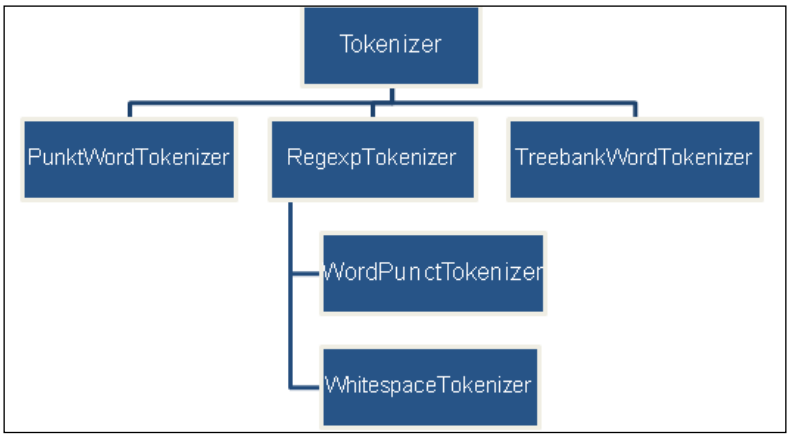
`TreebankWordTokenizer` uses conventions according to Penn Treebank Corpus. It works by separating contractions. This is shown here:

```
>>> import nltk
>>> text=nltk.word_tokenize(" Don't hesitate to ask questions")
>>> print(text)
['Do', "n't", 'hesitate', 'to', 'ask', 'questions']
```

Another word tokenizer is `PunktWordTokenizer`. It works by splitting punctuation; each word is kept instead of creating an entirely new token. Another word tokenizer is `WordPunctTokenizer`. It provides splitting by making punctuation an entirely new token. This type of splitting is usually desirable:

```
>>> from nltk.tokenize import WordPunctTokenizer
>>> tokenizer=WordPunctTokenizer()
>>> tokenizer.tokenize(" Don't hesitate to ask questions")
['Don', "'", 't', 'hesitate', 'to', 'ask', 'questions']
```

The inheritance tree for tokenizers is given here:



Tokenization using regular expressions

The tokenization of words can be performed by constructing **regular expressions** in these two ways:

- By matching with words
- By matching spaces or gaps

We can import `RegexpTokenizer` from NLTK. We can create a Regular Expression that can match the tokens present in the text:

```
>>> import nltk
>>> from nltk.tokenize import RegexpTokenizer
>>> tokenizer=RegexpTokenizer([\w]+")
>>> tokenizer.tokenize("Don't hesitate to ask questions")
["Don't", 'hesitate', 'to', 'ask', 'questions']
```

Instead of instantiating class, an alternative way of tokenization would be to use this function:

```
>>> import nltk
>>> from nltk.tokenize import regexp_tokenize
>>> sent="Don't hesitate to ask questions"
>>> print(regexp_tokenize(sent, pattern='\w+|\$[\d\.]+|\S+'))
['Don', "'t", 'hesitate', 'to', 'ask', 'questions']
```

`RegularexpTokenizer` uses the `re.findall()` function to perform tokenization by matching tokens. It uses the `re.split()` function to perform tokenization by matching gaps or spaces.

Let's have a look at an example of how to tokenize using whitespaces:

```
>>> import nltk
>>> from nltk.tokenize import RegexpTokenizer
>>> tokenizer=RegexpTokenizer('\s+',gaps=True)
>>> tokenizer.tokenize("Don't hesitate to ask questions")
["Don't", 'hesitate', 'to', 'ask', 'questions']
```

To select the words starting with a capital letter, the following code is used:

```
>>> import nltk
>>> from nltk.tokenize import RegexpTokenizer
>>> sent=" She secured 90.56 % in class X . She is a meritorious
student"
>>> capt = RegexpTokenizer('[A-Z]\w+')
>>> capt.tokenize(sent)
['She', 'She']
```

The following code shows how a predefined Regular Expression is used by a subclass of `RegexpTokenizer`:

```
>>> import nltk
>>> sent=" She secured 90.56 % in class X . She is a meritorious
student"
>>> from nltk.tokenize import BlanklineTokenizer
>>> BlanklineTokenizer().tokenize(sent)
[' She secured 90.56 % in class X \n. She is a meritorious student\n']
```

The tokenization of strings can be done using whitespace—tab, space, or newline:

```
>>> import nltk
>>> sent=" She secured 90.56 % in class X . She is a meritorious
student"
>>> from nltk.tokenize import WhitespaceTokenizer
>>> WhitespaceTokenizer().tokenize(sent)
['She', 'secured', '90.56', '%', 'in', 'class', 'X', '.', 'She', 'is',
'a', 'meritorious', 'student']
```

`WordPunctTokenizer` makes use of the regular expression `\w+|[^\w\s]+` to perform the tokenization of text into alphabetic and non-alphabetic characters.

Tokenization using the `split()` method is depicted in the following code:

```
>>>import nltk
>>>sent= She secured 90.56 % in class X. She is a meritorious student"
>>> sent.split()
['She', 'secured', '90.56', '%', 'in', 'class', 'X', '.', 'She', 'is',
'a', 'meritorious', 'student']
>>> sent.split('')
['', 'She', 'secured', '90.56', '%', 'in', 'class', 'X', '.', 'She',
'is', 'a', 'meritorious', 'student']
>>> sent=" She secured 90.56 % in class X \n. She is a meritorious
student\n"
>>> sent.split('\n')
[' She secured 90.56 % in class X ', '. She is a meritorious student',
'']
```

Similar to `sent.split('\n')`, `LineTokenizer` works by tokenizing text into lines:

```
>>> import nltk
>>> from nltk.tokenize import BlanklineTokenizer
>>> sent=" She secured 90.56 % in class X \n. She is a meritorious
student\n"
>>> BlanklineTokenizer().tokenize(sent)
[' She secured 90.56 % in class X \n. She is a meritorious student\n']
```

```
>>> from nltk.tokenize import LineTokenizer
>>> LineTokenizer(blanklines='keep').tokenize(sent)
[' She secured 90.56 % in class X ', '. She is a meritorious student']
>>> LineTokenizer(blanklines='discard').tokenize(sent)
[' She secured 90.56 % in class X ', '. She is a meritorious student']
```

`SpaceTokenizer` works similar to `sent.split('')`:

```
>>> import nltk
>>> sent=" She secured 90.56 % in class X \n. She is a meritorious
student\n"
>>> from nltk.tokenize import SpaceTokenizer
>>> SpaceTokenizer().tokenize(sent)
['', 'She', 'secured', '90.56', '%', 'in', 'class', 'X', '\n.', 'She',
'is', 'a', 'meritorious', 'student\n']
```

`nltk.tokenize.util` module works by returning the sequence of tuples that are offsets of the tokens in a sentence:

```
>>> import nltk
>>> from nltk.tokenize import WhitespaceTokenizer
>>> sent=" She secured 90.56 % in class X \n. She is a meritorious
student\n"
>>> list(WhitespaceTokenizer().span_tokenize(sent))
[(1, 4), (5, 12), (13, 18), (19, 20), (21, 23), (24, 29), (30, 31),
(33, 34), (35, 38), (39, 41), (42, 43), (44, 55), (56, 63)]
```

Given a sequence of spans, the sequence of relative spans can be returned:

```
>>> import nltk
>>> from nltk.tokenize import WhitespaceTokenizer
>>> from nltk.tokenize.util import spans_to_relative
>>> sent=" She secured 90.56 % in class X \n. She is a meritorious
student\n"
>>>list(spans_to_relative(WhitespaceTokenizer().span_tokenize(sent)))
[(1, 3), (1, 7), (1, 5), (1, 1), (1, 2), (1, 5), (1, 1), (2, 1), (1,
3), (1, 2), (1, 1), (1, 11), (1, 7)]
```

`nltk.tokenize.util.string_span_tokenize(sent,separator)` will return the offsets of tokens in sent by splitting at each incidence of the separator:

```
>>> import nltk
>>> from nltk.tokenize.util import string_span_tokenize
>>> sent=" She secured 90.56 % in class X \n. She is a meritorious
student\n"
>>> list(string_span_tokenize(sent, ""))
[(1, 4), (5, 12), (13, 18), (19, 20), (21, 23), (24, 29), (30, 31),
(32, 34), (35, 38), (39, 41), (42, 43), (44, 55), (56, 64)]
```

Normalization

In order to carry out processing on natural language text, we need to perform normalization that mainly involves eliminating punctuation, converting the entire text into lowercase or uppercase, converting numbers into words, expanding abbreviations, canonicalization of text, and so on.

Eliminating punctuation

Sometimes, while tokenizing, it is desirable to remove punctuation. Removal of punctuation is considered one of the primary tasks while doing normalization in NLTK.

Consider the following example:

```
>>> text=[" It is a pleasant evening.","Guests, who came from US
arrived at the venue","Food was tasty."]
>>> from nltk.tokenize import word_tokenize
>>> tokenized_docs=[word_tokenize(doc) for doc in text]
>>> print(tokenized_docs)
[['It', 'is', 'a', 'pleasant', 'evening', '.'], ['Guests', ',', 'who',
'came', 'from', 'US', 'arrived', 'at', 'the', 'venue'], ['Food',
'was', 'tasty', '.']]
```

The preceding code obtains the tokenized text. The following code will remove punctuation from tokenized text:

```
>>> import re
>>> import string
>>> text=[" It is a pleasant evening.","Guests, who came from US
arrived at the venue","Food was tasty."]
>>> from nltk.tokenize import word_tokenize
>>> tokenized_docs=[word_tokenize(doc) for doc in text]
>>> x=re.compile('[%s]' % re.escape(string.punctuation))
>>> tokenized_docs_no_punctuation = []
>>> for review in tokenized_docs:
    new_review = []
    for token in review:
    new_token = x.sub(u'', token)
    if not new_token == u'':
            new_review.append(new_token)
    tokenized_docs_no_punctuation.append(new_review)
>>> print(tokenized_docs_no_punctuation)
[['It', 'is', 'a', 'pleasant', 'evening'], ['Guests', 'who', 'came',
'from', 'US', 'arrived', 'at', 'the', 'venue'], ['Food', 'was',
'tasty']]
```

Conversion into lowercase and uppercase

A given text can be converted into lowercase or uppercase text entirely using the functions `lower()` and `upper()`. The task of converting text into uppercase or lowercase falls under the category of normalization.

Consider the following example of case conversion:

```
>>> text='HARdWork IS KEy to SUCCESS'
>>> print(text.lower())
hardwork is key to success
>>> print(text.upper())
HARDWORK IS KEY TO SUCCESS
```

Dealing with stop words

Stop words are words that need to be filtered out during the task of information retrieval or other natural language tasks, as these words do not contribute much to the overall meaning of the sentence. There are many search engines that work by deleting stop words so as to reduce the search space. Elimination of `stopwords` is considered one of the normalization tasks that is crucial in NLP.

NLTK has a list of stop words for many languages. We need to unzip `datafile` so that the list of stop words can be accessed from `nltk_data/corpora/stopwords/`:

```
>>> import nltk
>>> from nltk.corpus import stopwords
>>> stops=set(stopwords.words('english'))
>>> words=["Don't", 'hesitate','to','ask','questions']
>>> [word for word in words if word not in stops]
["Don't", 'hesitate', 'ask', 'questions']
```

The instance of `nltk.corpus.reader.WordListCorpusReader` is a `stopwords` corpus. It has the `words()` function, whose argument is `fileid`. Here, it is English; this refers to all the stop words present in the English file. If the `words()` function has no argument, then it will refer to all the stop words of all the languages.

Other languages in which stop word removal can be done, or the number of languages whose file of stop words is present in NLTK can be found using the `fileids()` function:

```
>>> stopwords.fileids()
['danish', 'dutch', 'english', 'finnish', 'french', 'german',
'hungarian', 'italian', 'norwegian', 'portuguese', 'russian',
'spanish', 'swedish', 'turkish']
```

Any of these previously listed languages can be used as an argument to the `words()` function so as to get the stop words in that language.

Calculate stopwords in English

Let's see an example of how to calculate `stopwords`:

```
>>> import nltk
>>> from nltk.corpus import stopwords
>>> stopwords.words('english')
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you',
'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his',
'himself', 'she', 'her', 'hers', 'herself', 'it', 'its', 'itself',
'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which',
'who', 'whom', 'this', 'that', 'these', 'those', 'am', 'is', 'are',
'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having',
'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if',
'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for',
'with', 'about', 'against', 'between', 'into', 'through', 'during',
'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in',
'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then',
'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any',
'both', 'each', 'few', 'more', 'most', 'other', 'some', 'such', 'no',
'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's',
't', 'can', 'will', 'just', 'don', 'should', 'now']
>>> def para_fraction(text):
stopwords = nltk.corpus.stopwords.words('english')
para = [w for w in text if w.lower() not in stopwords]
return len(para) / len(text)

>>> para_fraction(nltk.corpus.reuters.words())
0.7364374824583169

>>> para_fraction(nltk.corpus.inaugural.words())
0.5229560503653893
```

Normalization may also involve converting numbers into words (for example, `1` can be replaced by `one`) and expanding abbreviations (for instance, `can't` can be replaced by `cannot`). This can be achieved by representing them in replacement patterns. This is discussed in the next section.

Substituting and correcting tokens

In this section, we will discuss the replacement of tokens with other tokens. We will also about how we can correct the spelling of tokens by replacing incorrectly spelled tokens with correctly spelled tokens.

Replacing words using regular expressions

In order to remove errors or perform text normalization, word replacement is done. One way by which text replacement is done is by using regular expressions. Previously, we faced problems while performing tokenization for contractions. Using text replacement, we can replace contractions with their expanded versions. For example, `doesn't` can be replaced by `does not`.

We will begin by writing the following code, naming this program `replacers.py`, and saving it in the `nltkdata` folder:

```
import re
replacement_patterns = [
(r'won\'t', 'will not'),
(r'can\'t', 'cannot'),
(r'i\'m', 'i am'),
(r'ain\'t', 'is not'),
(r'(\w+)\'ll', '\g<1> will'),
(r'(\w+)n\'t', '\g<1> not'),
(r'(\w+)\'ve', '\g<1> have'),
(r'(\w+)\'s', '\g<1> is'),
(r'(\w+)\'re', '\g<1> are'),
(r'(\w+)\'d', '\g<1> would')
]
class RegexpReplacer(object):
    def __init__(self, patterns=replacement_patterns):
        self.patterns = [(re.compile(regex), repl) for (regex, repl)
in
        patterns]
    def replace(self, text):
        s = text
        for (pattern, repl) in self.patterns:
            (s, count) = re.subn(pattern, repl, s)
        return s
```

Here, replacement patterns are defined in which the first term denotes the pattern to be matched and the second term is its corresponding replacement pattern. The `RegexpReplacer` class has been defined to perform the task of compiling pattern pairs and it provides a method called `replace()`, whose function is to perform the replacement of a pattern with another pattern.

Example of the replacement of a text with another text

Let's see an example of how we can substitute a text with another text:

```
>>> import nltk
>>> from replacers import RegexpReplacer
>>> replacer= RegexpReplacer()
>>> replacer.replace("Don't hesitate to ask questions")
'Do not hesitate to ask questions'
>>> replacer.replace("She must've gone to the market but she didn't
go")
'She must have gone to the market but she did not go'
```

The function of `RegexpReplacer.replace()` is substituting every instance of a replacement pattern with its corresponding substitution pattern. Here, `must've` is replaced by `must have` and `didn't` is replaced by `did not`, since the replacement pattern in `replacers.py` has already been defined by tuple pairs, that is, `(r'(\w+)\'ve', '\g<1> have')` and `(r'(\w+)n\'t', '\g<1> not')`.

We can not only perform the replacement of contractions; we can also substitute a token with any other token.

Performing substitution before tokenization

Tokens substitution can be performed prior to tokenization so as to avoid the problem that occurs during tokenization for contractions:

```
>>> import nltk
>>> from nltk.tokenize import word_tokenize
>>> from replacers import RegexpReplacer
>>> replacer=RegexpReplacer()
>>> word_tokenize("Don't hesitate to ask questions")
['Do', "n't", 'hesitate', 'to', 'ask', 'questions']
>>> word_tokenize(replacer.replace("Don't hesitate to ask questions"))
['Do', 'not', 'hesitate', 'to', 'ask', 'questions']
```

Dealing with repeating characters

Sometimes, people write words involving repeating characters that cause grammatical errors. For instance consider a sentence, `I like it lotttttt`. Here, `lotttttt` refers to `lot`. So now, we'll eliminate these repeating characters using the backreference approach, in which a character refers to the previous characters in a group in a regular expression. This is also considered one of the normalization tasks.

Firstly, append the following code to the previously created `replacers.py`:

```
class RepeatReplacer(object):
    def __init__(self):
        self.repeat_regexp = re.compile(r'(\w*)(\w)\2(\w*)')
        self.repl = r'\1\2\3'
    def replace(self, word):
        repl_word = self.repeat_regexp.sub(self.repl, word)
        if repl_word != word:
            return self.replace(repl_word)
        else:
            return repl_word
```

Example of deleting repeating characters

Let's see an example of how we can delete repeating characters from a token:

```
>>> import nltk
>>> from replacers import RepeatReplacer
>>> replacer=RepeatReplacer()
>>> replacer.replace('lotttt')
'lot'
>>> replacer.replace('ohhhhh')
'oh'
>>> replacer.replace('ooohhhhh')
'oh'
```

The `RepeatReplacer` class works by compiling regular expressions and replacement strings and is defined using `backreference.Repeat_regexp`, which is present in `replacers.py`. It matches the starting characters that can be zero or many `(\w*)`, ending characters that can be zero or many `(\w*)`, or a character `(\w)` that is followed by same character.

For example, `lotttt` is split into `(lo)(t)t(tt)`. Here, one `t` is reduced and the string becomes lottt. The process of splitting continues, and finally, the resultant string obtained is lot.

The problem with `RepeatReplacer` is that it will convert `happy` to `hapy`, which is inappropriate. To avoid this problem, we can embed `wordnet` along with it.

In the `replacers.py` program created previously, add the following lines to include `wordnet`:

```
import re
from nltk.corpus import wordnet
```

```
class RepeatReplacer(object):
    def __init__(self):
        self.repeat_regexp = re.compile(r'(\w*)(\w)\2(\w*)')
        self.repl = r'\1\2\3'
    def replace(self, word):
        if wordnet.synsets(word):
            return word
        repl_word = self.repeat_regexp.sub(self.repl, word)
        if repl_word != word:
            return self.replace(repl_word)
        else:
            return repl_word
```

Now, let's take a look at how the previously mentioned problem can be overcome:

```
>>> import nltk
>>> from replacers import RepeatReplacer
>>> replacer=RepeatReplacer()
>>> replacer.replace('happy')
'happy'
```

Replacing a word with its synonym

Now we will see how we can substitute a given word by its synonym. To the already existing `replacers.py`, we can add a class called `WordReplacer` that provides mapping between a word and its synonym:

```
class WordReplacer(object):
    def __init__(self, word_map):
        self.word_map = word_map
    def replace(self, word):
        return self.word_map.get(word, word)
```

Example of substituting word a with its synonym

Let's have a look at an example of substituting a word with its synonym:

```
>>> import nltk
>>> from replacers import WordReplacer
>>> replacer=WordReplacer({'congrats':'congratulations'})
>>> replacer.replace('congrats')
'congratulations'
>>> replacer.replace('maths')
'maths'
```

In this code, the `replace()` function looks for the corresponding synonym for a word in `word_map`. If the synonym is present for a given word, then the word will be replaced by its synonym. If the synonym for a given word is not present, then no replacement will be performed; the same word will be returned.

Applying Zipf's law to text

Zipf's law states that the frequency of a token in a text is directly proportional to its rank or position in the sorted list. This law describes how tokens are distributed in languages: some tokens occur very frequently, some occur with intermediate frequency, and some tokens rarely occur.

Let's see the code for obtaining the log-log plot in NLTK that is based on Zipf's law:

```
>>> import nltk
>>> from nltk.corpus import gutenberg
>>> from nltk.probability import FreqDist
>>> import matplotlib
>>> import matplotlib.pyplot as plt
>>> matplotlib.use('TkAgg')
>>> fd = FreqDist()
>>> for text in gutenberg.fileids():
. . . for word in gutenberg.words(text):
. . . fd.inc(word)
>>> ranks = []
>>> freqs = []
>>> for rank, word in enumerate(fd):
. . . ranks.append(rank+1)
. . . freqs.append(fd[word])
. . .
>>> plt.loglog(ranks, freqs)
>>> plt.xlabel('frequency(f)', fontsize=14, fontweight='bold')
>>> plt.ylabel('rank(r)', fontsize=14, fontweight='bold')
>>> plt.grid(True)
>>> plt.show()
```

The preceding code will obtain a plot of rank versus the frequency of words in a document. So, we can check whether Zipf's law holds for all the documents or not by seeing the proportionality relationship between rank and the frequency of words.

Similarity measures

There are many similarity measures that can be used for performing NLP tasks. The `nltk.metrics` package in NLTK is used to provide various evaluation or similarity measures, which is conducive to perform various NLP tasks.

In order to test the performance of taggers, chunkers, and so on, in NLP, the standard scores retrieved from information retrieval can be used.

Let's have a look at how the output of named entity recognizer can be analyzed using the standard scores obtained from a training file:

```
>>> from __future__ import print_function
>>> from nltk.metrics import *
>>> training='PERSON OTHER PERSON OTHER OTHER ORGANIZATION'.split()
>>> testing='PERSON OTHER OTHER OTHER OTHER OTHER'.split()
>>> print(accuracy(training,testing))
0.6666666666666666
>>> trainset=set(training)
>>> testset=set(testing)
>>> precision(trainset,testset)
1.0
>>> print(recall(trainset,testset))
0.6666666666666666
>>> print(f_measure(trainset,testset))
0.8
```

Applying similarity measures using Ethe edit distance algorithm

Edit distance or the Levenshtein edit distance between two strings is used to compute the number of characters that can be inserted, substituted, or deleted in order to make two strings equal.

The operations performed in Edit Distance include the following:

- Copying letters from the first string to the second string (cost 0) and substituting a letter with another (cost 1):

  D(i-1,j-1) + d(si,tj)(Substitution / copy)

- Deleting a letter in the first string (cost 1)

  D(i,j-1)+1 (deletion)

- Inserting a letter in the second string (cost 1):
  D(i,j) = min D(i-1,j)+1 (insertion)

The Python code for Edit Distance that is included in the `nltk.metrics` package is as follows:

```
from __future__ import print_function
def _edit_dist_init(len1, len2):
    lev = []
    for i in range(len1):
        lev.append([0] * len2)  # initialize 2D array to zero
    for i in range(len1):
        lev[i][0] = i           # column 0: 0,1,2,3,4,...
    for j in range(len2):
        lev[0][j] = j           # row 0: 0,1,2,3,4,...
    return lev

def_edit_dist_step(lev,i,j,s1,s2,transpositions=False):
c1 =s1[i-1]
c2 =s2[j-1]

# skipping a character in s1
a =lev[i-1][j] +1
# skipping a character in s2
b =lev[i][j -1]+1
# substitution
c =lev[i-1][j-1]+(c1!=c2)
# transposition
d =c+1 # never picked by default
if transpositions and i>1 and j>1:
if s1[i -2]==c2 and s2[j -2]==c1:
d =lev[i-2][j-2]+1
# pick the cheapest
lev[i][j] =min(a,b,c,d)

def edit_distance(s1, s2, transpositions=False):
    # set up a 2-D array
    len1 = len(s1)
    len2 = len(s2)
    lev = _edit_dist_init(len1 + 1, len2 + 1)

    # iterate over the array
    for i in range(len1):
```

```
            for j in range(len2):
                _edit_dist_step(lev, i + 1, j + 1, s1, s2,
transpositions=transpositions)
    return lev[len1][len2]
```

Let's have a look at the Edit Distance calculated in NLTK using the `nltk.metrics` package:

```
>>> import nltk
>>> from nltk.metrics import *
>>> edit_distance("relate","relation")
3
>>> edit_distance("suggestion","calculation")
7
```

Here, when we calculate the edit distance between `relate` and `relation`, three operations (one substitution and two insertions) are performed. While calculating the edit distance between `suggestion` and `calculation`, seven operations (six substitutions and one insertion) are performed.

Applying similarity measures using Jaccard's Coefficient

Jaccard's coefficient, or Tanimoto coefficient, may be defined as a measure of the overlap of two sets, X and Y.

It may be defined as follows:

- $Jaccard(X,Y) = |X \cap Y| / |X \cup Y|$
- $Jaccard(X,X) = 1$
- $Jaccard(X,Y) = 0 \text{ if } X \cap Y = 0$

The code for Jaccard's similarity may be given as follows:

```
def jacc_similarity(query, document):
first=set(query).intersection(set(document))
second=set(query).union(set(document))
return len(first)/len(second)
```

Let's have a look at the implementation of Jaccard's similarity coefficient using NLTK:

```
>>> import nltk
>>> from nltk.metrics import *
```

```
>>> X=set([10,20,30,40])
>>> Y=set([20,30,60])
>>> print(jaccard_distance(X,Y))
0.6
```

Applying similarity measures using the Smith Waterman distance

The Smith Waterman distance is similar to edit distance. This similarity metric was developed in order to detect the optical alignments between related protein sequences and DNA. It consists of costs to be assigned to and functions for alphabet mapping to cost values (substitution); cost is also assigned to gap G (insertion or deletion):

1. *0 //*start over
2. *D(i-1,j-1) -d(si,tj) //*subst/copy
3. *D(i,j) = max D(i-1,j)-G //*insert
1. *D(i,j-1)-G //*delete

Distance is maximum over all *i,j* in table of *D(i,j)*

4. *G = 1 //*example value for gap
5. *d(c,c) = -2 //*context dependent substitution cost
6. *d(c,d) = +1 //*context dependent substitution cost

Similar to Edit distance, the Python code for Smith Waterman can be embedded with the `nltk.metrics` package to perform string similarity using Smith Waterman in NLTK.

Other string similarity metrics

Binary distance is a string similarity metric. It returns the value `0.0` if two labels are identical; otherwise, it returns the value `1.0`.

The Python code for Binary distance metrics is:

```
def binary_distance(label1, label2):
 return 0.0 if label1 == label2 else 1.0
```

Let's see how Binary distance metrics is implemented in NLTK:

```
>>> import nltk
>>> from nltk.metrics import *
>>> X = set([10,20,30,40])
>>> Y= set([30,50,70])
>>> binary_distance(X, Y)
1.0
```

Masi distance is based on partial agreement when multiple labels are present.

The Python code included in `nltk.metrics` for `masi` distance is as follows:

```
def masi_distance(label1, label2):
    len_intersection = len(label1.intersection(label2))
    len_union = len(label1.union(label2))
    len_label1 = len(label1)
    len_label2 = len(label2)
    if len_label1 == len_label2 and len_label1 == len_intersection:
        m = 1
    elif len_intersection == min(len_label1, len_label2):
        m = 0.67
    elif len_intersection > 0:
        m = 0.33
    else:
        m = 0

    return 1 - (len_intersection / float(len_union)) * m
```

Let's see the implementation of `masi` distance in NLTK:

```
>>> import nltk
>>> from __future__ import print_function
>>> from nltk.metrics import *
>>> X = set([10,20,30,40])
>>> Y= set([30,50,70])
>>> print(masi_distance(X,Y))
0.945
```

Summary

In this chapter, you have learned various operations that can be performed on a text that is a collection of strings. You have understood the concept of tokenization, substitution, and normalization, and applied various similarity measures to strings using NLTK. We have also discussed Zipf's law, which may be applicable to some of the existing documents.

In the next chapter, we'll discuss various language modeling techniques and different NLP tasks.

2
Statistical Language Modeling

Computational linguistics is an emerging field that is widely used in analytics, software applications, and contexts where people communicate with machines. Computational linguistics may be defined as a subfield of artificial intelligence. Applications of computational linguistics include machine translation, speech recognition, intelligent Web searching, information retrieval, and intelligent spelling checkers. It is important to understand the preprocessing tasks or the computations that can be performed on natural language text. In the following chapter, we will discuss ways to calculate word frequencies, the **Maximum Likelihood Estimation** (**MLE**) model, interpolation on data, and so on. But first, let's go through the various topics that we will cover in this chapter. They are as follows:

- Calculating word frequencies (1-gram, 2-gram, 3-gram)
- Developing MLE for a given text
- Applying smoothing on the MLE model
- Developing a back-off mechanism for MLE
- Applying interpolation on data to get a mix and match
- Evaluating a language model through perplexity
- Applying Metropolis-Hastings in modeling languages
- Applying Gibbs sampling in language processing

Understanding word frequency

Collocations may be defined as the collection of two or more tokens that tend to exist together. For example, the United States, the United Kingdom, Union of Soviet Socialist Republics, and so on.

Unigram represents a single token. The following code will be used for generate `unigrams` for Alpino Corpus:

```
>>> import nltk
>>> from nltk.util import ngrams
>>> from nltk.corpus import alpino
>>> alpino.words()
['De', 'verzekeringsmaatschappijen', 'verhelen', ...]>>>
unigrams=ngrams(alpino.words(),1)
>>> for i in unigrams:
print(i)
```

Consider another example for generating `quadgrams` or `fourgrams` from `alpinocorpus`:

```
>>> import nltk
>>> from nltk.util import ngrams
>>> from nltk.corpus import alpino
>>> alpino.words()
['De', 'verzekeringsmaatschappijen', 'verhelen', ...]
>>> quadgrams=ngrams(alpino.words(),4)
>>> for i in quadgrams:
print(i)
```

`bigram` refers to a pair of tokens. To find bigrams in the text, firstly, lowercased words are searched, a list of lowercased words in the text is created, and `BigramCollocationFinder` is produced. The `BigramAssocMeasures` found in the `nltk.metrics` package can be used to find bigrams in the text:

```
>>> import nltk
>>> from nltk.collocations import BigramCollocationFinder
>>> from nltk.corpus import webtext
>>> from nltk.metrics import BigramAssocMeasures
>>> tokens=[t.lower() for t in webtext.words('grail.txt')]
>>> words=BigramCollocationFinder.from_words(tokens)
>>> words.nbest(BigramAssocMeasures.likelihood_ratio, 10)
[("'", 's'), ('arthur', ':'), ('#', '1'), ("'", 't'), ('villager',
'#'), ('#', '2'), (']', '['), ('1', ':'), ('oh', ','), ('black',
'knight')]
```

In the preceding code, we can add a word filter that can be used to eliminate `stopwords` and punctuation:

```
>>> from nltk.corpus import stopwords
>>> from nltk.corpus import webtext
>>> from nltk.collocations import BigramCollocationFinder
>>> from nltk.metrics import BigramAssocMeasures
```

```
>>> set = set(stopwords.words('english'))
>>> stops_filter = lambda w: len(w) < 3 or w in set
>>> tokens=[t.lower() for t in webtext.words('grail.txt')]
>>> words=BigramCollocationFinder.from_words(tokens)
>>> words.apply_word_filter(stops_filter)
>>> words.nbest(BigramAssocMeasures.likelihood_ratio, 10)
[('black', 'knight'), ('clop', 'clop'), ('head', 'knight'), ('mumble',
'mumble'), ('squeak', 'squeak'), ('saw', 'saw'), ('holy', 'grail'),
('run', 'away'), ('french', 'guard'), ('cartoon', 'character')]
```

Here, we can change the frequency of bigrams from 10 to any other number.

Another way of generating bigrams from a text is using collocation finders. This is given in the following code:

```
>>> import nltk
>>> from nltk.collocation import *
>>> text1="Hardwork is the key to success. Never give up!"
>>> word = nltk.wordpunct_tokenize(text1)
>>> finder = BigramCollocationFinder.from_words(word)
>>> bigram_measures = nltk.collocations.BigramAssocMeasures()
>>> value = finder.score_ngrams(bigram_measures.raw_freq)
>>> sorted(bigram for bigram, score in value)
[('.', 'Never'), ('Hardwork', 'is'), ('Never', 'give'), ('give',
'up'), ('is', 'the'), ('key', 'to'), ('success', '.'), ('the', 'key'),
('to', 'success'), ('up', '!')]
```

We will now see another code for generating bigrams from `alpino corpus`:

```
>>> import nltk
>>> from nltk.util import ngrams
>>> from nltk.corpus import alpino
>>> alpino.words()
['De', 'verzekeringsmaatschappijen', 'verhelen', ...]
>>> bigrams_tokens=ngrams(alpino.words(),2)
>>> for i in bigrams_tokens:
print(i)
```

This code will generate bigrams from `alpino corpus`.

We will now see the code for generating `trigrams`:

```
>>> import nltk
>>> from nltk.util import ngrams
>>> from nltk.corpus import alpino
>>> alpino.words()
```

```
['De', 'verzekeringsmaatschappijen', 'verhelen', ...]>>> trigrams_
tokens=ngrams(alpino.words(),3)
>>> for i in trigrams_tokens:
print(i)
```

For generating `fourgrams` and generating the frequency of `fourgrams`, the following code is used:

```
>>> import nltk
>>> import nltk
>>> from nltk.collocations import *
>>> text="Hello how are you doing ? I hope you find the book
interesting"
>>> tokens=nltk.wordpunct_tokenize(text)
>>> fourgrams=nltk.collocations.QuadgramCollocationFinder.from_
words(tokens)
>>> for fourgram, freq in fourgrams.ngram_fd.items():
print(fourgram,freq)

('hope', 'you', 'find', 'the') 1
('Hello', 'how', 'are', 'you') 1
('you', 'doing', '?', 'I') 1
('are', 'you', 'doing', '?') 1
('how', 'are', 'you', 'doing') 1
('?', 'I', 'hope', 'you') 1
('doing', '?', 'I', 'hope') 1
('find', 'the', 'book', 'interesting') 1
('you', 'find', 'the', 'book') 1
('I', 'hope', 'you', 'find') 1
```

We will now see the code for generating `ngrams` for a given sentence:

```
>>> import nltk
>>> sent=" Hello , please read the book thoroughly . If you have any
queries , then don't hesitate to ask . There is no shortcut to success
."
>>> n=5
>>> fivegrams=ngrams(sent.split(),n)
>>> for grams in fivegrams:
   print(grams)


('Hello', ',', 'please', 'read', 'the')
(',', 'please', 'read', 'the', 'book')
('please', 'read', 'the', 'book', 'thoroughly')
('read', 'the', 'book', 'thoroughly', '.')
```

```
('the', 'book', 'thoroughly', '.', 'If')
('book', 'thoroughly', '.', 'If', 'you')
('thoroughly', '.', 'If', 'you', 'have')
('.', 'If', 'you', 'have', 'any')
('If', 'you', 'have', 'any', 'queries')
('you', 'have', 'any', 'queries', ',')
('have', 'any', 'queries', ',', 'then')
('any', 'queries', ',', 'then', "don't")
('queries', ',', 'then', "don't", 'hesitate')
(',', 'then', "don't", 'hesitate', 'to')
('then', "don't", 'hesitate', 'to', 'ask')
("don't", 'hesitate', 'to', 'ask', '.')
('hesitate', 'to', 'ask', '.', 'There')
('to', 'ask', '.', 'There', 'is')
('ask', '.', 'There', 'is', 'no')
('.', 'There', 'is', 'no', 'shortcut')
('There', 'is', 'no', 'shortcut', 'to')
('is', 'no', 'shortcut', 'to', 'success')
('no', 'shortcut', 'to', 'success', '.')
```

Develop MLE for a given text

MLE, also referred to as multinomial logistic regression or a conditional exponential classifier, is an essential task in the field of NLP. It was first introduced in 1996 by Berger and Della Pietra. Maximum Entropy is defined in NLTK in the `nltk.classify.maxent` module. In this module, all the probability distributions are considered that are in accordance with the training data. This model is used to refer to two features, namely input-feature and joint feature. An input feature may be called the feature of unlabeled words. A joined feature may be called the feature of labeled words. MLE is used to generate `freqdist` that contains the probability distribution for a given occurrence in a text. `param freqdist` consists of frequency distribution on which probability distribution is based.

We'll now see the code for the Maximum Entropy Model in NLTK:

```
from__future__import print_function,unicode_literals
__docformat__='epytext en'

try:
import numpy
except ImportError:
    pass
```

```
import tempfile
import os
from collections import defaultdict
from nltk import compat
from nltk.data import gzip_open_unicode
from nltk.util import OrderedDict
from nltk.probability import DictionaryProbDist
from nltk.classify.api import ClassifierI
from nltk.classify.util import CutoffChecker,accuracy,log_likelihood
from nltk.classify.megam import (call_megam,
write_megam_file,parse_megam_weights)
from nltk.classify.tadm import call_tadm,write_tadm_file,parse_tadm_
weights
```

In the preceding code, `nltk.probability` consists of the `FreqDist` class that can be used to determine the frequency of the occurrence of individual tokens in a text.

The `ProbDistI` is used to determine the probability distribution of individual occurrences in a text. There are basically two kinds of probability distributions: Derived Probability Distribution and Analytic Probability Distribution. Distributed Probability Distributions are obtained from frequency distribution. Analytic Probability Distributions are obtained from parameters, such as variance.

In order to obtain the frequency distribution, the maximum likelihood estimate is used. It computes the probability of every occurrence on the basis of its frequency in the frequency distribution:

```
class MLEProbDist(ProbDistI):

    def __init__(self, freqdist, bins=None):
        self._freqdist = freqdist

    def freqdist(self):
"""
```

It will find the frequency distribution on the basis of probability distribution:

```
"""
    return self._freqdist

    def prob(self, sample):
        return self._freqdist.freq(sample)

    def max(self):
        return self._freqdist.max()
```

```
    def samples(self):
        return self._freqdist.keys()

    def __repr__(self):
"""
        It will return string representation of ProbDist
"""
        return '<MLEProbDist based on %d samples>' % self._
freqdist.N()


class LidstoneProbDist(ProbDistI):
"""
```

It is used to obtain frequency distribution. It is represented by a real number, Gamma, whose range lies between `0` and `1`. The `LidstoneProbDist` calculates the probability of a given observation with count `c`, outcomes `N`, and bins `B` as follows: *(c+Gamma)/(N+B*Gamma)*.

It also means that Gamma is added to the count of each bin and MLE is computed from the given frequency distribution:

```
"""
SUM_TO_ONE = False
    def __init__(self, freqdist, gamma, bins=None):
"""
```

Lidstone is used to compute the probability distribution in order to obtain `freqdist`.

`paramfreqdist` may be defined as the frequency distribution on which probability estimates are based.

`param` bins may be defined as sample values that can be obtained from the probability distribution. The sum of probabilities is equal to one:

```
"""
        if (bins == 0) or (bins is None and freqdist.N() == 0):
            name = self.__class__.__name__[:-8]
            raise ValueError('A %s probability distribution ' % name +
'must have at least one bin.')
        if (bins is not None) and (bins < freqdist.B()):
            name = self.__class__.__name__[:-8]
            raise ValueError('\nThe number of bins in a %s
distribution ' % name +
```

```
'(%d) must be greater than or equal to\n' % bins +
'the number of bins in the FreqDist used ' +
'to create it (%d).' % freqdist.B())

        self._freqdist = freqdist
        self._gamma = float(gamma)
        self._N = self._freqdist.N()

        if bins is None:
            bins = freqdist.B()
        self._bins = bins

        self._divisor = self._N + bins * gamma
        if self._divisor == 0.0:
            # In extreme cases we force the probability to be 0,
            # which it will be, since the count will be 0:
            self._gamma = 0
            self._divisor = 1

def freqdist(self):
"""
```

It obtains frequency distribution, which is based upon the probability distribution:

```
    """
        return self._freqdist

def prob(self, sample):
c = self._freqdist[sample]
        return (c + self._gamma) / self._divisor

   def max(self):
 # To obtain most probable sample, choose the one
# that occurs very frequently.
        return self._freqdist.max()

def samples(self):
        return self._freqdist.keys()

def discount(self):
    gb = self._gamma * self._bins
        return gb / (self._N + gb)

    def __repr__(self):
"""
```

```
        String representation of ProbDist is obtained.


"""
        return '<LidstoneProbDist based on %d samples>' % self._
freqdist.N()


class LaplaceProbDist(LidstoneProbDist):
"""
```

It is used to obtain frequency distribution. It calculates the probability of a sample with count `c`, outcomes `N`, and bins `B` as follows:

(c+1)/(N+B)

It also means that 1 is added to the count of every bin, and the maximum likelihood is estimated for the resultant frequency distribution:

```
"""
    def __init__(self, freqdist, bins=None):
"""
```

`LaplaceProbDist` is used to obtain the probability distribution for generating `freqdist`.

`param freqdist` is used to obtain the frequency distribution, which is based on probability estimates.

`Param` bins may be defined as the frequency of sample values that can be generated. The sum of probabilities must be `1`:

```
"""
        LidstoneProbDist.__init__(self, freqdist, 1, bins)

    def __repr__(self):
"""
        String representation of ProbDist is obtained.
"""
        return '<LaplaceProbDist based on %d samples>' % self._
freqdist.N()

class ELEProbDist(LidstoneProbDist):
"""
```

It is used to obtain frequency distribution. It calculates the probability of a sample with count c, outcomes N, and bins B as follows:

(c+0.5)/(N+B/2)

It also means that `0.5` is added to the count of every bin and the maximum likelihood is estimated for the resultant frequency distribution:

```
"""
    def __init__(self, freqdist, bins=None):
"""
```

The expected likelihood estimation is used to obtain the probability distribution for generating `freqdist.param.freqdist` is used to obtain the frequency distribution, which is based on probability estimates.

`param` bins may be defined as the frequency of sample values that can be generated. The sum of probabilities must be 1:

```
"""
LidstoneProbDist.__init__(self, freqdist, 0.5, bins)

    def __repr__(self):
"""
        String representation of ProbDist is obtained.
    """
        return '<ELEProbDist based on %d samples>' % self._
freqdist.N()



class WittenBellProbDist(ProbDistI):
"""
```

The `WittenBellProbDist` is used to obtain the probability distribution. It is used to obtain the uniform probability mass on the basis of the frequency of the sample seen before. The probability mass for the sample is given as follows:

T / (N + T)

Here, *T* is the number of samples observed and *N* is total number of events observed. It is equal to the maximum likelihood estimate of a new sample that is occurring. The sum of all the probabilities is equal to 1:

```
Here,
      p = T / Z (N + T), if count = 0
      p = c / (N + T), otherwise
```

```
    """

        def __init__(self, freqdist, bins=None):
    """
```

It obtains the probability distribution. This probability is used to provide the uniform probability mass to an unseen sample. The probability mass for the sample is given as follows:

T / (N + T)

Here, *T* is the number of samples observed and *N* is the total number of events observed. It is equal to the maximum likelihood estimate of a new sample that is occurring. The sum of all the probabilities is equal to *1*:

```
        Here,
              p = T / Z (N + T), if count = 0
              p = c / (N + T), otherwise
```

Z is the normalizing factor that is calculated using these values and a bin value.

`Param freqdist` is used to estimate the frequency counts from which the probability distribution is obtained.

`Param` bins may be defined as the number of possible types of samples:

```
    """
            assert bins is None or bins >= freqdist.B(),\
    'bins parameter must not be less than %d=freqdist.B()' % freqdist.B()
            if bins is None:
                bins = freqdist.B()
            self._freqdist = freqdist
            self._T = self._freqdist.B()
            self._Z = bins - self._freqdist.B()
            self._N = self._freqdist.N()
            # self._P0 is P(0), precalculated for efficiency:
            if self._N==0:
                # if freqdist is empty, we approximate P(0) by a
    UniformProbDist:
                self._P0 = 1.0 / self._Z
            else:
                self._P0 = self._T / float(self._Z * (self._N + self._T))

        def prob(self, sample):
            # inherit docs from ProbDistI
            c = self._freqdist[sample]
```

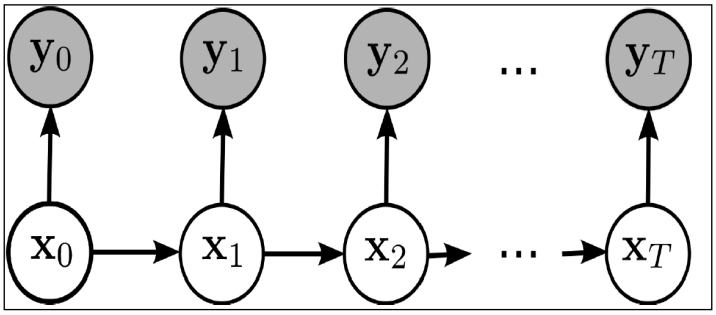
```
            return (c / float(self._N + self._T) if c != 0 else self._P0)

        def max(self):
            return self._freqdist.max()

        def samples(self):
            return self._freqdist.keys()

        def freqdist(self):
            return self._freqdist

        def discount(self):
            raise NotImplementedError()

        def __repr__(self):
    """
            String representation of ProbDist is obtained.


    """
            return '<WittenBellProbDist based on %d samples>' % self._
    freqdist.N()
```

We can perform testing using maximum likelihood estimation. Let's consider the following code for MLE in NLTK:

```
>>> import nltk
>>> from nltk.probability import *
>>> train_and_test(mle)
28.76%
>>> train_and_test(LaplaceProbDist)
69.16%
>>> train_and_test(ELEProbDist)
76.38%
>>> def lidstone(gamma):
    return lambda fd, bins: LidstoneProbDist(fd, gamma, bins)

>>> train_and_test(lidstone(0.1))
86.17%
>>> train_and_test(lidstone(0.5))
76.38%
>>> train_and_test(lidstone(1.0))
69.16%
```

Hidden Markov Model estimation

Hidden Markov Model (**HMM**) comprises of observed states and the latent states that help in determining them. Consider the diagrammatic description of HMM. Here, x represents the latent state and y represents the observed state.



We can perform testing using HMM estimation. Let's consider the `Brown` Corpus and the code given here:

```
>>> import nltk
>>> corpus = nltk.corpus.brown.tagged_sents(categories='adventure')
[:700]
>>> print(len(corpus))
700
>>> from nltk.util import unique_list
>>> tag_set = unique_list(tag for sent in corpus for (word,tag) in
sent)
>>> print(len(tag_set))
104
>>> symbols = unique_list(word for sent in corpus for (word,tag) in
sent)
>>> print(len(symbols))
1908
>>> print(len(tag_set))
104
>>> symbols = unique_list(word for sent in corpus for (word,tag) in
sent)
>>> print(len(symbols))
1908
>>> trainer = nltk.tag.HiddenMarkovModelTrainer(tag_set, symbols)
>>> train_corpus = []
>>> test_corpus = []
>>> for i in range(len(corpus)):
if i % 10:
```

```
train_corpus += [corpus[i]]
else:
test_corpus += [corpus[i]]


>>> print(len(train_corpus))
630
>>> print(len(test_corpus))
70
>>> def train_and_test(est):
hmm = trainer.train_supervised(train_corpus, estimator=est)
print('%.2f%%' % (100 * hmm.evaluate(test_corpus)))
```

In the preceding code, we have created a 90% training and 10% testing file and we have tested the estimator.

Applying smoothing on the MLE model

Smoothing is used to handle the words that have not occurred previously. So, the probability of unknown words is 0. To solve this problem, smoothing is used.

Add-one smoothing

In the 18th century, Laplace invented add-one smoothing. In add-one smoothing, 1 is added to the count of each word. Instead of 1, any other value can also be added to the count of unknown words so that unknown words can be handled and their probability is non-zero. Pseudo count is the value (that is, either 1 or nonzero) that is added to the counts of unknown words to make their probability nonzero.

Let's consider the following code for add-one smoothing in NLTK:

```
>>> import nltk
>>> corpus=u"<s> hello how are you doing ? Hope you find the book
interesting. </s>".split()
>>> sentence=u"<s>how are you doing</s>".split()
>>> vocabulary=set(corpus)
>>> len(vocabulary)
13
>>> cfd = nltk.ConditionalFreqDist(nltk.bigrams(corpus))
>>> # The corpus counts of each bigram in the sentence:
>>> [cfd[a][b] for (a,b) in nltk.bigrams(sentence)]
[0, 1, 0]
>>> # The counts for each word in the sentence:
>>> [cfd[a].N() for (a,b) in nltk.bigrams(sentence)]
```

```
[0, 1, 2]
>>> # There is already a FreqDist method for MLE probability:
>>> [cfd[a].freq(b) for (a,b) in nltk.bigrams(sentence)]
[0, 1.0, 0.0]
>>> # Laplace smoothing of each bigram count:
>>> [1 + cfd[a][b] for (a,b) in nltk.bigrams(sentence)]
[1, 2, 1]
>>> # We need to normalise the counts for each word:
>>> [len(vocabulary) + cfd[a].N() for (a,b) in nltk.bigrams(sentence)]
[13, 14, 15]
>>> # The smoothed Laplace probability for each bigram:
>>> [1.0 * (1+cfd[a][b]) / (len(vocabulary)+cfd[a].N()) for (a,b) in
nltk.bigrams(sentence)]
[0.07692307692307693, 0.14285714285714285, 0.06666666666666667]
```

Consider another way of performing Add-one smoothing or generating a Laplace probability distribution:

```
>>> # MLEProbDist is the unsmoothed probability distribution:
>>> cpd_mle = nltk.ConditionalProbDist(cfd, nltk.MLEProbDist,
bins=len(vocabulary))
>>> # Now we can get the MLE probabilities by using the .prob method:
>>> [cpd_mle[a].prob(b) for (a,b) in nltk.bigrams(sentence)]
[0, 1.0, 0.0]
>>> # LaplaceProbDist is the add-one smoothed ProbDist:
>>> cpd_laplace = nltk.ConditionalProbDist(cfd, nltk.LaplaceProbDist,
bins=len(vocabulary))
>>> # Getting the Laplace probabilities is the same as for MLE:
>>> [cpd_laplace[a].prob(b) for (a,b) in nltk.bigrams(sentence)]
[0.07692307692307693, 0.14285714285714285, 0.06666666666666667]
```

Good Turing

Good Turing was introduced by Alan Turing along with his statistical assistant I.J. Good. It is an efficient smoothing method that increases the performance of statistical techniques performed for linguistic tasks, such as word sense disambiguation (WSD), named entity recognition (NER), spelling correction, machine translation, and so on. This method helps to predict the probability of unseen objects. In this method, binomial distribution is exhibited by our objects of interest. This method is used to compute the mass probability for zero or low count samples on the basis of higher count samples . Simple Good Turing performs approximation from frequency to frequency by linear regression into a linear line in log space. If $c\backslash$ is the adjusted count, it will compute the following:

$c\backslash = (c + 1)\, N(c + 1) / N(c) \quad for\ c >= 1$

The samples with zero frequency in training = *N(1)* for *c == 0*.

Here, *c* is the original count and *N(i)* is the number of event types observed with count `i`.

Bill Gale and Geoffrey Sampson have presented Simple Good Turing:

```
class SimpleGoodTuringProbDist(ProbDistI):
"""


    Given a pair (pi, qi),  where pi refers to the frequency and
    qi refers to the frequency of frequency, our aim is to minimize
the
    square variation. E(p) and E(q) is the mean of pi and qi.

    - slope,  b = sigma ((pi-E(p)(qi-E(q))) / sigma ((pi-E(p))(pi-
E(p)))
    - intercept: a = E(q) - b.E(p)
"""
    SUM_TO_ONE = False
    def __init__(self, freqdist, bins=None):
"""
        param freqdist refers to the count of frequency from which
probability
        distribution is estimated.
        Param bins is used to estimate the possible number of samples.
"""
        assert bins is None or bins > freqdist.B(),\
'bins parameter must not be less than %d=freqdist.B()+1' %
(freqdist.B()+1)
        if bins is None:
            bins = freqdist.B() + 1
        self._freqdist = freqdist
        self._bins = bins
        r, nr = self._r_Nr()
        self.find_best_fit(r, nr)
        self._switch(r, nr)
        self._renormalize(r, nr)

    def _r_Nr_non_zero(self):
        r_Nr = self._freqdist.r_Nr()
        del r_Nr[0]
        return r_Nr
```

```
    def _r_Nr(self):
"""
Split the frequency distribution in two list (r, Nr), where Nr(r) > 0
"""
        nonzero = self._r_Nr_non_zero()

        if not nonzero:
            return [], []
        return zip(*sorted(nonzero.items()))

    def find_best_fit(self, r, nr):
"""
        Use simple linear regression to tune parameters self._slope
and
        self._intercept in the log-log space based on count and
Nr(count)
        (Work in log space to avoid floating point underflow.)
"""
        # For higher sample frequencies the data points becomes
horizontal
        # along line Nr=1. To create a more evident linear model in
log-log
        # space, we average positive Nr values with the surrounding
zero
        # values. (Church and Gale, 1991)

        if not r or not nr:
            # Empty r or nr?
            return

        zr = []
        for j in range(len(r)):
            i = (r[j-1] if j > 0 else 0)
            k = (2 * r[j] - i if j == len(r) - 1 else r[j+1])
            zr_ = 2.0 * nr[j] / (k - i)
            zr.append(zr_)

        log_r = [math.log(i) for i in r]
        log_zr = [math.log(i) for i in zr]

        xy_cov = x_var = 0.0
        x_mean = 1.0 * sum(log_r) / len(log_r)
        y_mean = 1.0 * sum(log_zr) / len(log_zr)
        for (x, y) in zip(log_r, log_zr):
```

```
                xy_cov += (x - x_mean) * (y - y_mean)
                x_var += (x - x_mean)**2
            self._slope = (xy_cov / x_var if x_var != 0 else 0.0)
            if self._slope >= -1:
                warnings.warn('SimpleGoodTuring did not find a proper best
    fit '
    'line for smoothing probabilities of occurrences. '
    'The probability estimates are likely to be '
    'unreliable.')
            self._intercept = y_mean - self._slope * x_mean

        def _switch(self, r, nr):
    """
            Calculate the r frontier where we must switch from Nr to Sr
            when estimating E[Nr].
    """
            for i, r_ in enumerate(r):
                if len(r) == i + 1 or r[i+1] != r_ + 1:
                    # We are at the end of r, or there is a gap in r
                    self._switch_at = r_
                    break

                Sr = self.smoothedNr
                smooth_r_star = (r_ + 1) * Sr(r_+1) / Sr(r_)
                unsmooth_r_star = 1.0 * (r_ + 1) * nr[i+1] / nr[i]

                std = math.sqrt(self._variance(r_, nr[i], nr[i+1]))
                if abs(unsmooth_r_star-smooth_r_star) <= 1.96 * std:
                    self._switch_at = r_
                    break

        def _variance(self, r, nr, nr_1):
            r = float(r)
            nr = float(nr)
            nr_1 = float(nr_1)
            return (r + 1.0)**2 * (nr_1 / nr**2) * (1.0 + nr_1 / nr)

        def _renormalize(self, r, nr):
    """
```

Renormalization is very crucial to ensure that the proper distribution of probability is obtained. It can be obtained by making the probability estimate of an unseen sample *N(1)/N* and then, renormalizing all the previously seen sample probabilities:

```
"""
        prob_cov = 0.0
        for r_, nr_ in zip(r, nr):
            prob_cov  += nr_ * self._prob_measure(r_)
        if prob_cov:
            self._renormal = (1 - self._prob_measure(0)) / prob_cov

   def smoothedNr(self, r):
"""
        Return the number of samples with count r.


"""

        # Nr = a*r^b (with b < -1 to give the appropriate hyperbolic
        # relationship)
        # Estimate a and b by simple linear regression technique on
        # the logarithmic form of the equation: log Nr = a + b*log(r)

        return math.exp(self._intercept + self._slope * math.log(r))

    def prob(self, sample):
"""
        Return the sample's probability.

"""
        count = self._freqdist[sample]
        p = self._prob_measure(count)
        if count == 0:
            if self._bins == self._freqdist.B():
                p = 0.0
            else:
                p = p / (1.0 * self._bins - self._freqdist.B())
        else:
            p = p * self._renormal
        return p

    def _prob_measure(self, count):
        if count == 0 and self._freqdist.N() == 0 :
            return 1.0
        elif count == 0 and self._freqdist.N() != 0:
            return 1.0 * self._freqdist.Nr(1) / self._freqdist.N()
```

```
            if self._switch_at > count:
                Er_1 = 1.0 * self._freqdist.Nr(count+1)
                Er = 1.0 * self._freqdist.Nr(count)
            else:
                Er_1 = self.smoothedNr(count+1)
                Er = self.smoothedNr(count)

            r_star = (count + 1) * Er_1 / Er
            return r_star / self._freqdist.N()

        def check(self):
            prob_sum = 0.0
            for i in  range(0, len(self._Nr)):
                prob_sum += self._Nr[i] * self._prob_measure(i) / self._
    renormal
            print("Probability Sum:", prob_sum)
            #assert prob_sum != 1.0, "probability sum should be one!"

        def discount(self):
    """
            It is used to provide the total probability transfers from the
            seen events to the unseen events.
    """
            return  1.0 * self.smoothedNr(1) / self._freqdist.N()

        def max(self):
            return self._freqdist.max()

        def samples(self):
            return self._freqdist.keys()

        def freqdist(self):
            return self._freqdist

        def __repr__(self):
    """
            It obtains the string representation of ProbDist.
    """
            return '<SimpleGoodTuringProbDist based on %d samples>'\
                    % self._freqdist.N()
```

Let's see the code for Simple Good Turing in NLTK:

```
>>> gt = lambda fd, bins: SimpleGoodTuringProbDist(fd, bins=1e5)
>>> train_and_test(gt)
5.17%
```

Kneser Ney estimation

Kneser Ney is used with trigrams. Let's see the following code in NLTK for the Kneser Ney estimation:

```
>>> import nltk
>>> corpus = [[((x[0],y[0],z[0]),(x[1],y[1],z[1]))
   for x, y, z in nltk.trigrams(sent)]
  for sent in corpus[:100]]
>>> tag_set = unique_list(tag for sent in corpus for (word,tag) in
sent)
>>> len(tag_set)
906
>>> symbols = unique_list(word for sent in corpus for (word,tag) in
sent)
>>> len(symbols)
1341
>>> trainer = nltk.tag.HiddenMarkovModelTrainer(tag_set, symbols)
>>> train_corpus = []
>>> test_corpus = []
>>> for i in range(len(corpus)):
if i % 10:
train_corpus += [corpus[i]]
else:
test_corpus += [corpus[i]]

>>> len(train_corpus)
90
>>> len(test_corpus)
10
>>> kn = lambda fd, bins: KneserNeyProbDist(fd)
>>> train_and_test(kn)
0.86%
```

Witten Bell estimation

Witten Bell is the smoothing algorithm that was designed to deal with unknown words having zero probability. Let's consider the following code for Witten Bell estimation in NLTK:

```
>>> train_and_test(WittenBellProbDist)
6.90%
```

Develop a back-off mechanism for MLE

Katz back-off may be defined as a generative n gram language model that computes the conditional probability of a given token given its previous information in n gram. According to this model, in training, if n gram is seen more than n times, then the conditional probability of a token, given its previous information, is proportional to the MLE of that n gram. Else, the conditional probability is equivalent to the back-off conditional probability of (n-1) gram.

The following is the code for Katz's back-off model in NLTK:

```
def prob(self, word, context):
"""
Evaluate the probability of this word in this context using Katz
Backoff.
: param word: the word to get the probability of
: type word: str
:param context: the context the word is in
:type context: list(str)
"""
context = tuple(context)
if(context+(word,) in self._ngrams) or (self._n == 1):
return self[context].prob(word)
else:
return self._alpha(context) * self._backoff.prob(word,context[1:])
```

Applying interpolation on data to get mix and match

The limitation of using an additive smoothed bigram is that we back off to a state of ignorance when we deal with rare text. For example, the word captivating occurs five times in a training data: thrice followed by *by* and twice followed by *the*. With additive smoothing, the occurrence of *a* and *new* before captivating is the same. Both the occurrences are plausible, but the former is more probable as compared to latter. This problem can be rectified using unigram probabilities. We can develop an interpolation model in which both the unigram and bigram probabilities can be combined.

In SRILM, we perform interpolation by first training a unigram model with `-order 1` and `–order 2` used for the bigram model:

```
ngram - count - text / home / linux / ieng6 / ln165w / public / data
/ engand hintrain . txt \ - vocab / home / linux / ieng6 / ln165w /
public / data / engandhinlexicon . txt \ - order 1 - addsmooth 0.0001
- lm wsj1 . lm
```

Evaluate a language model through perplexity

The `nltk.model.ngram` module in NLTK has a submodule, `perplexity(text)`. This submodule evaluates the perplexity of a given text. Perplexity is defined as 2**Cross Entropy for the text. Perplexity defines how a probability model or probability distribution can be useful to predict a text.

The code for evaluating the perplexity of text as present in the `nltk.model.ngram` module is as follows:

```
def perplexity(self, text):
"""
        Calculates the perplexity of the given text.
        This is simply 2 ** cross-entropy for the text.

        :param text: words to calculate perplexity of
        :type text: list(str)
"""

        return pow(2.0, self.entropy(text))
```

Applying metropolis hastings in modeling languages

There are various ways to perform processing on posterior distribution in **Markov Chain Monte Carlo** (**MCMC**). One way is using the Metropolis-Hastings sampler. In order to implement the Metropolis-Hastings algorithm, we require standard uniform distribution, proposal distribution, and target distribution that is proportional to posterior probability. An example of Metropolis-Hastings is discussed in the following topic.

Applying Gibbs sampling in language processing

With the help of Gibbs sampling, Markov chain is built by sampling from the conditional probability. When the iteration over all the parameters is completed, then one cycle of the Gibbs sampler is completed. When it is not possible to sample from conditional distribution, then Metropolis-Hastings can be used. This is referred to as Metropolis within Gibbs. Gibbs sampling may be defined as Metropolis-hastings with special proposal distribution. On each iteration, we draw a proposal for a new value of a specific parameter.

Consider an example of throwing two coins that is characterized by the number of heads and the number of tosses of a coin:

```
def bern(theta,z,N):
"""Bernoulli likelihood with N trials and z successes."""
return np.clip(theta**z*(1-theta)**(N-z),0,1)
def bern2(theta1,theta2,z1,z2,N1,N2):
"""Bernoulli likelihood with N trials and z successes."""
return bern(theta1,z1,N1)*bern(theta2,z2,N2)
def make_thetas(xmin,xmax,n):
xs=np.linspace(xmin,xmax,n)
widths=(xs[1:]-xs[:-1])/2.0
thetas=xs[:-1]+widths
returnt hetas
def make_plots(X,Y,prior,likelihood,posterior,projection=None):
fig,ax=plt.subplots(1,3,subplot_kw=dict(projection=projection,aspect='
equal'),figsize=(12,3))
ifprojection=='3d':
ax[0].plot_surface(X,Y,prior,alpha=0.3,cmap=plt.cm.jet)
ax[1].plot_surface(X,Y,likelihood,alpha=0.3,cmap=plt.cm.jet)
ax[2].plot_surface(X,Y,posterior,alpha=0.3,cmap=plt.cm.jet)
else:
ax[0].contour(X,Y,prior)
ax[1].contour(X,Y,likelihood)
ax[2].contour(X,Y,posterior)
ax[0].set_title('Prior')
ax[1].set_title('Likelihood')
ax[2].set_title('Posteior')
plt.tight_layout()
thetas1=make_thetas(0,1,101)
thetas2=make_thetas(0,1,101)
X,Y=np.meshgrid(thetas1,thetas2)
```

For Metropolis, the following values are considered:

```
a=2
b=3

z1=11
N1=14
z2=7
N2=14

prior=lambdatheta1,theta2:stats.beta(a,b).pdf(theta1)*stats.beta(a,b).
pdf(theta2)
```

```
lik=partial(bern2,z1=z1,z2=z2,N1=N1,N2=N2)
target=lambdatheta1,theta2:prior(theta1,theta2)*lik(theta1,theta2)

theta=np.array([0.5,0.5])
niters=10000
burnin=500
sigma=np.diag([0.2,0.2])

thetas=np.zeros((niters-burnin,2),np.float)
foriinrange(niters):
new_theta=stats.multivariate_normal(theta,sigma).rvs()
p=min(target(*new_theta)/target(*theta),1)
ifnp.random.rand()<p:
theta=new_theta
ifi>=burnin:
thetas[i-burnin]=theta
kde=stats.gaussian_kde(thetas.T)
XY=np.vstack([X.ravel(),Y.ravel()])
posterior_metroplis=kde(XY).reshape(X.shape)
make_plots(X,Y,prior(X,Y),lik(X,Y),posterior_metroplis)
make_plots(X,Y,prior(X,Y),lik(X,Y),posterior_
metroplis,projection='3d')
```

For Gibbs, the following values are considered:

```
a=2
b=3

z1=11
N1=14
z2=7
N2=14

prior=lambda theta1,theta2:stats.beta(a,b).pdf(theta1)*stats.
beta(a,b).pdf(theta2)
lik=partial(bern2,z1=z1,z2=z2,N1=N1,N2=N2)
target=lambdatheta1,theta2:prior(theta1,theta2)*lik(theta1,theta2)

theta=np.array([0.5,0.5])
niters=10000
burnin=500
sigma=np.diag([0.2,0.2])

thetas=np.zeros((niters-burnin,2),np.float)
foriinrange(niters):
```

```
theta=[stats.beta(a+z1,b+N1-z1).rvs(),theta[1]]
theta=[theta[0],stats.beta(a+z2,b+N2-z2).rvs()]

ifi>=burnin:
thetas[i-burnin]=theta
kde=stats.gaussian_kde(thetas.T)
XY=np.vstack([X.ravel(),Y.ravel()])
posterior_gibbs=kde(XY).reshape(X.shape)
make_plots(X,Y,prior(X,Y),lik(X,Y),posterior_gibbs)
make_plots(X,Y,prior(X,Y),lik(X,Y),posterior_gibbs,projection='3d')
```

In the preceding codes of Metropolis and Gibbs, 2D and 3D plots of prior, likelihood, and posterior would be obtained.

Summary

In this chapter, we have discussed about word frequencies (unigram, bigram, and trigram). You have studied Maximum Likelihood Estimation and its implementation in NLTK. We have discussed about the interpolation method, the backoff method, Gibbs sampling, and Metropolis-hastings. We have also discussed how we can perform language modeling through perplexity.

In the next chapter, we will discuss about Stemmer and Lemmatizer, and creating the Morphological generator using machine learning tools.

3
Morphology – Getting Our Feet Wet

Morphology may be defined as the study of the composition of words using morphemes. A morpheme is the smallest unit of language that has meaning. In this chapter, we will discuss stemming and lemmatizing, stemmer and lemmatizer for non-English languages, developing a morphological analyzer and morphological generator using machine learning tools, search engines, and many such concepts.

In brief, this chapter will include the following topics:

- Introducing morphology
- Understanding stemmer
- Understanding lemmatization
- Developing a stemmer for non-English languages
- Morphological analyzer
- Morphological generator
- Search engine

Introducing morphology

Morphology may be defined as the study of the production of tokens with the help of **morphemes**. A morpheme is the basic unit of language carrying meaning. There are two types of morpheme: stems and affixes (suffixes, prefixes, infixes, and circumfixes).

Stems are also referred to as free morphemes, since they can even exist without adding affixes. Affixes are referred to as bound morphemes, since they cannot exist in a free form and they always exist along with free morphemes. Consider the word `unbelievable`. Here, `believe` is a stem or a free morpheme. It can exist on its own. The morphemes `un` and `able` are affixes or bound morphemes. They cannot exist in a free form, but they exist together with stem. There are three kinds of language, namely **isolating languages**, **agglutinative languages**, and **inflecting languages**. Morphology has a different meaning in all these languages. Isolating languages are those languages in which words are merely free morphemes and they do not carry any tense (past, present, and future) and number (singular or plural) information. Mandarin Chinese is an example of an isolating language. Agglutinative languages are those in which small words combine together to convey compound information. Turkish is an example of an agglutinative language. Inflecting languages are those in which words are broken down into simpler units, but all these simpler units exhibit different meanings. Latin is an example of an inflecting language. Morphological processes are of the following types: inflection, derivation, semiaffixes and combining forms, and cliticization. Inflection means transforming the word into a form so that it represents person, number, tense, gender, case, aspect, and mood. Here, the syntactic category of a token remains the same. In derivation, the syntactic category of a word is also changed. Semiaffixes are bound morphemes that exhibit words, such as quality, for example, noteworthy, antisocial, anticlockwise, and so on.

Understanding stemmer

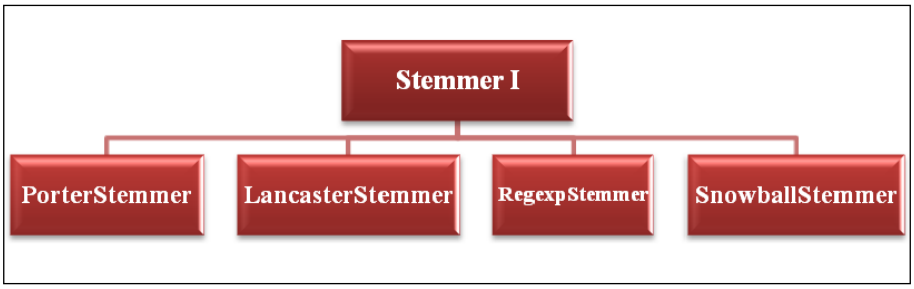
Stemming may be defined as the process of obtaining a stem from a word by eliminating the affixes from a word. For example, in the case of the word `raining`, stemmer would return the root word or stem word `rain` by removing the affix from raining. In order to increase the accuracy of information retrieval, search engines mostly use stemming to get the stems and store them as indexed words. Search engines call words with the same meaning synonyms, which may be a kind of query expansion known as conflation. *Martin Porter* has designed a well-known stemming algorithm known as the *Porter stemming algorithm*. This algorithm is basically designed to replace and eliminate some well-known suffices present in English words. To perform stemming in NLTK, we can simply do an instantiation of the `PorterStemmer` class and then perform stemming by calling the `stem` method.

Let's see the code for stemming using the `PorterStemmer` class in NLTK:

```
>>> import nltk
>>> from nltk.stem import PorterStemmer
>>> stemmerporter = PorterStemmer()
>>> stemmerporter.stem('working')
```

```
'work'
>>> stemmerporter.stem('happiness')
'happi'
```

The `PorterStemmer` class has been trained and has knowledge of the many stems and word forms of English. The process of stemming takes place in a series of steps and transforms the word into a shorter word or a word that has a similar meaning to the root word. The **Stemmer I** interface defines the `stem()` method, and all the stemmers are inherited from the **Stemmer I** interface. The inheritance diagram is depicted here:



Another stemming algorithm known as the *Lancaster stemming algorithm* was introduced at Lancaster University. Similar to the `PorterStemmer` class, the `LancasterStemmer` class is used in NLTK to implement Lancaster stemming. However, one of the major differences between the two algorithms is that Lancaster stemming involves the use of more words of different sentiments as compared to Porter Stemming.

Let's consider the following code that depicts Lancaster stemming in NLTK:

```
>>> import nltk
>>> from nltk.stem import LancasterStemmer
>>> stemmerlan=LancasterStemmer()
>>> stemmerlan.stem('working')
'work'
>>> stemmerlan.stem('happiness')
'happy'
```

We can also build our own stemmer in NLTK using `RegexpStemmer`. It works by accepting a string and eliminating the string from the prefix or suffix of a word when a match is found.

Let's consider an example of stemming using `RegexpStemmer` in NLTK:

```
>>> import nltk
>>> from nltk.stem import RegexpStemmer
>>> stemmerregexp=RegexpStemmer('ing')
>>> stemmerregexp.stem('working')
'work'
>>> stemmerregexp.stem('happiness')
'happiness'
>>> stemmerregexp.stem('pairing')
'pair'
```

We can use `RegexpStemmer` in the cases in which stemming cannot be performed using `PorterStemmer` and `LancasterStemmer`.

`SnowballStemmer` is used to perform stemming in 13 languages other than English. In order to perform stemming using `SnowballStemmer`, firstly, an instance is created in the language in which stemming needs to be performed. Then, using the `stem()` method, stemming is performed.

Consider the following example of performing stemming in Spanish and French in NLTK using `SnowballStemmer`:

```
>>> import nltk
>>> from nltk.stem import SnowballStemmer
>>> SnowballStemmer.languages
('danish', 'dutch', 'english', 'finnish', 'french', 'german',
'hungarian', 'italian', 'norwegian', 'porter', 'portuguese',
'romanian', 'russian', 'spanish', 'swedish')
>>> spanishstemmer=SnowballStemmer('spanish')
>>> spanishstemmer.stem('comiendo')
'com'
>>> frenchstemmer=SnowballStemmer('french')
>>> frenchstemmer.stem('manger')
'mang'
```

`Nltk.stem.api` consists of the `Stemmer I` class in which the `stem` function is performed.

Consider the following code present in NLTK that enables us to perform stemming:

```
Class StemmerI(object):
"""
It is an interface that helps to eliminate morphological affixes from
the tokens and the process is known as stemming.
"""
```

```
def stem(self, token):
"""
Eliminate affixes from token and stem is returned.
"""
raise NotImplementedError()
```

Let's see the code used to perform stemming using multiple stemmers:

```
>>> import nltk
>>> from nltk.stem.porter import PorterStemmer
>>> from nltk.stem.lancaster import LancasterStemmer
>>> from nltk.stem import SnowballStemmer
>>> def obtain_tokens():
With open('/home/p/NLTK/sample1.txt') as stem: tok = nltk.word_
tokenize(stem.read())
return tokens
>>> def  stemming(filtered):
stem=[]
for x in filtered:
stem.append(PorterStemmer().stem(x))
return stem
>>>  if_name_=="_main_":
tok= obtain_tokens()
>>>print("tokens is %s")%(tok)
>>>stem_tokens= stemming(tok)
>>>print("After stemming is %s")%stem_tokens
>>>res=dict(zip(tok,stem_tokens))
>>>print("{tok:stemmed}=%s")%(result)
```

Understanding lemmatization

Lemmatization is the process in which we transform the word into a form with a different word category. The word formed after lemmatization is entirely different. The built-in `morphy()` function is used for lemmatization in `WordNetLemmatizer`. The inputted word is left unchanged if it is not found in WordNet. In the argument, `pos` refers to the part of speech category of the inputted word.

Consider an example of lemmatization in NLTK:

```
>>> import nltk
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer_output=WordNetLemmatizer()
>>> lemmatizer_output.lemmatize('working')
'working'
```

```
>>> lemmatizer_output.lemmatize('working',pos='v')
'work'
>>> lemmatizer_output.lemmatize('works')
'work'
```

The `WordNetLemmatizer` library may be defined as a wrapper around the so-called WordNet corpus, and it makes use of the `morphy()` function present in `WordNetCorpusReader` to extract a lemma. If no lemma is extracted, then the word is only returned in its original form. For example, for `works`, the lemma returned is the singular form, `work`.

Let's consider the following code that illustrates the difference between stemming and lemmatization :

```
>>> import nltk
>>> from nltk.stem import PorterStemmer
>>> stemmer_output=PorterStemmer()
>>> stemmer_output.stem('happiness')
'happi'
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer_output=WordNetLemmatizer()
>>> lemmatizer_output.lemmatize('happiness')
'happiness'
```

In the preceding code, `happiness` is converted to `happi` by stemming. Lemmatization doesn't find the root word for `happiness`, so it returns the word `happiness`.

Developing a stemmer for non-English language

Polyglot is a software that is used to provide models called morfessor models that are used to obtain morphemes from tokens. The Morpho project's goal is to create unsupervised data-driven processes. The main aim of the Morpho project is to focus on the creation of morphemes, which is the smallest unit of syntax. Morphemes play an important role in natural language processing. Morphemes are useful in automatic recognition and the creation of language. With the help of the vocabulary dictionaries of Polyglot, morfessor models on the 50,000 tokens of different languages were used.

Let's see the code for obtaining the language table using `polyglot`:

```
from polyglot.downloader import downloader
print(downloader.supported_languages_table("morph2"))
```

The output obtained from preceding code is the languages listed here:

1. Piedmontese language	2. Lombard language	3. Gan Chinese
4. Sicilian	5. Scots	6. Kirghiz, Kyrgyz
7. Pashto, Pushto	8. Kurdish	9. Portuguese
10. Kannada	11. Korean	12. Khmer
13. Kazakh	14. Ilokano	15. Polish
16. Panjabi, Punjabi	17. Georgian	18. Chuvash
19. Alemannic	20. Czech	21. Welsh
22. Chechen	23. Catalan; Valencian	24. Northern Sami
25. Sanskrit (Sa?sk?ta)	26. Slovene	27. Javanese
28. Slovak	29. Bosnian-Croatian-Serbian	30. Bavarian
31. Swedish	32. Swahili	33. Sundanese
34. Serbian	35. Albanian	36. Japanese
37. Western Frisian	38. French	39. Finnish
40. Upper Sorbian	41. Faroese	42. Persian
43. Sinhala, Sinhalese	44. Italian	45. Amharic
46. Aragonese	47. Volapük	48. Icelandic
49. Sakha	50. Afrikaans	51. Indonesian
52. Interlingua	53. Azerbaijani	54. Ido
55. Arabic	56. Assamese	57. Yoruba
58. Yiddish	59. Waray-Waray	60. Croatian
61. Hungarian	62. Haitian; Haitian Creole	63. Quechua
64. Armenian	65. Hebrew (modern)	66. Silesian
67. Hindi	68. Divehi; Dhivehi; Mald...	69. German
70. Danish	71. Occitan	72. Tagalog
73. Turkmen	74. Thai	75. Tajik
76. Greek, Modern	77. Telugu	78. Tamil
79. Oriya	80. Ossetian, Ossetic	81. Tatar
82. Turkish	83. Kapampangan	84. Venetian
85. Manx	86. Gujarati	87. Galician
88. Irish	89. Scottish Gaelic; Gaelic	90. Nepali
91. Cebuano	92. Zazaki	93. Walloon
94. Dutch	95. Norwegian	96. Norwegian Nynorsk
97. West Flemish	98. Chinese	99. Bosnian
100. Breton	101. Belarusian	102. Bulgarian
103. Bashkir	104. Egyptian Arabic	105. Tibetan Standard, Tib...
106. Bengali	107. Burmese	108. Romansh
109. Marathi (Mara?hi)	110. Malay	111. Maltese
112. Russian	113. Macedonian	114. Malayalam
115. Mongolian	116. Malagasy	117. Vietnamese
118. Spanish; Castilian	119. Estonian	120. Basque
121. Bishnupriya Manipuri	122. Asturian	123. English
124. Esperanto	125. Luxembourgish, Letzeb...	126. Latin
127. Uighur, Uyghur	128. Ukrainian	129. Limburgish, Limburgan...
130. Latvian	131. Urdu	132. Lithuanian
133. Fiji Hindi	134. Uzbek	135. Romanian, Moldavian, ...

The necessary models can be downloaded using the following code:

```
%%bash
polyglot download morph2.en morph2.ar

[polyglot_data] Downloading package morph2.en to
[polyglot_data]     /home/rmyeid/polyglot_data...
[polyglot_data]   Package morph2.en is already up-to-date!
[polyglot_data] Downloading package morph2.ar to
[polyglot_data]     /home/rmyeid/polyglot_data...
[polyglot_data]   Package morph2.ar is already up-to-date!
```

Consider an example that is used to obtain an output from `polyglot`:

```
from polyglot.text import Text, Word
tokens =["unconditional" ,"precooked", "impossible", "painful",
"entered"]
for s in tokens:
s=Word(s, language="en")
print("{:<20}{}".format(s,s.morphemes))

unconditional['un','conditional']
precooked['pre','cook','ed']
impossible['im','possible']
painful['pain','ful']
entered['enter','ed']
```

If tokenization is not performed properly, then we can perform morphological analysis for the process of splitting the text into the original constituents:

```
sent="Ihopeyoufindthebookinteresting"
para=Text(sent)
para.language="en"
para.morphemes
WordList(['I','hope','you','find','the','book','interesting'])
```

Morphological analyzer

Morphological analysis may be defined as the process of obtaining grammatical information from tokens, given their suffix information. Morphological analysis can be performed in three ways: morpheme-based morphology (or anitem and arrangement approach), lexeme-based morphology (or an item and process approach), and word-based morphology (or a word and paradigm approach). A morphological analyzer may be defined as a program that is responsible for the analysis of the morphology of a given input token. It analyzes a given token and generates morphological information, such as gender, number, class, and so on, as an output.

In order to perform morphological analysis on a given non-whitespace token, the `pyEnchant` dictionary is used.

Let's consider the following code that performs morphological analysis:

```
>>> import enchant
>>> s = enchant.Dict("en_US")
>>> tok=[]
>>> def tokenize(st1):
if not st1:return
for j in xrange(len(st1),-1,-1):
if s.check(st1[0:j]):
tok.append(st1[0:i])
st1=st[j:]
tokenize(st1)
break
>>> tokenize("itismyfavouritebook")
>>> tok
['it', 'is', 'my','favourite','book']
>>> tok=[ ]
>>> tokenize("ihopeyoufindthebookinteresting")
>>> tok
['i','hope','you','find','the','book','interesting']
```

We can determine the category of the word with the help of the following points:

- **Morphological hints**: The suffix's information helps us detect the category of a word. For example, the -ness and –ment suffixes exist with nouns.
- **Syntactic hints**: Contextual information is conducive to determine the category of a word. For example, if we have found the word that has the noun category, then syntactic hints will be useful for determining whether an adjective would appear before the noun or after the noun category.
- **Semantic hints**: A semantic hint is also useful for determining the word's category. For example, if we already know that a word represents the name of a location, then it will fall under the noun category.
- **Open class**: This is class of words that are not fixed, and their number keeps on increasing every day, whenever a new word is added to their list. Words in the open class are usually nouns. Prepositions are mostly in a closed class. For example, there can be an unlimited number of words in the of Persons list. So, it is an open class.
- **Morphology captured by the Part of Speech tagset**: The Part of Speech tagset captures information that helps us perform morphology. For example, the word `plays` would appear with the third person and a singular noun.

- **Omorfi**:**Omorfi** (**Open morphology of Finnish**) is a package that has been licensed by GNU GPL version 3. It is used for performing numerous tasks, such as language modeling, morphological analysis, rule-based machine translation, information retrieval, statistical machine translation, morphological segmentation, ontologies, and spell checking and correction.

Morphological generator

A morphological generator is a program that performs the task of morphological generation. Morphological generation may be considered an opposite task of morphological analysis. Here, given the description of a word in terms of number, category, stem, and so on, the original word is retrieved. For example, if *root* = *go*, *part of speech* = *verb*, *tense*= *present*, and if it occurs along with a third person and singular subject, then a morphological generator would generate its surface form, goes.

There is a lot of Python-based software that performs morphological analysis and generation. Some of them are as follows:

- **ParaMorfo**: It is used to perform morphological generation and analysis of Spanish and Guarani nouns, adjectives, and verbs.
- **HornMorpho**: It is used for the morphological generation and analysis of Oromo and Amharic nouns and verbs, as well as Tigrinya verbs.
- **AntiMorfo**: It is used for the morphological generation and analysis of Quechua adjectives, verbs, and nouns, as well as Spanish verbs.
- **MorfoMelayu**: It is used for the morphological analysis of Malay words.

Other examples of software that is used to perform morphological analysis and generation are as follows:

- Morph is a morphological generator and analyzer for English for the RASP system
- Morphy is a morphological generator, analyzer, and POS tagger for German
- Morphisto is a morphological generator and analyzer for German
- Morfette performs supervised learning (inflectional morphology) for Spanish and French

Search engine

PyStemmer 1.0.1 consists of Snowball stemming algorithms that are used for performing information retrieval tasks and for constructing a search engine. It consists of the Porter stemming algorithm and many other stemming algorithms that are useful for performing stemming and information retrieval tasks in many languages, including many European languages.

We can construct a vector space search engine by converting the texts into vectors.

The following are the steps involved in constructing a vector space search engine:

1. Consider the following code for the removal of stopwords and tokenization:

   A stemmer is a program that accepts words and converts them into stems. Tokens that have the same stem have nearly the same meanings. Stopwords are also eliminated from a text.

```
def eliminatestopwords(self,list):
"""
Eliminate words which occur often and have not much significance
from context point of view.
"""
return[ word for word in list if word not in self.stopwords ]

def tokenize(self,string):
"""
Perform the task of splitting text into stop words and tokens
"""
Str=self.clean(str)
Words=str.split("")
return [self.stemmer.stem(word,0,len(word)-1) for word in words]
```

2. Consider the following code for mapping keywords into vector dimensions:

```
def obtainvectorkeywordindex(self, documentList):
"""
In the document vectors,  generate the keyword for the given
position of element
"""


#Perform mapping of text into strings
vocabstring = "".join(documentList)

vocablist = self.parser.tokenise(vocabstring)
```

```
#Eliminate common words that have no search significance
vocablist = self.parser.eliminatestopwords(vocablist)
uniqueVocablist = util.removeDuplicates(vocablist)

vectorIndex={}
 offset=0
#Attach a position to keywords that performs mapping with
dimension that is used to depict this token
 for word in uniqueVocablist:
vectorIndex[word]=offset
offset+=1
 return vectorIndex  #(keyword:position)
```

3. Here, a simple term count model is used. Consider the following code for the conversion of text strings into vectors:

```
def constructVector(self, wordString):

        # Initialise the vector with 0's
        Vector_val = [0] * len(self.vectorKeywordIndex)
        tokList = self.parser.tokenize(tokString)
        tokList = self.parser.eliminatestopwords(tokList)
        for word in toklist:
                vector[self.vectorKeywordIndex[word]] += 1;
# simple Term Count Model is used
        return vector
```

4. Searching similar documents by finding the cosine of an angle between the vectors of a document, we can prove whether two given documents are similar or not. If the cosine value is 1, then the angle's value is 0 degrees and the vectors are said to be parallel (this means that the documents are said to be related). If the cosine value is 0 and value of the angle is 90 degrees, then the vectors are said to be perpendicular (this means that the documents are not said to be related). Let's see the code for computing the cosine between the text vectors using SciPy:

```
def cosine(vec1, vec2):
"""
                cosine  = ( X * Y ) / ||X|| x ||Y||
"""
return float(dot(vec1,vec2) / (norm(vec1) * norm(vec2)))
```

5. We perform the mapping of keywords to vector space. We construct a temporary text that represents the items to be searched and then compare it with document vectors with the help of cosine measurement. Let's see the following code for searching the vector space:

```
def searching(self,searchinglist):
""" search for text that are  matched on the  basis oflist of
items """
        askVector = self.buildQueryVector(searchinglist)

ratings = [util.cosine(askVector, textVector) for textVector in
self.documentVectors]
        ratings.sort(reverse=True)
        return ratings
```

6. We will now consider the following code that can be used for detecting languages from the source text:

```
>>>  import nltk
>>>  import sys
>>> try:
from nltk import wordpunct_tokenize
from nltk.corpus import stopwords
except ImportError:
print( 'Error has occured')



#-------------------------------------------------------------------
-----
>>> def _calculate_languages_ratios(text):
"""
Compute probability of given document that can be written in
different languages and give a dictionary that appears like
{'german': 2, 'french': 4, 'english': 1}
"""
 languages_ratios = {}
'''
nltk.wordpunct_tokenize() splits all punctuations into separate
tokens
 wordpunct_tokenize("I hope you like the book interesting .")
[' I',' hope ','you ','like ','the ','book' ,'interesting ','.']
'''

tok = wordpunct_tokenize(text)
wor = [word.lower() for word in tok]

  # Compute occurence of unique stopwords in a text
for language in stopwords.fileids():
stopwords_set = set(stopwords.words(language))
```

```
words_set = set(words)
common_elements = words_set.intersection(stopwords_set)
languages_ratios[language] = len(common_elements)
# language "score"
return languages_ratios

#--------------------------------------------------------------

>>> def detect_language(text):
"""
Compute the probability of given text that is written in different
languages and obtain the one that is highest scored. It makes
use of stopwords calculation approach, finds out unique stopwords
present in a analyzed text.
"""
ratios = _calculate_languages_ratios(text)
most_rated_language = max(ratios, key=ratios.get)
return most_rated_language



if __name__=='__main__':

 text = '''
All over this cosmos, most of the people believe that there is
an invisible supreme power that is the creator and the runner of
this world. Human being is supposed to be the most intelligent and
loved creation by that power and that is being searched by human
beings in different ways into different things. As a result people
reveal His assumed form as per their own perceptions and beliefs.
It has given birth to different religions and people are divided
on the name of religion viz. Hindu, Muslim, Sikhs, Christian etc.
People do not stop at this. They debate the superiority of one
over the other and fight to establish their views. Shrewd people
like politicians oppose and support them at their own convenience
to divide them and control them. It has intensified to the extent
that even parents of a
new born baby teach it about religious differences and recommend
their own religion superior to that of others and let the child
learn to hate other people just because of religion. Jonathan
Swift, an eighteenth century novelist, observes that we have just
enough religion to make us hate, but not enough to make us love
one another.
The word 'religion' does not have a derogatory meaning - A literal
meaning of religion is 'A
personal or institutionalized system grounded in belief in a God
or Gods and the activities connected
```

```
with this'. At its basic level, 'religion is just a set of
teachings that tells people how to lead a good
life'. It has never been the purpose of religion to divide people
into groups of isolated followers that
cannot live in harmony together. No religion claims to teach
intolerance or even instructs its believers to segregate a
certain religious group or even take the fundamental rights of
an individual solely based on their religious choices. It is also
said that 'Majhab nhi sikhata aaps mai bair krna'.But this very
majhab or religion takes a very heinous form when it is misused
by the shrewd politicians and the fanatics e.g. in Ayodhya on 6th
December, 1992 some right wing political parties
and communal organizations incited the Hindus to demolish the 16th
century Babri Masjid in the
name of religion to polarize Hindus votes. Muslim fanatics in
Bangladesh retaliated and destroyed a
number of temples, assassinated innocent Hindus and raped Hindu
girls who had nothing to do with
the demolition of Babri Masjid. This very inhuman act has been
presented by Taslima Nasrin, a Bangladeshi Doctor-cum-Writer
in her controversial novel 'Lajja' (1993) in which, she seems
to utilizes fiction's mass emotional appeal, rather than its
potential for nuance and universality.
'''

>>> language = detect_language(text)

>>> print(language)
```

The preceding code will search for stopwords and detect the language of the text, that is, English.

Summary

The field of computational linguistics has numerous applications. We need to perform preprocessing on our original text in order to implement or build an application. In this chapter, we have discussed stemming, lemmatization, and morphological analysis and generation, and their implementation in NLTK. We have also discussed search engines and their implementation.

In the next chapter, we will discuss parts of speech, tagging, and chunking.

4
Parts-of-Speech Tagging – Identifying Words

Parts-of-speech (**POS**) tagging is one of the many tasks in NLP. It is defined as the process of assigning a particular parts-of-speech tag to individual words in a sentence. The parts-of-speech tag identifies whether a word is a noun, verb, adjective, and so on. There are numerous applications of parts-of-speech tagging, such as information retrieval, machine translation, NER, language analysis, and so on.

This chapter will include the following topics:

- Creating POS tagged corpora
- Selecting a machine learning algorithm
- Statistical modeling involving the n-gram approach
- Developing a chunker using POS tagged data

Introducing parts-of-speech tagging

Parts-of-speech tagging is the process of assigning a category (for example, noun, verb, adjective, and so on) tag to individual tokens in a sentence. In NLTK, taggers are present in the `nltk.tag` package and it is inherited by the `TaggerIbase` class.

Consider an example to implement POS tagging for a given sentence in NLTK:

```
>>> import nltk
>>> text1=nltk.word_tokenize("It is a pleasant day today")
>>> nltk.pos_tag(text1)
[('It', 'PRP'), ('is', 'VBZ'), ('a', 'DT'), ('pleasant', 'JJ'),
('day', 'NN'), ('today', 'NN')]
```

We can implement the `tag()` method in all the subclasses of `TaggerI`. In order to evaluate tagger, `TaggerI` has provided the `evaluate()` method. A combination of taggers can be used to form a back-off chain so that the next tagger can be used for tagging if one tagger is not tagging.

Let's see the list of available tags provided by **Penn Treebank** (`https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`):

```
CC - Coordinating conjunction
CD - Cardinal number
DT - Determiner
EX - Existential there
FW - Foreign word
IN - Preposition or subordinating conjunction
JJ - Adjective
JJR - Adjective, comparative
JJS - Adjective, superlative
LS - List item marker
MD - Modal
NN - Noun, singular or mass
NNS - Noun, plural
NNP - Proper noun, singular
NNPS - Proper noun, plural
PDT - Predeterminer
POS - Possessive ending
PRP - Personal pronoun
PRP$ - Possessive pronoun (prolog version PRP-S)
RB - Adverb
RBR - Adverb, comparative
RBS - Adverb, superlative
RP - Particle
SYM - Symbol
TO - to
UH - Interjection
VB - Verb, base form
VBD - Verb, past tense
VBG - Verb, gerund or present participle
VBN - Verb, past participle
VBP - Verb, non-3rd person singular present
VBZ - Verb, 3rd person singular present
WDT - Wh-determiner
WP - Wh-pronoun
WP$ - Possessive wh-pronoun (prolog version WP-S)
WRB - Wh-adverb
```

NLTK may provide the information of tags. Consider the following code, which provides information about the `NNS` tag:

```
>>> nltk.help.upenn_tagset('NNS')
NNS: noun, common, plural
    undergraduates scotches bric-a-brac products bodyguards facets
coasts
    divestitures storehouses designs clubs fragrances averages
    subjectivists apprehensions muses factory-jobs ...
```

Let's see another example in which a regular expression may also be queried:

```
>>> nltk.help.upenn_tagset('VB.*')
VB: verb, base form
    ask assemble assess assign assume atone attention avoid bake
balkanize
    bank begin behold believe bend benefit bevel beware bless boil
bomb
    boost brace break bring broil brush build ...
VBD: verb, past tense
    dipped pleaded swiped regummed soaked tidied convened halted
registered
    cushioned exacted snubbed strode aimed adopted belied figgered
    speculated wore appreciated contemplated ...
VBG: verb, present participle or gerund
    telegraphing stirring focusing angering judging stalling lactating
    hankerin' alleging veering capping approaching traveling besieging
    encrypting interrupting erasing wincing ...
VBN: verb, past participle
    multihulled dilapidated aerosolized chaired languished panelized
used
 experimented flourished imitated reunifed factored condensed sheared
    unsettled primed dubbed desired ...
VBP: verb, present tense, not 3rd person singular
    predominate wrap resort sue twist spill cure lengthen brush
terminate
    appear tend stray glisten obtain comprise detest tease attract
    emphasize mold postpone sever return wag ...
VBZ: verb, present tense, 3rd person singular
    bases reconstructs marks mixes displeases seals carps weaves
snatches
    slumps stretches authorizes smolders pictures emerges stockpiles
    seduces fizzes uses bolsters slaps speaks pleads ...R
```

The preceding code gives information regarding all the tags of verb phrases.

Let's look at an example that depicts words' sense disambiguation achieved through POS tagging:

```
>>> import nltk
>>> text=nltk.word_tokenize("I cannot bear the pain of bear")
>>> nltk.pos_tag(text)
[('I', 'PRP'), ('can', 'MD'), ('not', 'RB'), ('bear', 'VB'), ('the',
'DT'), ('pain', 'NN'), ('of', 'IN'), ('bear', 'NN')]
```

Here, in the previous sentence, `bear` is a verb, which means to tolerate, and it also is an animal, which means that it is a noun.

In NLTK, a tagged token is represented as a tuple consisting of a token and its tag. We can create this tuple in NLTK using the `str2tuple()` function:

```
>>> import nltk
>>> taggedword=nltk.tag.str2tuple('bear/NN')
>>> taggedword
('bear', 'NN')
>>> taggedword[0]
'bear'
>>> taggedword[1]
'NN'
```

Let's consider an example in which sequences of tuples can be generated from the given text:

```
>>> import nltk
>>> sentence='''The/DT sacred/VBN Ganga/NNP flows/VBZ in/IN this/DT
region/NN ./. This/DT is/VBZ a/DT pilgrimage/NN ./. People/NNP from/IN
all/DT over/IN the/DT country/NN visit/NN this/DT place/NN ./. '''
>>> [nltk.tag.str2tuple(t) for t in sentence.split()]
[('The', 'DT'), ('sacred', 'VBN'), ('Ganga', 'NNP'), ('flows', 'VBZ'),
('in', 'IN'), ('this', 'DT'), ('region', 'NN'), ('.', '.'), ('This',
'DT'), ('is', 'VBZ'), ('a', 'DT'), ('pilgrimage', 'NN'), ('.', '.'),
('People', 'NNP'), ('from', 'IN'), ('all', 'DT'), ('over', 'IN'),
('the', 'DT'), ('country', 'NN'), ('visit', 'NN'), ('this', 'DT'),
('place', 'NN'), ('.', '.')]
```

Now, consider the following code that converts the tuple (`word` and `pos` tag) into a word and a tag:

```
>>> import nltk
>>> taggedtok = ('bear', 'NN')
>>> from nltk.tag.util import tuple2str
>>> tuple2str(taggedtok)
'bear/NN'
```

Let's see the occurrence of some common tags in the Treebank corpus:

```
>>> import nltk
>>> from nltk.corpus import treebank
>>> treebank_tagged = treebank.tagged_words(tagset='universal')
>>> tag = nltk.FreqDist(tag for (word, tag) in treebank_tagged)
>>> tag.most_common()
[('NOUN', 28867), ('VERB', 13564), ('.', 11715), ('ADP', 9857),
('DET', 8725), ('X', 6613), ('ADJ', 6397), ('NUM', 3546), ('PRT',
3219), ('ADV', 3171), ('PRON', 2737), ('CONJ', 2265)]
```

Consider the following code, which calculates the number of tags occurring before a noun tag:

```
>>> import nltk
>>> from nltk.corpus import treebank
>>> treebank_tagged = treebank.tagged_words(tagset='universal')
>>> tagpairs = nltk.bigrams(treebank_tagged)
>>> preceders_noun = [x[1] for (x, y) in tagpairs if y[1] == 'NOUN']
>>> freqdist = nltk.FreqDist(preceders_noun)
>>> [tag for (tag, _) in freqdist.most_common()]
['NOUN', 'DET', 'ADJ', 'ADP', '.', 'VERB', 'NUM', 'PRT', 'CONJ',
'PRON', 'X', 'ADV']
```

We can also provide POS tags to tokens using dictionaries in Python. Let's see the following code that illustrates the creation of a tuple (`word:pos` tag) using dictionaries in Python:

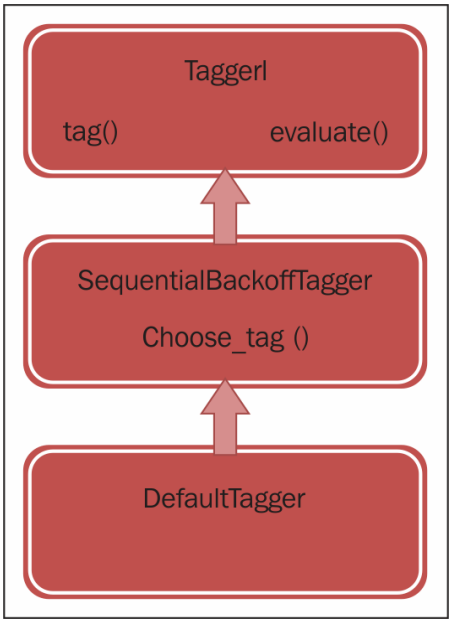
```
>>> import nltk
>>> tag={}
>>> tag
{}
>>> tag['beautiful']='ADJ'
>>> tag
{'beautiful': 'ADJ'}
>>> tag['boy']='N'
>>> tag['read']='V'
>>> tag['generously']='ADV'
>>> tag
{'boy': 'N', 'beautiful': 'ADJ', 'generously': 'ADV', 'read': 'V'}
```

Default tagging

Default tagging is a kind of tagging that assigns identical parts-of-speech tags to all the tokens. The subclass of `SequentialBackoffTagger` is `DefaultTagger`. The `choose_tag()` method must be implemented by `SequentialBackoffTagger`. This method includes the following arguments:

- A collection of tokens
- The index of the token that should be tagged
- The previous tags list

The hierarchy of tagger is depicted as follows:



Let's now see the following code, which depicts the working of `DefaultTagger`:

```
>>> import nltk
>>> from nltk.tag import DefaultTagger
>>> tag = DefaultTagger('NN')
>>> tag.tag(['Beautiful', 'morning'])
[('Beautiful', 'NN'), ('morning', 'NN')]
```

We can convert a tagged sentence into an untagged sentence with the help of `nltk.tag.untag()`. After calling this function, the tags on individual tokens will be eliminated.

Let's see the code for untagging a sentence:

```
>>> from nltk.tag import untag
>>> untag([('beautiful', 'NN'), ('morning', 'NN')])
['beautiful', 'morning']
```

Creating POS-tagged corpora

A **corpus** may be known as a collection of documents. A **corpora** is the collection of multiple corpus.

Let's see the following code, which will generate a `data` directory inside the `home` directory:

```
>>> import nltk
>>> import os,os.path
>>> create = os.path.expanduser('~/nltkdoc')
>>> if not os.path.exists(create):
  os.mkdir(create)


>>> os.path.exists(create)
True
>>> import nltk.data
>>> create in nltk.data.path
True
```

This code will create a data directory named `~/nltkdoc` inside the `home` directory. The last line of this code will return `True` and will ensure that the `data` directory has been created. If the last line of the code returns `False`, then it means that the `data` directory has not been created and we need to create it manually. After creating the `data` directory manually, we can test the last line and it will then return `True`. Within this directory, we can create another directory named `nltkcorpora` that will hold the whole corpus. The path will be `~/nltkdoc/nltkcorpora`. Also, we can create a subdirectory named `important` that will hold all the necessary files.

The path will be `~/nltkdoc/nltkcorpora/important`.

Let's see the following code to load a text file into the subdirectory:

```
>>> import nltk.data
>>> nltk.data.load('nltkcorpora/important/firstdoc.txt',format='raw')
'nltk\n'
```

Here, in the previous code, we have mentioned `format='raw'`, since `nltk.data.load()` cannot interpret `.txt` files.

There is a word list corpus in NLTK known as the `Names` corpus. It consists of two files, namely, `male.txt` and `female.txt`.

Let's see the code to generate the length of `male.txt` and `female.txt`:

```
>>> import nltk
>>> from nltk.corpus import names
>>> names.fileids()
['female.txt', 'male.txt']
>>> len(names.words('male.txt'))
2943
>>> len(names.words('female.txt'))
5001
```

NLTK also consists of a large collection of English words. Let's see the code that describes the number of words present in the English word file:

```
>>> import nltk
>>> from nltk.corpus import words
>>> words.fileids()
['en', 'en-basic']
>>> len(words.words('en'))
235886
>>> len(words.words('en-basic'))
850
```

Consider the following code used in NLTK for defining the **Maxent Treebank** POS tagger:

```
def pos_tag(tok):
    """
```

We can use POS tagger given by NLTK to tag a list of tokens:

```
>>> from nltk.tag import pos_tag
>>> from nltk.tokenize import word_tokenize
>>> pos_tag(word_tokenize("Papa's favourite hobby is reading."))
        [('Papa', 'NNP'), ("'s", 'POS'), ('favourite', 'JJ'),
('hobby', 'NN'), ('is',
        'VBZ'), ('reading', 'VB'),  ('.', '.')]

    :param tokens: list of tokens that need to be tagged
    :type tok: list(str)
    :return: The tagged tokens
```

```
        :rtype: list(tuple(str, str))
        """
        tagger = load(_POS_TAGGER)
        return tagger.tag(tok)

    def batch_pos_tag(sent):
        """
        We can use part of speech tagger given by NLTK to perform tagging
    of list of tokens.
        """
        tagger = load(_POS_TAGGER)
        return tagger.batch_tag(sent)
```

Selecting a machine learning algorithm

POS tagging is also referred to as word category disambiguation or grammatical tagging. POS tagging may be of two types: rule-based or stochastic/probabilistic. E. Brill's tagger is based on the rule-based tagging algorithm.

A POS classifier takes a document as input and obtains word features. It trains itself with the help of these word features combined with the already available training labels. This type of classifier is referred to as a second order classifier, and it makes use of the bootstrap classifier in order to generate the tags for words.

A `backoff` classifier is one in which backoff procedure is performed. The output is obtained in such a manner that the trigram POS tagger relies on the bigram POS tagger, which in turn relies on the unigram POS tagger.

While training a POS classifier, a feature set is generated. This feature set may comprise the following:

- Information about the current word
- Information about the previous word or prefix
- Information about the next word or successor

In NLTK, `FastBrillTagger` is based on unigram. It makes use of a dictionary of words that are already known and the pos tag information.

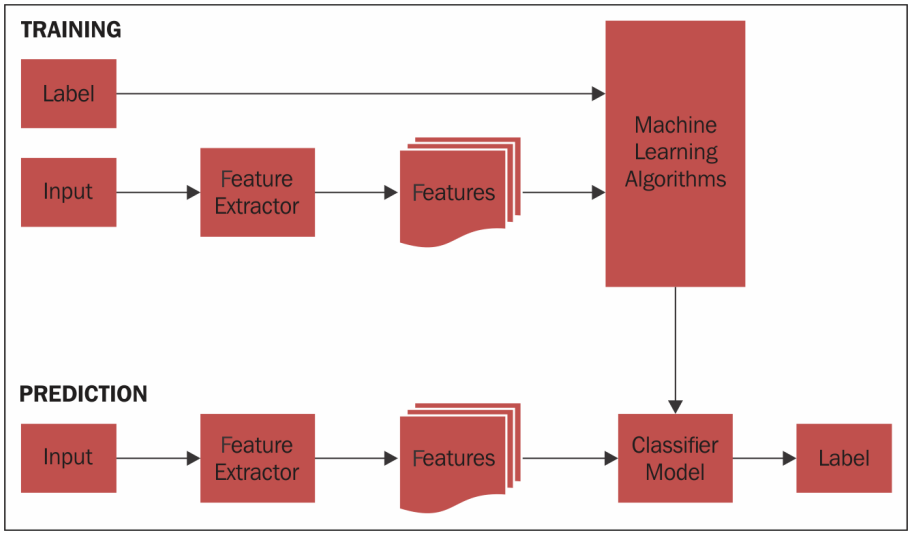
Let's see the code for `FastBrillTagger` used in NLTK:

```
from nltk.tag import UnigramTagger
from nltk.tag import FastBrillTaggerTrainer

from nltk.tag.brill import SymmetricProximateTokensTemplate
```

```
from nltk.tag.brill import ProximateTokensTemplate
from nltk.tag.brill import ProximateTagsRule
from nltk.tag.brill import ProximateWordsRule

ctx = [ # Context = surrounding words and tags.
    SymmetricProximateTokensTemplate(ProximateTagsRule, (1, 1)),
    SymmetricProximateTokensTemplate(ProximateTagsRule, (1, 2)),
    SymmetricProximateTokensTemplate(ProximateTagsRule, (1, 3)),
    SymmetricProximateTokensTemplate(ProximateTagsRule, (2, 2)),
    SymmetricProximateTokensTemplate(ProximateWordsRule, (0, 0)),
    SymmetricProximateTokensTemplate(ProximateWordsRule, (1, 1)),
    SymmetricProximateTokensTemplate(ProximateWordsRule, (1, 2)),
    ProximateTokensTemplate(ProximateTagsRule, (-1, -1), (1, 1)),
]

tagger = UnigramTagger(sentences)
tagger = FastBrillTaggerTrainer(tagger, ctx, trace=0)
tagger = tagger.train(sentences, max_rules=100)
```

Classification may be defined as the process of deciding a POS tag for a given input.

In **supervised classification**, a training corpus is used that comprises a word and its correct tag. In **unsupervised classification**, any pair of words and a correct tag list does not exist:



In supervised classification, during training, a feature extractor accepts the input and labels and generates a set of features. These features set along with the label act as input to machine learning algorithms. During the testing or prediction phase, a feature extractor is used that generates features from unknown inputs, and the output is sent to a classifier model that generates an output in the form of label or pos tag information with the help of machine learning algorithms.

The **maximum entropy** classifier is one in that searches the parameter set in order to maximize the total likelihood of the corpus used for training.

It may be defined as follows:

$$P(features\_word) = \Sigma_{x \text{ in } corpus} P(label\_word(x) | features\_word(x))$$
$$P(label\_word | features\_word) = P(label\_word, features\_word) / \Sigma_{label\_word} P(label\_word, features\_word)$$

Statistical modeling involving the n-gram approach

Unigram means a single word. In a unigram tagger, a single token is used to find the particular parts-of-speech tag.

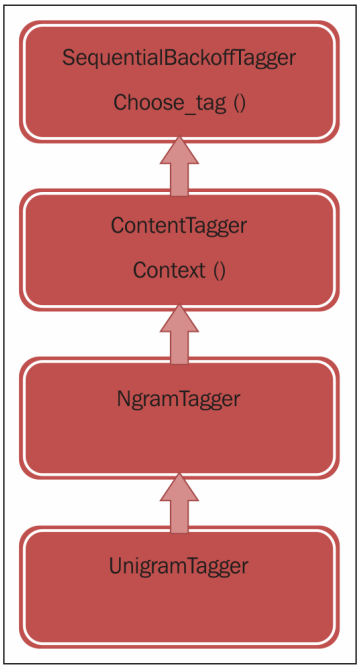
Training of `UnigramTagger` can be performed by providing it with a list of sentences at the time of initialization.

Let's see the following code in NLTK, which performs `UnigramTagger` training:

```
>>> import nltk
>>> from nltk.tag import UnigramTagger
>>> from nltk.corpus import treebank
>>> training= treebank.tagged_sents()[:7000]
>>> unitagger=UnigramTagger(training)
>>> treebank.sents()[0]
['Pierre', 'Vinken', ',', '61', 'years', 'old', ',', 'will', 'join',
'the', 'board', 'as', 'a', 'nonexecutive', 'director', 'Nov.', '29',
'.']
>>> unitagger.tag(treebank.sents()[0])
[('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ('61', 'CD'),
('years', 'NNS'), ('old', 'JJ'), (',', ','), ('will', 'MD'), ('join',
'VB'), ('the', 'DT'), ('board', 'NN'), ('as', 'IN'), ('a', 'DT'),
('nonexecutive', 'JJ'), ('director', 'NN'), ('Nov.', 'NNP'), ('29',
'CD'), ('.', '.')]
```

In the preceding code, we have performed training using the first 7000 sentences of the Treebank corpus.

The hierarchy followed by `UnigramTagger` is depicted in the following inheritance diagram:



To evaluate `UnigramTagger`, let's see the following code, which calculates the accuracy:

```
>>> import nltk
>>> from nltk.corpus import treebank
>>> from nltk.tag import UnigramTagger
>>> training= treebank.tagged_sents()[:7000]
>>> unitagger=UnigramTagger(training)
>>> testing = treebank.tagged_sents()[2000:]
>>> unitagger.evaluate(testing)
0.963400866227395
```

So, it is 96% accurate in correctly performing pos tagging.

Since `UnigramTagger` inherits from `ContextTagger`, we can map the context key with a specific tag.

Consider the following example of tagging using `UnigramTagger`:

```
>>> import nltk
>>> from nltk.corpus import treebank
>>> from nltk.tag import UnigramTagger
>>> unitag = UnigramTagger(model={'Vinken': 'NN'})
>>> unitag.tag(treebank.sents()[0])
[('Pierre', None), ('Vinken', 'NN'), (',', None), ('61', None),
('years', None), ('old', None), (',', None), ('will', None), ('join',
None), ('the', None), ('board', None), ('as', None), ('a', None),
('nonexecutive', None), ('director', None), ('Nov.', None), ('29',
None), ('.', None)]
```

Here, in the preceding code, `UnigramTagger` only tags `'Vinken'` with the `'NN'` tag and the rest are tagged with the `'None'` tag since we have provided the tag for the word `'Vinken'` in the context model and no other words are included in the context model.

In a given context, `ContextTagger` uses the frequency of a given tag to decide the occurrence of the most probable tag. In order to use minimum threshold frequency, we can pass a specific value to the cutoff value. Let's see the code that evaluates `UnigramTagger`:

```
>>> unitagger = UnigramTagger(training, cutoff=5)
>>> unitagger.evaluate(testing)
0.7974218445306567
```

Backoff tagging may be defined as a feature of `SequentialBackoffTagger`. All the taggers are chained together so that if one of the taggers is unable to tag a token, then the token may be passed to the next tagger.

Let's see the following code, which uses back-off tagging. Here, `DefaultTagger` and `UnigramTagger` are used to tag a token. If any tagger of them is unable to tag a word, then the next tagger may be used to tag it:

```
>>> import nltk
>>> from nltk.tag import UnigramTagger
>>> from nltk.tag import DefaultTagger
>>> from nltk.corpus import treebank
>>> testing = treebank.tagged_sents()[2000:]
>>> training= treebank.tagged_sents()[:7000]
>>> tag1=DefaultTagger('NN')
>>> tag2=UnigramTagger(training,backoff=tag1)
>>> tag2.evaluate(testing)
0.963400866227395
```

The subclasses of `NgramTagger` are`UnigramTagger`, `BigramTagger`, and `TrigramTagger`. `BigramTagger` makes use of the previous tag as contextual information. `TrigramTagger` uses the previous two tags as contextual information.

Consider the following code, which illustrates the implementation of `BigramTagger`:

```
>>> import nltk
>>> from nltk.tag import BigramTagger
>>> from nltk.corpus import treebank
>>> training_1= treebank.tagged_sents()[:7000]
>>> bigramtagger=BigramTagger(training_1)
>>> treebank.sents()[0]
['Pierre', 'Vinken', ',', '61', 'years', 'old', ',', 'will', 'join',
'the', 'board', 'as', 'a', 'nonexecutive', 'director', 'Nov.', '29',
'.']
>>> bigramtagger.tag(treebank.sents()[0])
[('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ('61', 'CD'),
('years', 'NNS'), ('old', 'JJ'), (',', ','), ('will', 'MD'), ('join',
'VB'), ('the', 'DT'), ('board', 'NN'), ('as', 'IN'), ('a', 'DT'),
('nonexecutive', 'JJ'), ('director', 'NN'), ('Nov.', 'NNP'), ('29',
'CD'), ('.', '.')]
>>> testing_1 = treebank.tagged_sents()[2000:]
>>> bigramtagger.evaluate(testing_1)
0.922942709936983
```

Let's see another code for `BigramTagger` and `TrigramTagger`:

```
>>> import nltk
>>> from nltk.tag import BigramTagger, TrigramTagger
>>> from nltk.corpus import treebank
>>> testing = treebank.tagged_sents()[2000:]
>>> training= treebank.tagged_sents()[:7000]
>>> bigramtag = BigramTagger(training)
>>> bigramtag.evaluate(testing)
0.9190426339881356
>>> trigramtag = TrigramTagger(training)
>>> trigramtag.evaluate(testing)
0.9101956195989079
```

`NgramTagger` can be used to generate a tagger for *n* greater than three as well. Let's see the following code in NLTK, which develops `QuadgramTagger`:

```
>>> import nltk
>>> from nltk.corpus import treebank
>>> from nltk import NgramTagger
>>> testing = treebank.tagged_sents()[2000:]
```

```
>>> training= treebank.tagged_sents()[:7000]
>>> quadgramtag = NgramTagger(4, training)
>>> quadgramtag.evaluate(testing)
0.9429767842847466
```

The `AffixTagger` is also a `ContextTagger` in that makes use of a prefix or suffix as the contextual information.

Let's see the following code, which uses `AffixTagger`:

```
>>> import nltk
>>> from nltk.corpus import treebank
>>> from nltk.tag import AffixTagger
>>> testing = treebank.tagged_sents()[2000:]
>>> training= treebank.tagged_sents()[:7000]
>>> affixtag = AffixTagger(training)
>>> affixtag.evaluate(testing)
0.29043249789601167
```

Let's see the following code, which learns the use of four character prefixes:

```
>>> import nltk
>>> from nltk.tag import AffixTagger
>>> from nltk.corpus import treebank
>>> testing = treebank.tagged_sents()[2000:]
>>> training= treebank.tagged_sents()[:7000]
>>> prefixtag = AffixTagger(training, affix_length=4)
>>> prefixtag.evaluate(testing)
0.21103516226368618
```

Consider the following code, which learns the use of three character suffixes:

```
>>> import nltk
>>> from nltk.tag import AffixTagger
>>> from nltk.corpus import treebank
>>> testing = treebank.tagged_sents()[2000:]
>>> training= treebank.tagged_sents()[:7000]
>>> suffixtag = AffixTagger(training, affix_length=-3)
>>> suffixtag.evaluate(testing)
0.29043249789601167
```

Consider the following code in NLTK, which that combines many affix taggers in the back-off chain:

```
>>> import nltk
>>> from nltk.tag import AffixTagger
>>> from nltk.corpus import treebank
```

```
>>> testing = treebank.tagged_sents()[2000:]
>>> training= treebank.tagged_sents()[:7000]
>>> prefixtagger=AffixTagger(training,affix_length=4)
>>> prefixtagger.evaluate(testing)
0.21103516226368618
>>> prefixtagger3=AffixTagger(training,affix_
length=3,backoff=prefixtagger)
>>> prefixtagger3.evaluate(testing)
0.25906767658107027
>>> suffixtagger3=AffixTagger(training,affix_length=-
3,backoff=prefixtagger3)
>>> suffixtagger3.evaluate(testing)
0.2939630929654946
>>> suffixtagger4=AffixTagger(training,affix_length=-
4,backoff=suffixtagger3)
>>> suffixtagger4.evaluate(testing)
0.3316090892296324
```

The **TnT** is **Trigrams n Tags**. `TnT` is a statistical-based tagger that is based on the second order Markov models.

Let's see the code in NLTK for `TnT`:

```
>>> import nltk
>>> from nltk.tag import tnt
>>> from nltk.corpus import treebank
>>> testing = treebank.tagged_sents()[2000:]
>>> training= treebank.tagged_sents()[:7000]
>>> tnt_tagger=tnt.TnT()
>>> tnt_tagger.train(training)
>>> tnt_tagger.evaluate(testing)
0.9882176652913768
```

`TnT` computes `ConditionalFreqDist` and `internalFreqDist` from the training text. These instances are used to compute unigrams, bigrams, and trigrams. In order to choose the best tag, TnT uses the ngram model.

Consider the following code of a `DefaultTagger` in which, if the value of the unknown tagger is provided explicitly, then `TRAINED` will be set to `TRUE`:

```
>>> import nltk
>>> from nltk.tag import DefaultTagger
>>> from nltk.tag import tnt
>>> from nltk.corpus import treebank
>>> testing = treebank.tagged_sents()[2000:]
>>> training= treebank.tagged_sents()[:7000]
```

```
>>> tnt_tagger=tnt.TnT()
>>> unknown=DefaultTagger('NN')
>>> tagger_tnt=tnt.TnT(unk=unknown,Trained=True)
>>> tnt_tagger.train(training)
>>> tnt_tagger.evaluate(testing)
0.988238192006897
```

Developing a chunker using pos-tagged corpora

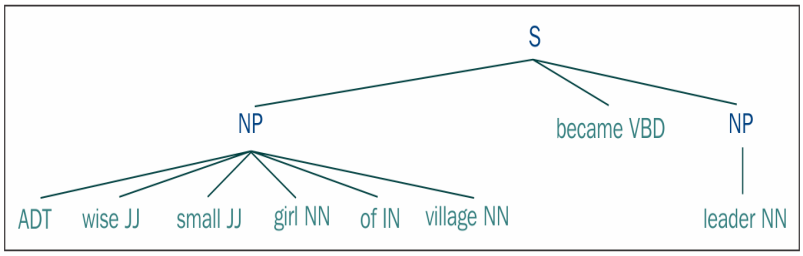
Chunking is the process used to perform entity detection. It is used for the segmentation and labeling of multiple sequences of tokens in a sentence.

To design a chunker, a chunk grammar should be defined. A chunk grammar holds the rules of how chunking should be done.

Let's consider the example that performs *Noun Phrase Chunking* by forming the chunk rules:

```
>>> import nltk
>>> sent=[("A","DT"),("wise", "JJ"), ("small", "JJ"),("girl", "NN"),
("of", "IN"), ("village", "N"),  ("became", "VBD"), ("leader", "NN")]
>>> sent=[("A","DT"),("wise", "JJ"), ("small", "JJ"),("girl", "NN"),
("of", "IN"), ("village", "NN"),  ("became", "VBD"), ("leader", "NN")]
>>> grammar = "NP: {<DT>?<JJ>*<NN><IN>?<NN>*}"
>>> find = nltk.RegexpParser(grammar)
>>> res = find.parse(sent)
>>> print(res)
(S
  (NP A/DT wise/JJ small/JJ girl/NN of/IN village/NN)
  became/VBD
  (NP leader/NN))
>>> res.draw()
```

The following parse tree is generated:

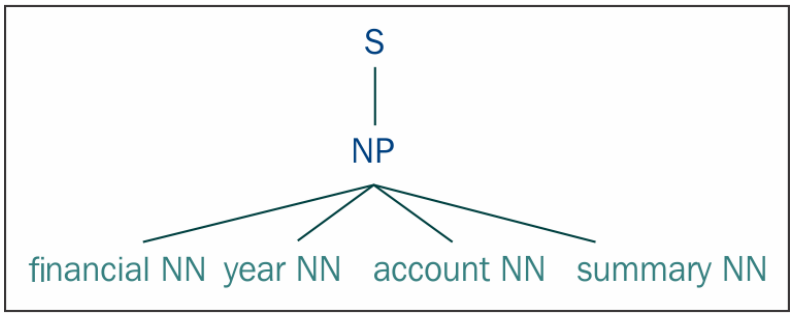


Here, the chunk rule for **Noun Phrase** is defined by keeping **DT** as optional, any number of **JJ**, followed by **NN**, optional **IN**, and any number of **NN**.

Consider another example in which the **Noun Phrase** chunk rule is created with any number of nouns:

```
>>> import nltk
>>> noun1=[("financial","NN"),("year","NN"),("account","NN"),("summar
y","NN")]
>>> gram="NP:{<NN>+}"
>>> find = nltk.RegexpParser(gram)
>>> print(find.parse(noun1))
(S (NP financial/NN year/NN account/NN summary/NN))
>>> x=find.parse(noun1)
>>> x.draw()
```

The output in the form of the parse tree is given here:



Chunking is the process in which some of the parts of a chunk are eliminated. Either an entire chunk may be used, a part of the chunk may be used from the middle and the remaining parts are eliminated, or a part of chunk may be used either from the beginning of the chunk or from the end of the chunk and the remaining part of the chunk is removed.

Consider the code for `UnigramChunker` in NLTK, which has been developed to perform chunking and parsing:

```
class UnigramChunker(nltk.ChunkParserI):
  def _init_(self,training):
    training_data=[[(x,y) for p,x,y in nltk.chunk.treeconlltags(sent)]
        for sent in training]
    self.tagger=nltk.UnigramTagger(training_data)
```

```
  def parsing(self,sent):
    postags=[pos1 for (word1,pos1) in sent]
    tagged_postags=self.tagger.tag(postags)
    chunk_tags=[chunking for (pos1,chunktag) in tagged_postags]
    conll_tags=[(word,pos1,chunktag) for ((word,pos1),chunktag)
        in zip(sent, chunk_tags)]
    return nltk.chunk.conlltaags2tree(conlltags)
```

Consider the following code, which can be used to estimate the accuracy of the chunker after it is trained:

```
import nltk.corpus, nltk.tag

def ubt_conll_chunk_accuracy(train_sents, test_sents):
    chunks_train =conll_tag_chunks(training)
    chunks_test =conll_tag_chunks(testing)

    chunker1 =nltk.tag.UnigramTagger(chunks_train)
    print 'u:', nltk.tag.accuracy(chunker1, chunks_test)

    chunker2 =nltk.tag.BigramTagger(chunks_train, backoff=chunker1)
    print 'ub:', nltk.tag.accuracy(chunker2, chunks_test)

    chunker3 =nltk.tag.TrigramTagger(chunks_train, backoff=chunker2)
    print 'ubt:', nltk.tag.accuracy(chunker3, chunks_test)

    chunker4 =nltk.tag.TrigramTagger(chunks_train, backoff=chunker1)
    print 'ut:', nltk.tag.accuracy(chunker4, chunks_test)

    chunker5 =nltk.tag.BigramTagger(chunks_train, backoff=chunker4)
    print 'utb:', nltk.tag.accuracy(chunker5, chunks_test)

# accuracy test for conll chunking
conll_train =nltk.corpus.conll2000.chunked_sents('train.txt')
conll_test =nltk.corpus.conll2000.chunked_sents('test.txt')
ubt_conll_chunk_accuracy(conll_train, conll_test)

# accuracy test for treebank chunking
treebank_sents =nltk.corpus.treebank_chunk.chunked_sents()
ubt_conll_chunk_accuracy(treebank_sents[:2000], treebank_sents[2000:])
```

Summary

In this chapter, we have discussed POS tagging, different POS taggers, and the approaches used for POS tagging. You have also learned about statistical modeling involving the n-gram approach, and have developed a chunker using POS tags information.

In the following chapter, we will discuss Treebank construction, CFG construction, different parsing algorithms, and so on.

5
Parsing – Analyzing Training Data

Parsing, also referred to as syntactic analysis, is one of the tasks in NLP. It is defined as the process of finding whether a character sequence, written in natural language, is in accordance with the rules defined in formal grammar. It is the process of breaking the sentences into words or phrase sequences and providing them a particular component category (noun, verb, preposition, and so on).

This chapter will include the following topics:

- Treebank construction
- Extracting **Context-free Grammar** (**CFG**) rules from Treebank
- Creating a probabilistic Context-free Grammar from CFG
- CYK chart parsing algorithm
- Earley chart parsing algorithm

Introducing parsing

Parsing is one of the steps involved in NLP. It is defined as the process of determining the part-of-speech category for an individual component in a sentence and analyzing whether a given sentence is in accordance with grammar rules or not. The term parsing has been derived from the Latin word *pars* (oration is) which means part-of-speech.

Consider an example—*Ram bought a book*. This sentence is grammatically correct. But, instead of this sentence, if we have a sentence Book bought a Ram, then by adding the semantic information to the parse tree so constructed, we can conclude that although the sentence is grammatically correct, it is not semantically correct. So, the generation of a parse tree is followed by adding meaning to it as well. A parser is a software that accepts an input text and constructs a parse tree or a syntax tree. Parsing may be divided into two categories Top-down Parsing and Bottom-up Parsing. In Top-down Parsing, we begin from the start symbol and continue till we reach individual components. Some of the Top-down Parsers include the Recursive Descent Parser, LL Parser, and Earley Parser. In Bottom-up Parsing, we start from individual components and continue till we reach the start symbol. Some Bottom-up Parsers include the Operator-precedence parser, Simple precedence parser, Simple LR Parser, LALR Parser, Canonical LR (*LR(1)*) Parser, GLR Parser, CYK or (alternatively CKY) Parser, Recursive ascent parser, and Shift-reduce parser.

The `nltk.parse.api.ParserI` class is defined in NLTK. This class is used to obtain parses or syntactic structures for a given sentence. Parsers can be used to obtain syntactic structures, discourse structures, and morphological trees.

Chart parsing follows the dynamic programming approach. In this, once some results are obtained, these may be treated as the intermediate results and may be reused to obtain future results. Unlike in Top-down parsing, the same task is not performed again and again.

Treebank construction

The `nltk.corpus.package` consists of a number of `corpus reader`classes that can be used to obtain the contents of various corpora.

Treebank corpus can also be accessed from `nltk.corpus`. Identifiers for files can be obtained using `fileids()`:

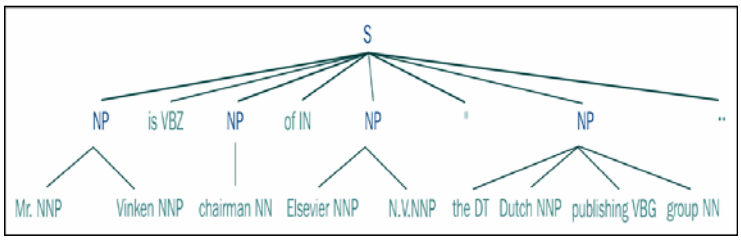
```
>>> import nltk
>>> import nltk.corpus
>>> print(str(nltk.corpus.treebank).replace('\\\\','/'))
<BracketParseCorpusReader in 'C:/nltk_data/corpora/treebank/combined'>
>>> nltk.corpus.treebank.fileids()
['wsj_0001.mrg', 'wsj_0002.mrg', 'wsj_0003.mrg', 'wsj_0004.
mrg', 'wsj_0005.mrg', 'wsj_0006.mrg', 'wsj_0007.mrg', 'wsj_0008.
mrg', 'wsj_0009.mrg', 'wsj_0010.mrg', 'wsj_0011.mrg', 'wsj_0012.
mrg', 'wsj_0013.mrg', 'wsj_0014.mrg', 'wsj_0015.mrg', 'wsj_0016.
mrg', 'wsj_0017.mrg', 'wsj_0018.mrg', 'wsj_0019.mrg', 'wsj_0020.
mrg', 'wsj_0021.mrg', 'wsj_0022.mrg', 'wsj_0023.mrg', 'wsj_0024.
mrg', 'wsj_0025.mrg', 'wsj_0026.mrg', 'wsj_0027.mrg', 'wsj_0028.mrg',
'wsj_0029.mrg', 'wsj_0030.mrg', 'wsj_0031.mrg', 'wsj_0032.
```

```
mrg', 'wsj_0033.mrg', 'wsj_0034.mrg', 'wsj_0035.mrg', 'wsj_0036.
mrg', 'wsj_0037.mrg', 'wsj_0038.mrg', 'wsj_0039.mrg', 'wsj_0040.
mrg', 'wsj_0041.mrg', 'wsj_0042.mrg', 'wsj_0043.mrg', 'wsj_0044.
mrg', 'wsj_0045.mrg', 'wsj_0046.mrg', 'wsj_0047.mrg', 'wsj_0048.
mrg', 'wsj_0049.mrg', 'wsj_0050.mrg', 'wsj_0051.mrg', 'wsj_0052.
mrg', 'wsj_0053.mrg', 'wsj_0054.mrg', 'wsj_0055.mrg', 'wsj_0056.
mrg', 'wsj_0057.mrg', 'wsj_0058.mrg', 'wsj_0059.mrg', 'wsj_0060.
mrg', 'wsj_0061.mrg', 'wsj_0062.mrg', 'wsj_0063.mrg', 'wsj_0064.
mrg', 'wsj_0065.mrg', 'wsj_0066.mrg', 'wsj_0067.mrg', 'wsj_0068.
mrg', 'wsj_0069.mrg', 'wsj_0070.mrg', 'wsj_0071.mrg', 'wsj_0072.
mrg', 'wsj_0073.mrg', 'wsj_0074.mrg', 'wsj_0075.mrg', 'wsj_0076.
mrg', 'wsj_0077.mrg', 'wsj_0078.mrg', 'wsj_0079.mrg', 'wsj_0080.
mrg', 'wsj_0081.mrg', 'wsj_0082.mrg', 'wsj_0083.mrg', 'wsj_0084.
mrg', 'wsj_0085.mrg', 'wsj_0086.mrg', 'wsj_0087.mrg', 'wsj_0088.
mrg', 'wsj_0089.mrg', 'wsj_0090.mrg', 'wsj_0091.mrg', 'wsj_0092.
mrg', 'wsj_0093.mrg', 'wsj_0094.mrg', 'wsj_0095.mrg', 'wsj_0096.
mrg', 'wsj_0097.mrg', 'wsj_0098.mrg', 'wsj_0099.mrg', 'wsj_0100.
mrg', 'wsj_0101.mrg', 'wsj_0102.mrg', 'wsj_0103.mrg', 'wsj_0104.
mrg', 'wsj_0105.mrg', 'wsj_0106.mrg', 'wsj_0107.mrg', 'wsj_0108.
mrg', 'wsj_0109.mrg', 'wsj_0110.mrg', 'wsj_0111.mrg', 'wsj_0112.
mrg', 'wsj_0113.mrg', 'wsj_0114.mrg', 'wsj_0115.mrg', 'wsj_0116.
mrg', 'wsj_0117.mrg', 'wsj_0118.mrg', 'wsj_0119.mrg', 'wsj_0120.
mrg', 'wsj_0121.mrg', 'wsj_0122.mrg', 'wsj_0123.mrg', 'wsj_0124.
mrg', 'wsj_0125.mrg', 'wsj_0126.mrg', 'wsj_0127.mrg', 'wsj_0128.
mrg', 'wsj_0129.mrg', 'wsj_0130.mrg', 'wsj_0131.mrg', 'wsj_0132.
mrg', 'wsj_0133.mrg', 'wsj_0134.mrg', 'wsj_0135.mrg', 'wsj_0136.
mrg', 'wsj_0137.mrg', 'wsj_0138.mrg', 'wsj_0139.mrg', 'wsj_0140.
mrg', 'wsj_0141.mrg', 'wsj_0142.mrg', 'wsj_0143.mrg', 'wsj_0144.
mrg', 'wsj_0145.mrg', 'wsj_0146.mrg', 'wsj_0147.mrg', 'wsj_0148.
mrg', 'wsj_0149.mrg', 'wsj_0150.mrg', 'wsj_0151.mrg', 'wsj_0152.
mrg', 'wsj_0153.mrg', 'wsj_0154.mrg', 'wsj_0155.mrg', 'wsj_0156.
mrg', 'wsj_0157.mrg', 'wsj_0158.mrg', 'wsj_0159.mrg', 'wsj_0160.
mrg', 'wsj_0161.mrg', 'wsj_0162.mrg', 'wsj_0163.mrg', 'wsj_0164.
mrg', 'wsj_0165.mrg', 'wsj_0166.mrg', 'wsj_0167.mrg', 'wsj_0168.
mrg', 'wsj_0169.mrg', 'wsj_0170.mrg', 'wsj_0171.mrg', 'wsj_0172.
mrg', 'wsj_0173.mrg', 'wsj_0174.mrg', 'wsj_0175.mrg', 'wsj_0176.
mrg', 'wsj_0177.mrg', 'wsj_0178.mrg', 'wsj_0179.mrg', 'wsj_0180.
mrg', 'wsj_0181.mrg', 'wsj_0182.mrg', 'wsj_0183.mrg', 'wsj_0184.
mrg', 'wsj_0185.mrg', 'wsj_0186.mrg', 'wsj_0187.mrg', 'wsj_0188.
mrg', 'wsj_0189.mrg', 'wsj_0190.mrg', 'wsj_0191.mrg', 'wsj_0192.
mrg', 'wsj_0193.mrg', 'wsj_0194.mrg', 'wsj_0195.mrg', 'wsj_0196.mrg',
'wsj_0197.mrg', 'wsj_0198.mrg', 'wsj_0199.mrg']
>>> from nltk.corpus import treebank
>>> print(treebank.words('wsj_0007.mrg'))
['McDermott', 'International', 'Inc.', 'said', '0', ...]
>>> print(treebank.tagged_words('wsj_0007.mrg'))
[('McDermott', 'NNP'), ('International', 'NNP'), ...]
```

Let's see the code in NLTK for accessing the Penn Treebank Corpus, which uses the Treebank Corpus Reader contained in the corpus module:

```
>>> import nltk
>>> from nltk.corpus import treebank
>>> print(treebank.parsed_sents('wsj_0007.mrg')[2])
(S
  (NP-SBJ
    (NP (NNP Bailey) (NNP Controls))
    (, ,)
    (VP
      (VBN based)
      (NP (-NONE- *))
      (PP-LOC-CLR
        (IN in)
        (NP (NP (NNP Wickliffe)) (, ,) (NP (NNP Ohio)))))
    (, ,))
  (VP
    (VBZ makes)
    (NP
      (JJ computerized)
      (JJ industrial)
      (NNS controls)
      (NNS systems)))
  (. .))

>>> import nltk
>>> from nltk.corpus import treebank_chunk
>>> treebank_chunk.chunked_sents()[1]
Tree('S', [Tree('NP', [('Mr.', 'NNP'), ('Vinken', 'NNP')]), ('is',
'VBZ'), Tree('NP', [('chairman', 'NN')]), ('of', 'IN'), Tree('NP',
[('Elsevier', 'NNP'), ('N.V.', 'NNP')]), (',', ','), Tree('NP',
[('the', 'DT'), ('Dutch', 'NNP'), ('publishing', 'VBG'), ('group',
'NN')]), ('.', '.')])
>>> treebank_chunk.chunked_sents()[1].draw()
```

The preceding code obtains the following parse tree:



```
>>> import nltk
>>> from nltk.corpus import treebank_chunk
>>> treebank_chunk.chunked_sents()[1].leaves()
[('Mr.', 'NNP'), ('Vinken', 'NNP'), ('is', 'VBZ'), ('chairman',
'NN'), ('of', 'IN'), ('Elsevier', 'NNP'), ('N.V.', 'NNP'), (',', ','),
('the', 'DT'), ('Dutch', 'NNP'), ('publishing', 'VBG'), ('group',
'NN'), ('.', '.')]
>>> treebank_chunk.chunked_sents()[1].pos()
[(('Mr.', 'NNP'), 'NP'), (('Vinken', 'NNP'), 'NP'), (('is', 'VBZ'),
'S'), (('chairman', 'NN'), 'NP'), (('of', 'IN'), 'S'), (('Elsevier',
'NNP'), 'NP'), (('N.V.', 'NNP'), 'NP'), ((',', ','), 'S'), (('the',
'DT'), 'NP'), (('Dutch', 'NNP'), 'NP'), (('publishing', 'VBG'), 'NP'),
(('group', 'NN'), 'NP'), (('.', '.'), 'S')]
>>> treebank_chunk.chunked_sents()[1].productions()
[S -> NP ('is', 'VBZ') NP ('of', 'IN') NP (',', ',') NP ('.', '.'),
NP -> ('Mr.', 'NNP') ('Vinken', 'NNP'), NP -> ('chairman', 'NN'), NP
-> ('Elsevier', 'NNP') ('N.V.', 'NNP'), NP -> ('the', 'DT') ('Dutch',
'NNP') ('publishing', 'VBG') ('group', 'NN')]
```

Part of speech annotations are included in the `tagged_words()` method:

```
>>> nltk.corpus.treebank.tagged_words()
[('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ...]
```

The type of tags and the count of these tags used in the Penn Treebank Corpus are shown here:

#	16
$	724
"	
,	4,886
-LRB-	120
-NONE-	6,592
-RRB-	126
.	384
:	563
``	712
CC	2,265
CD	3,546
DT	8,165
EX	88
FW	4

IN	9,857
JJ	5,834
JJR	381
JJS	182
LS	13
MD	927
NN	13,166
NNP	9,410
NNPS	244
NNS	6,047
PDT	27
POS	824
PRP	1,716
PRP$	766
RB	2,822
RBR	136
RBS	35
RP	216
SYM	1
TO	2,179
UH	3
VB	2,554
VBD	3,043
VBG	1,460
VBN	2,134
VBP	1,321
VBZ	2,125
WDT	445
WP	241
WP$	14

The tags and frequency can be obtained from the following code:

```
>>> import nltk
>>> from nltk.probability import FreqDist
>>> from nltk.corpus import treebank
>>> fd = FreqDist()
```

```
>>> fd.items()
dict_items([])
```

The preceding code obtains a list of tags and the frequency of each tag in the Treebank corpus.

Let's see the code in NLTK for accessing the Sinica Treebank Corpus:

```
>>> import nltk
>>> from nltk.corpus import sinica_treebank
>>> print(sinica_treebank.sents())
[['一'], ['友情'], ['嘉珍', '和', '我', '住在', '同一條', '巷子'], ...]
>>> sinica_treebank.parsed_sents()[27]
Tree('S', [Tree('NP', [Tree('NP', [Tree('N·的', [Tree('Nhaa', ['我']),
Tree('DE', ['的'])]), Tree('Ncb', ['腦海'])]), Tree('Ncda', ['中'])]),
Tree('Dd', ['頓時']), Tree('DM', ['一片']), Tree('VH11', ['空白'])])
```

Extracting Context Free Grammar (CFG) rules from Treebank

CFG was defined for natural languages in 1957 by Noam Chomsky. A CFG consists of the following components:

- A set of non terminal nodes (*N*)
- A set of terminal nodes (*T*)
- Start symbol (*S*)
- A set of production rules (*P*) of the form:

  $A \rightarrow a$

CFG rules are of two types—Phrase structure rules and Sentence structure rules.

A Phrase Structure Rule can be defined as follows—$A \rightarrow a$, where $A \hat{I} N$ and *a* consists of Terminals and Non terminals.

In Sentence level Construction of CFG, there are four structures:

- **Declarative structure**: Deals with declarative sentences (the subject is followed by a predicate).
- **Imperative structure**: Deals with imperative sentences, commands, or suggestions (sentences begin with a verb phrase and do not include a subject).

- **Yes-No structure**: Deals with question-answering sentences. The answers to these questions are either *yes* or *no*.
- **Wh-question structure**: Deals with question-answering sentences. Questions that begin following *Wh* words (Who, What, How, When, Where, Why, and Which).

General CFG rules are summarized here:

- *S→NP VP*
- *S→VP*
- *S→Aux NP VP*
- *S→Wh-NP VP*
- *S→Wh-NP Aux NP VP*
- *NP→(Det) (AP) Nom (PP)*
- *VP→Verb (NP) (NP) (PP)**
- *VP→Verb S*
- *PP→Prep (NP)*
- *AP→(Adv) Adj (PP)*

Consider an example that depicts the use of Context-free Grammar rules in NLTK:

```
>>> import nltk
>>> from nltk import Nonterminal, nonterminals, Production, CFG
>>> nonterminal1 = Nonterminal('NP')
>>> nonterminal2 = Nonterminal('VP')
>>> nonterminal3 = Nonterminal('PP')
>>> nonterminal1.symbol()
'NP'
>>> nonterminal2.symbol()
'VP'
>>> nonterminal3.symbol()
'PP'
>>> nonterminal1==nonterminal2
False
>>> nonterminal2==nonterminal3
False
>>> nonterminal1==nonterminal3
False
>>> S, NP, VP, PP = nonterminals('S, NP, VP, PP')
>>> N, V, P, DT = nonterminals('N, V, P, DT')
>>> production1 = Production(S, [NP, VP])
```

```
>>> production2 = Production(NP, [DT, NP])
>>> production3 = Production(VP, [V, NP,NP,PP])
>>> production1.lhs()
S
>>> production1.rhs()
(NP, VP)
>>> production3.lhs()
VP
>>> production3.rhs()
(V, NP, NP, PP)
>>> production3 == Production(VP, [V,NP,NP,PP])
True
>>> production2 == production3
False
```

An example for accessing ATIS grammar in NLTK is as follows:

```
>>> import nltk
>>> gram1 = nltk.data.load('grammars/large_grammars/atis.cfg')
>>> gram1
<Grammar with 5517 productions>
```

Extract the testing sentences from ATIS as follows:

```
>>> import nltk
>>> sent = nltk.data.load('grammars/large_grammars/atis_sentences.
txt')
>>> sent = nltk.parse.util.extract_test_sentences(sent)
>>> len(sent)
98
>>> testingsent=sent[25]
>>> testingsent[1]
11
>>> testingsent[0]
['list', 'those', 'flights', 'that', 'stop', 'over', 'in', 'salt',
'lake', 'city', '.']
>>> sent=testingsent[0]
```

Bottom-up parsing:

```
>>> import nltk
>>> gram1 = nltk.data.load('grammars/large_grammars/atis.cfg')
>>> sent = nltk.data.load('grammars/large_grammars/atis_sentences.
txt')
>>> sent = nltk.parse.util.extract_test_sentences(sent)
>>> testingsent=sent[25]
```

```
>>> sent=testingsent[0]
>>> parser1 = nltk.parse.BottomUpChartParser(gram1)
>>> chart1 = parser1.chart_parse(sent)
>>> print((chart1.num_edges()))
13454
>>> print((len(list(chart1.parses(gram1.start())))))
11
```

Bottom-up, Left Corner parsing:

```
>>> import nltk
>>> gram1 = nltk.data.load('grammars/large_grammars/atis.cfg')
>>> sent = nltk.data.load('grammars/large_grammars/atis_sentences.
txt')
>>> sent = nltk.parse.util.extract_test_sentences(sent)
>>> testingsent=sent[25]
>>> sent=testingsent[0]
>>> parser2 = nltk.parse.BottomUpLeftCornerChartParser(gram1)
>>> chart2 = parser2.chart_parse(sent)
>>> print((chart2.num_edges()))
8781
>>> print((len(list(chart2.parses(gram1.start())))))
11
```

Left Corner parsing with a Bottom-up filter:

```
>>> import nltk
>>> gram1 = nltk.data.load('grammars/large_grammars/atis.cfg')
>>> sent = nltk.data.load('grammars/large_grammars/atis_sentences.
txt')
>>> sent = nltk.parse.util.extract_test_sentences(sent)
>>> testingsent=sent[25]
>>> sent=testingsent[0]
>>> parser3 = nltk.parse.LeftCornerChartParser(gram1)
>>> chart3 = parser3.chart_parse(sent)
>>> print((chart3.num_edges()))
1280
>>> print((len(list(chart3.parses(gram1.start())))))
11
```

Top-down parsing:

```
>>> import nltk
>>> gram1 = nltk.data.load('grammars/large_grammars/atis.cfg')
>>> sent = nltk.data.load('grammars/large_grammars/atis_sentences.
txt')
```

```
>>> sent = nltk.parse.util.extract_test_sentences(sent)
>>> testingsent=sent[25]
>>> sent=testingsent[0]
>>>parser4 = nltk.parse.TopDownChartParser(gram1)
>>> chart4 = parser4.chart_parse(sent)
>>> print((chart4.num_edges()))
37763
>>> print((len(list(chart4.parses(gram1.start())))))
11
```

Incremental Bottom-up parsing:

```
>>> import nltk
>>> gram1 = nltk.data.load('grammars/large_grammars/atis.cfg')
>>> sent = nltk.data.load('grammars/large_grammars/atis_sentences.
txt')
>>> sent = nltk.parse.util.extract_test_sentences(sent)
>>> testingsent=sent[25]
>>> sent=testingsent[0]
>>> parser5 = nltk.parse.IncrementalBottomUpChartParser(gram1)
>>> chart5 = parser5.chart_parse(sent)
>>> print((chart5.num_edges()))
13454
>>> print((len(list(chart5.parses(gram1.start())))))
11
```

Incremental Bottom-up, Left Corner parsing:

```
>>> import nltk
>>> gram1 = nltk.data.load('grammars/large_grammars/atis.cfg')
>>> sent = nltk.data.load('grammars/large_grammars/atis_sentences.
txt')
>>> sent = nltk.parse.util.extract_test_sentences(sent)
>>> testingsent=sent[25]
>>> sent=testingsent[0]
>>> parser6 = nltk.parse.IncrementalBottomUpLeftCornerChartParser(gr
am1)
>>> chart6 = parser6.chart_parse(sent)
>>> print((chart6.num_edges()))
8781
>>> print((len(list(chart6.parses(gram1.start())))))
11
```

Incremental Left Corner parsing with a Bottom-up filter:

```
>>> import nltk
>>> gram1 = nltk.data.load('grammars/large_grammars/atis.cfg')
>>> sent = nltk.data.load('grammars/large_grammars/atis_sentences.
txt')
>>> sent = nltk.parse.util.extract_test_sentences(sent)
>>> testingsent=sent[25]
>>> sent=testingsent[0]
>>> parser7 = nltk.parse.IncrementalLeftCornerChartParser(gram1)
>>> chart7 = parser7.chart_parse(sent)
>>> print((chart7.num_edges()))
1280
>>> print((len(list(chart7.parses(gram1.start())))))
11
```

Incremental Top-down parsing:

```
>>> import nltk
>>> gram1 = nltk.data.load('grammars/large_grammars/atis.cfg')
>>> sent = nltk.data.load('grammars/large_grammars/atis_sentences.
txt')
>>> sent = nltk.parse.util.extract_test_sentences(sent)
>>> testingsent=sent[25]
>>> sent=testingsent[0]
>>> parser8 = nltk.parse.IncrementalTopDownChartParser(gram1)
>>> chart8 = parser8.chart_parse(sent)
>>> print((chart8.num_edges()))
37763
>>> print((len(list(chart8.parses(gram1.start())))))
11
```

Earley parsing:

```
>>> import nltk
>>> gram1 = nltk.data.load('grammars/large_grammars/atis.cfg')
>>> sent = nltk.data.load('grammars/large_grammars/atis_sentences.
txt')
>>> sent = nltk.parse.util.extract_test_sentences(sent)
>>> testingsent=sent[25]
>>> sent=testingsent[0]
>>> parser9 = nltk.parse.EarleyChartParser(gram1)
>>> chart9 = parser9.chart_parse(sent)
>>> print((chart9.num_edges()))
37763
>>> print((len(list(chart9.parses(gram1.start())))))
11
```

Creating a probabilistic Context Free Grammar from CFG

In **Probabilistic Context-free Grammar** (**PCFG**), probabilities are attached to all the production rules present in CFG. The sum of these probabilities is 1. It generates the same parse structures as CFG, but it also assigns a probability to each parse tree. The probability of a parsed tree is obtained by taking the product of probabilities of all the production rules used in building the tree.

Let's see the following code in NLTK, that illustrates the formation of rules in PCFG:

```
>>> import nltk
>>> from nltk.corpus import treebank
>>> from itertools import islice
>>> from nltk.grammar import PCFG, induce_pcfg, toy_pcfg1, toy_pcfg2
>>> gram2 = PCFG.from string("""
A -> B B [.3] | C B C [.7]
B -> B D [.5] | C [.5]
C -> 'a' [.1] | 'b' [0.9]
D -> 'b' [1.0]
""")
>>> prod1 = gram2.productions()[0]
>>> prod1
A -> B B [0.3]
>>> prod2 = gram2.productions()[1]
>>> prod2
A -> C B C [0.7]
>>> prod2.lhs()
A
>>> prod2.rhs()
(C, B, C)
>>> print((prod2.prob()))
0.7
>>> gram2.start()
A
>>> gram2.productions()
[A -> B B [0.3], A -> C B C [0.7], B -> B D [0.5], B -> C [0.5], C ->
'a' [0.1], C -> 'b' [0.9], D -> 'b' [1.0]]
```

Let's see the code in NLTK that illustrates Probabilistic Chart Parsing:

```
>>> import nltk
>>> from nltk.corpus import treebank
>>> from itertools import islice
>>> from nltk.grammar import PCFG, induce_pcfg, toy_pcfg1, toy_pcfg2
>>> tokens = "Jack told Bob to bring my cookie".split()
>>> grammar = toy_pcfg2
>>> print(grammar)
Grammar with 23 productions (start state = S)
    S -> NP VP [1.0]
    VP -> V NP [0.59]
    VP -> V [0.4]
    VP -> VP PP [0.01]
    NP -> Det N [0.41]
    NP -> Name [0.28]
    NP -> NP PP [0.31]
    PP -> P NP [1.0]
    V -> 'saw' [0.21]
    V -> 'ate' [0.51]
    V -> 'ran' [0.28]
    N -> 'boy' [0.11]
    N -> 'cookie' [0.12]
    N -> 'table' [0.13]
    N -> 'telescope' [0.14]
    N -> 'hill' [0.5]
    Name -> 'Jack' [0.52]
    Name -> 'Bob' [0.48]
    P -> 'with' [0.61]
    P -> 'under' [0.39]
    Det -> 'the' [0.41]
    Det -> 'a' [0.31]
    Det -> 'my' [0.28]
```

CYK chart parsing algorithm

The drawback of Recursive Descent Parsing is that it causes the Left Recursion Problem and is very complex. So, CYK chart parsing was introduced. It uses the Dynamic Programming approach. CYK is one of the simplest chart parsing algorithms. The CYK algorithm is capable of constructing a chart in *O(n3)* time. Both CYK and Earley are Bottom-up chart parsing algorithms. But, the Earley algorithm also makes use of Top-down predictions when invalid parses are constructed.

Consider the following example of CYK parsing:

```
tok = ["the", "kids", "opened", "the", "box", "on", "the", "floor"]
gram = nltk.parse_cfg("""
S -> NP VP
NP -> Det N | NP PP
VP -> V NP | VP PP
PP -> P NP
Det -> 'the'
N -> 'kids' | 'box' | 'floor'
V -> 'opened' P -> 'on'
 """)
```

Consider the following code to construct the initializing table:

```
def init_nfst(tok, gram):
numtokens1 = len(tok)
 # fill w/ dots
nfst = [["." for i in range(numtokens1+1)] !!!!!!! for j in
range(numtokens1+1)]
# fill in diagonal
for i in range(numtokens1):
prod= gram.productions(rhs=tok[i])
nfst[i][i+1] = prod[0].lhs()
return nfst
```

Consider the following code to fill in the table:

```
def complete_nfst(nfst, tok, trace=False):
index1 = {} for prod in gram.productions():
#make lookup reverse
index1[prod.rhs()] = prod.lhs()
numtokens1 = len(tok) for span in range(2, numtokens1+1):
for start in range(numtokens1+1-span):
#go down towards diagonal
end1 = start1 + span for mid in range(start1+1, end1):
nt1, nt2 = nfst[start1][mid1], nfst[mid1][end1]
if (nt1,nt2) in index1:
if trace:
print "[%s] %3s [%s] %3s [%s] ==> [%s] %3s [%s]" % \ (start, nt1,
mid1, nt2, end1, start1, index[(nt1,nt2)], end) nfst[start1][end1] =
index[(nt1,nt2)]
return nfst
```

Following is the code in Python for constructing the display table:

```
def display(wfst, tok):
print '\nWFST ' + ' '.join([("%-4d" % i) for i in range(1,
len(wfst))])
for i in range(len(wfst)-1):
print "%d " % i,
for j in range(1, len(wfst)):
print "%-4s" % wfst[i][j],
print
```

The result can be obtained from the following code:

```
tok = ["the", "kids", "opened", "the", "box", "on", "the", "floor"]
res1 = init_wfst(tok, gram)
display(res1, tok)
res2 = complete_wfst(res1,tok)
display(res2, tok)
```

Earley chart parsing algorithm

Earley algorithm was given by Earley in 1970. This algorithm is similar to Top-down parsing. It can handle left-recursion, and it doesn't need CNF. It fills in a chart in the left to right manner.

Consider an example that illustrates parsing using the Earley chart parser:

```
>>> import nltk
>>> nltk.parse.earleychart.demo(print_times=False, trace=1,sent='I saw
a dog', numparses=2)
* Sentence:
I saw a dog
['I', 'saw', 'a', 'dog']

|.    I    .   saw   .    a    .   dog   .|
|[---------]         .         .         .| [0:1] 'I'
|.         [---------]         .         .| [1:2] 'saw'
|.         .         [---------]         .| [2:3] 'a'
|.         .         .         [---------]| [3:4] 'dog'
|>         .         .         .         .| [0:0] S  -> * NP VP
|>         .         .         .         .| [0:0] NP -> * NP PP
|>         .         .         .         .| [0:0] NP -> * Det Noun
|>         .         .         .         .| [0:0] NP -> * 'I'
|[---------]         .         .         .| [0:1] NP -> 'I' *
|[---------->         .         .         .| [0:1] S  -> NP * VP
```

```
|[---------->         .         .         .| [0:1] NP -> NP * PP
|.         >         .         .         .| [1:1] VP -> * VP PP
|.         >         .         .         .| [1:1] VP -> * Verb NP
|.         >         .         .         .| [1:1] VP -> * Verb
|.         >         .         .         .| [1:1] Verb -> * 'saw'
|.         [---------]         .         .| [1:2] Verb -> 'saw' *
|.         [---------->        .         .| [1:2] VP -> Verb * NP
|.         [---------]         .         .| [1:2] VP -> Verb *
|[--------------------]         .         .| [0:2] S  -> NP VP *
|.         [---------->        .         .| [1:2] VP -> VP * PP
|.         .         >         .         .| [2:2] NP -> * NP PP
|.         .         >         .         .| [2:2] NP -> * Det Noun
|.         .         >         .         .| [2:2] Det -> * 'a'
|.         .         [---------]         .| [2:3] Det -> 'a' *
|.         .         [---------->        .| [2:3] NP -> Det * Noun
|.         .         .         >         .| [3:3] Noun -> * 'dog'
|.         .         .         [---------]| [3:4] Noun -> 'dog' *
|.         .         [--------------------]| [2:4] NP -> Det Noun *
|.         [------------------------------]| [1:4] VP -> Verb NP *
|.         .         [------------------->| [2:4] NP -> NP * PP
|[======================================]| [0:4] S  -> NP VP *
|.         [------------------------------>| [1:4] VP -> VP * PP
```

Consider an example that illustrates parsing using the Chart parser in NLTK:

```
>>> import nltk
>>> nltk.parse.chart.demo(2, print_times=False, trace=1,sent='John saw
a dog', numparses=1)
* Sentence:
John saw a dog
['John', 'saw', 'a', 'dog']

* Strategy: Bottom-up

|.   John  .   saw   .    a    .   dog   .|
|[---------]         .         .         .| [0:1] 'John'
|.         [---------]         .         .| [1:2] 'saw'
|.         .         [---------]         .| [2:3] 'a'
|.         .         .         [---------]| [3:4] 'dog'
|>         .         .         .         .| [0:0] NP -> * 'John'
|[---------]         .         .         .| [0:1] NP -> 'John' *
|>         .         .         .         .| [0:0] S  -> * NP VP
|>         .         .         .         .| [0:0] NP -> * NP PP
|[---------->        .         .         .| [0:1] S  -> NP * VP
```

```
|[---------->         .         .         .| [0:1] NP -> NP * PP
|.         >         .         .         .| [1:1] Verb -> * 'saw'
|.         [---------]         .         .| [1:2] Verb -> 'saw' *
|.         >         .         .         .| [1:1] VP -> * Verb NP
|.         >         .         .         .| [1:1] VP -> * Verb
|.         [---------->         .         .| [1:2] VP -> Verb * NP
|.         [---------]         .         .| [1:2] VP -> Verb *
|.         >         .         .         .| [1:1] VP -> * VP PP
|[-------------------]         .         .| [0:2] S  -> NP VP *
|.         [---------->         .         .| [1:2] VP -> VP * PP
|.         .         >         .         .| [2:2] Det -> * 'a'
|.         .         [---------]         .| [2:3] Det -> 'a' *
|.         .         >         .         .| [2:2] NP -> * Det Noun
|.         .         [---------->         .| [2:3] NP -> Det * Noun
|.         .         .         >         .| [3:3] Noun -> * 'dog'
|.         .         .         [---------]| [3:4] Noun -> 'dog' *
|.         .         [-------------------]| [2:4] NP -> Det Noun *
|.         .         >         .         .| [2:2] S  -> * NP VP
|.         .         >         .         .| [2:2] NP -> * NP PP
|.         [-----------------------------]| [1:4] VP -> Verb NP *
|.         .         [-------------------->| [2:4] S  -> NP * VP
|.         .         [-------------------->| [2:4] NP -> NP * PP
|[=====================================]| [0:4] S  -> NP VP *
|.         [----------------------------->| [1:4] VP -> VP * PP
Nr edges in chart: 33
(S (NP John) (VP (Verb saw) (NP (Det a) (Noun dog))))
```

Consider an example that illustrates parsing using the Stepping Chart parser in NLTK:

```
>>> import nltk
>>> nltk.parse.chart.demo(5, print_times=False, trace=1,sent='John saw
a dog', numparses=2)
* Sentence:
John saw a dog
['John', 'saw', 'a', 'dog']

* Strategy: Stepping (top-down vs bottom-up)

*** SWITCH TO TOP DOWN
|[---------]         .         .         .| [0:1] 'John'
|.         [---------]         .         .| [1:2] 'saw'
|.         .         [---------]         .| [2:3] 'a'
|.         .         .         [---------]| [3:4] 'dog'
|>         .         .         .         .| [0:0] S  -> * NP VP
```

```
|>          .         .         .         .| [0:0] NP -> * NP PP
|>          .         .         .         .| [0:0] NP -> * Det Noun
|>          .         .         .         .| [0:0] NP -> * 'John'
|[---------]         .         .         .| [0:1] NP -> 'John' *
|[--------->         .         .         .| [0:1] S  -> NP * VP
|[--------->         .         .         .| [0:1] NP -> NP * PP
|.         >         .         .         .| [1:1] VP -> * VP PP
|.         >         .         .         .| [1:1] VP -> * Verb NP
|.         >         .         .         .| [1:1] VP -> * Verb
|.         >         .         .         .| [1:1] Verb -> * 'saw'
|.         [---------]         .         .| [1:2] Verb -> 'saw' *
|.         [--------->         .         .| [1:2] VP -> Verb * NP
|.         [---------]         .         .| [1:2] VP -> Verb *
|[-------------------]         .         .| [0:2] S  -> NP VP *
|.         [--------->         .         .| [1:2] VP -> VP * PP
|.         .         >         .         .| [2:2] NP -> * NP PP
|.         .         >         .         .| [2:2] NP -> * Det Noun
*** SWITCH TO BOTTOM UP
|.         .         >         .         .| [2:2] Det -> * 'a'
|.         .         .         >         .| [3:3] Noun -> * 'dog'
|.         .         [---------]         .| [2:3] Det -> 'a' *
|.         .         .         [---------]| [3:4] Noun -> 'dog' *
|.         .         [--------->         .| [2:3] NP -> Det * Noun
|.         .         [-------------------]| [2:4] NP -> Det Noun *
|.         [-----------------------------]| [1:4] VP -> Verb NP *
|.         .         [------------------->| [2:4] NP -> NP * PP
|[=====================================]| [0:4] S  -> NP VP *
|.         [------------------------------>| [1:4] VP -> VP * PP
|.         .         >         .         .| [2:2] S  -> * NP VP
|.         .         [-------------------->| [2:4] S  -> NP * VP
*** SWITCH TO TOP DOWN
|.         .         .         .         >| [4:4] VP -> * VP PP
|.         .         .         .         >| [4:4] VP -> * Verb NP
|.         .         .         .         >| [4:4] VP -> * Verb
*** SWITCH TO BOTTOM UP
*** SWITCH TO TOP DOWN
*** SWITCH TO BOTTOM UP
*** SWITCH TO TOP DOWN
*** SWITCH TO BOTTOM UP
*** SWITCH TO TOP DOWN
*** SWITCH TO BOTTOM UP
Nr edges in chart: 37
```

Let's see the code for Feature chart parsing in NLTK:

```
>>> import nltk
>>>nltk.parse.featurechart.demo(print_times=False,print_
grammar=True,parser=nltk.parse.featurechart.FeatureChartParser,sent='I
saw a dog')

Grammar with 18 productions (start state = S[])
    S[] -> NP[] VP[]
    PP[] -> Prep[] NP[]
    NP[] -> NP[] PP[]
    VP[] -> VP[] PP[]
    VP[] -> Verb[] NP[]
    VP[] -> Verb[]
    NP[] -> Det[pl=?x] Noun[pl=?x]
    NP[] -> 'John'
    NP[] -> 'I'
    Det[] -> 'the'
    Det[] -> 'my'
    Det[-pl] -> 'a'
    Noun[-pl] -> 'dog'
    Noun[-pl] -> 'cookie'
    Verb[] -> 'ate'
    Verb[] -> 'saw'
    Prep[] -> 'with'
    Prep[] -> 'under'

* FeatureChartParser
Sentence: I saw a dog
|. I .saw. a .dog.|
|[---]   .   .   .| [0:1] 'I'
|.   [---]   .   .| [1:2] 'saw'
|.   .   [---]   .| [2:3] 'a'
|.   .   .   [---]| [3:4] 'dog'
|[---]   .   .   .| [0:1] NP[] -> 'I' *
|[--->   .   .   .| [0:1] S[] -> NP[] * VP[] {}
|[--->   .   .   .| [0:1] NP[] -> NP[] * PP[] {}
|.   [---]   .   .| [1:2] Verb[] -> 'saw' *
|.   [--->   .   .| [1:2] VP[] -> Verb[] * NP[] {}
|.   [---]   .   .| [1:2] VP[] -> Verb[] *
|.   [--->   .   .| [1:2] VP[] -> VP[] * PP[] {}
|[-------]   .   .| [0:2] S[] -> NP[] VP[] *
|.   .   [---]   .| [2:3] Det[-pl] -> 'a' *
|.   .   [--->   .| [2:3] NP[] -> Det[pl=?x] * Noun[pl=?x] {?x: False}
```

```
|.   .   .   [---]| [3:4] Noun[-pl] -> 'dog' *
|.   .   [-------]| [2:4] NP[] -> Det[-pl] Noun[-pl] *
|.   .   [------->| [2:4] S[] -> NP[] * VP[] {}
|.   .   [------->| [2:4] NP[] -> NP[] * PP[] {}
|.   [-----------]| [1:4] VP[] -> Verb[] NP[] *
|.   [----------->| [1:4] VP[] -> VP[] * PP[] {}
|[===============]| [0:4] S[] -> NP[] VP[] *
(S[]
  (NP[] I)
  (VP[] (Verb[] saw) (NP[] (Det[-pl] a) (Noun[-pl] dog))))
```

The following code is found in NLTK for the implementation of the Earley algorithm:

```
def demo(print_times=True, print_grammar=False,
         print_trees=True, trace=2,
         sent='I saw John with a dog with my cookie', numparses=5):
    """
    A demonstration of the Earley parsers.
    """
    import sys, time
    from nltk.parse.chart import demo_grammar

    # The grammar for ChartParser and SteppingChartParser:
    grammar = demo_grammar()
    if print_grammar:
        print("* Grammar")
        print(grammar)

    # Tokenize the sample sentence.
    print("* Sentence:")
    print(sent)
    tokens = sent.split()
    print(tokens)
    print()

    # Do the parsing.
    earley = EarleyChartParser(grammar, trace=trace)
    t = time.clock()
    chart = earley.chart_parse(tokens)
    parses = list(chart.parses(grammar.start()))
    t = time.clock()-t

    # Print results.
    if numparses:
        assert len(parses)==numparses, 'Not all parses found'
    if print_trees:
```

```
            for tree in parses: print(tree)
        else:
            print("Nr trees:", len(parses))
        if print_times:
            print("Time:", t)

if __name__ == '__main__': demo()
```

Summary

In this chapter, we discussed Parsing, accessing the Treebank Corpus, and the implementation of Context-free Grammar, Probabilistic Context-free Grammar, the CYK algorithm, and the Earley algorithm. Hence, in this chapter, we discussed about the syntactic analysis phase of NLP.

In the next chapter, we will discuss about semantic analysis, which is another phase of NLP. We will discuss about NER using different approaches and obtain ways for performing disambiguation tasks.

6
Semantic Analysis – Meaning Matters

Semantic analysis, or meaning generation is one of the tasks in NLP. It is defined as the process of determining the meaning of character sequences or word sequences. It may be used for performing the task of disambiguation.

This chapter will include the following topics:

- NER
- NER system using the HMM
- Training NER using machine learning toolkits
- NER using POS tagging
- Generation of the synset id from Wordnet
- Disambiguating senses using Wordnet

Introducing semantic analysis

NLP means performing computations on natural language. One of the steps performed while processing a natural language is semantic analysis. While analyzing an input sentence, if the syntactic structure of a sentence is built, then the semantic analysis of a sentence will be done. Semantic interpretation means mapping a meaning to a sentence. Contextual interpretation is mapping the logical form to the knowledge representation. The primitive or the basic unit of semantic analysis is referred to as meaning or sense. One of the tools dealing with senses is ELIZA. ELIZA was developed in the sixties by Joseph Weizenbaum. It made use of substitution and pattern matching techniques to analyze the sentence and provide an output to the given input. MARGIE was developed by Robert Schank in the seventies. It could represent all the English verbs using 11 primitives. MARGIE could interpret the sense of a sentence and represent it with the help of primitives. It further gave way to the concept of scripts. From MARGIE, **Script Applier Mechanism** (**SAM**) was developed. It could translate a sentence from different languages, such as English, Chinese, Russian, Dutch, and Spanish. In order to perform processing on textual data, a Python library or TextBlob is used. TextBlob provides APIs for performing NLP tasks, such as Part-of-Speech tagging, extraction of Noun Phrases, classification, machine translation, sentiment analysis.

Semantic analysis can be used to query a database and retrieve information. Another Python library, Gensim, can be used to perform document indexing, topic modeling, and similarity retrieval. Polyglot is an NLP tool that supports various multilingual applications. It provides NER for 40 different languages, tokenization for 165 different languages, language detection for 196 different languages, sentiment analysis for 136 different languages, POS tagging for 16 different languages, Morphological Analysis for 135 different languages, word embedding for 137 different languages, and transliteration for 69 different languages. MontyLingua is an NLP tool that is used to perform the semantic interpretation of English text. From English sentences, it extracts semantic information, such as verbs, nouns, adjectives, dates, phrases, and so on.

Sentences can be formally represented using logics. The basic expressions or sentences in propositional logic are represented using propositional symbols, such as P,Q, R, and so on. Complex expressions in propositional logic can be represented using Boolean operators. For example, to represent the sentence *If it is raining, I'll wear a raincoat* using propositional logic:

- **P**: It is raining.
- **Q**: I'll wear raincoat.
- **P→Q**: If it is raining, I'll wear a raincoat.

Consider the following code to represent operators used in NLTK:

```
>>> import nltk
>>> nltk.boolean_ops()
negation  -
conjunction      &
disjunction      |
implication      ->
equivalence      <->
```

Well-formed Formulas (**WFF**) are formed using propositional symbols or using a combination of propositional symbols and Boolean operators.

Let's see the following code in NLTK, that categorizes logical expressions into different subclasses:

```
>>> import nltk
>>> input_expr = nltk.sem.Expression.from string
>>> input_expr('X | (Y -> Z)')
<OrExpression (X | (Y -> Z))>
>>> input_expr('-(X & Y)')
<NegatedExpression -(X & Y)>
>>> input_expr('X & Y')
<AndExpression (X & Y)>
>>> input_expr('X <-> -- X')
<IffExpression (X <-> --X)>
```

For mapping `True` or `False` values to logical expressions, the `Valuation` function is used in NLTK:

```
>>> import nltk
>>> value = nltk.Valuation([('X', True), ('Y', False), ('Z', True)])
>>> value['Z']
True
>>> domain = set()
>>> v = nltk.Assignment(domain)
>>> u = nltk.Model(domain, value)
>>> print(u.evaluate('(X & Y)', v))
False
>>> print(u.evaluate('-(X & Y)', v))
True
>>> print(u.evaluate('(X & Z)', v))
True
>>> print(u.evaluate('(X | Y)', v))
True
```

First order predicate logic involving constants and predicates in NLTK are depicted in the following code:

```
>>> import nltk
>>> input_expr = nltk.sem.Expression.from string
>>> expression = input_expr('run(marcus)', type_check=True)
>>> expression.argument
<ConstantExpressionmarcus>
>>> expression.argument.type
e
>>> expression.function
<ConstantExpression run>
>>> expression.function.type
<e,?>
>>> sign = {'run': '<e, t>'}
>>> expression = input_expr('run(marcus)', signature=sign)
>>> expression.function.type
e
```

The `signature` is used in NLTK to map associated types and non-logical constants. Consider the following code in NLTK that helps to generate a query and retrieve data from the database:

```
>>> import nltk
>>> nltk.data.show_cfg('grammars/book_grammars/sql1.fcfg')
% start S
S[SEM=(?np + WHERE + ?vp)] -> NP[SEM=?np] VP[SEM=?vp]
VP[SEM=(?v + ?pp)] -> IV[SEM=?v] PP[SEM=?pp]
VP[SEM=(?v + ?ap)] -> IV[SEM=?v] AP[SEM=?ap]
VP[SEM=(?v + ?np)] -> TV[SEM=?v] NP[SEM=?np]
VP[SEM=(?vp1 + ?c + ?vp2)] -> VP[SEM=?vp1] Conj[SEM=?c] VP[SEM=?vp2]
NP[SEM=(?det + ?n)] ->Det[SEM=?det] N[SEM=?n]
NP[SEM=(?n + ?pp)]  -> N[SEM=?n] PP[SEM=?pp]
NP[SEM=?n]  -> N[SEM=?n]  | CardN[SEM=?n]
CardN[SEM='1000'] -> '1,000,000'
PP[SEM=(?p + ?np)] -> P[SEM=?p] NP[SEM=?np]
AP[SEM=?pp] -> A[SEM=?a] PP[SEM=?pp]
NP[SEM='Country="greece"'] -> 'Greece'
NP[SEM='Country="china"'] -> 'China'
Det[SEM='SELECT'] -> 'Which' | 'What'
Conj[SEM='AND'] -> 'and'
N[SEM='City FROM city_table'] -> 'cities'
N[SEM='Population'] -> 'populations'
IV[SEM=''] -> 'are'
```

```
TV[SEM=''] -> 'have'
A -> 'located'
P[SEM=''] -> 'in'
P[SEM='>'] -> 'above'
>>> from nltk import load_parser
>>> test = load_parser('grammars/book_grammars/sql1.fcfg')
>>> q=" What cities are in Greece"
>>> t = list(test.parse(q.split()))
>>> ans = t[0].label()['SEM']
>>> ans = [s for s in ans if s]
>>> q = ' '.join(ans)
>>> print(q)
SELECT City FROM city_table WHERE Country="greece"
>>> from nltk.sem import chat80
>>> r = chat80.sql_query('corpora/city_database/city.db', q)
>>> for p in r:
print(p[0], end=" ")

athens
```

Introducing NER

Named entity recognition (NER) is the process in which proper nouns or named entities are located in a document. Then, these Named Entities are classified into different categories, such as Name of Person, Location, Organization, and so on.

There are 12 NER tagsets defined by IIIT-Hyderabad IJCNLP 2008. These are described here:

SNO.	Named entity tag	Meaning
1	NEP	Name of Person
2	NED	Name of Designation
3	NEO	Name of Organization
4	NEA	Name of Abbreviation
5	NEB	Name of Brand
6	NETP	Title of Person
7	NETO	Title of Object
8	NEL	Name of Location

SNO.	Named entity tag	Meaning
9	NETI	Time
10	NEN	Number
11	NEM	Measure
12	NETE	Terms

One of the applications of NER is information extraction. In NLTK, we can perform the task of information extraction by storing the tuple (entity, relation, entity), and then, the entity value can be retrieved.

Consider an example in NLTK that shows how information extraction is performed:

```
>>> import nltk
>>> locations=[('Jaipur', 'IN', 'Rajasthan'),('Ajmer', 'IN',
'Rajasthan'),('Udaipur', 'IN', 'Rajasthan'),('Mumbai', 'IN',
'Maharashtra'),('Ahmedabad', 'IN', 'Gujrat')]
>>> q = [x1 for (x1, relation, x2) in locations if x2=='Rajasthan']
>>> print(q)
['Jaipur', 'Ajmer', 'Udaipur']
```

The `nltk.tag.stanford` module is used that makes use of stanford taggers to perform NER. We can download tagger models from `http://nlp.stanford.edu/software`.

Let's see the following example in NLTK that can be used to perform NER using the `Stanford` tagger:

```
>>> from nltk.tag import StanfordNERTagger
>>> sentence = StanfordNERTagger('english.all.3class.distsim.crf.ser.
gz')
>>> sentence.tag('John goes to NY'.split())
[('John', 'PERSON'), ('goes', 'O'), ('to', 'O'),('NY', 'LOCATION')]
```

A classifier has been trained in NLTK to detect Named Entities. Using the function `nltk.ne.chunk()`, named entities can be identified from a text. If the parameter binary is set to `true`, then the named entities are detected and tagged with the `NE` tag; otherwise the named entities are tagged with tags such as PERSON, GPE, and ORGANIZATION.

Let's see the following code, that detects Named Entities, if they exist, and tags them with the `NE` tag:

```
>>> import nltk
>>> sentences1 = nltk.corpus.treebank.tagged_sents()[17]
>>> print(nltk.ne_chunk(sentences1, binary=True))
(S
  The/DT
total/NN
of/IN
  18/CD
deaths/NNS
from/IN
malignant/JJ
mesothelioma/NN
  ,/,
lung/NN
cancer/NN
and/CC
asbestosis/NN
was/VBD
far/RB
higher/JJR
than/IN
  */-NONE-
expected/VBN
  *?*/-NONE-
  ,/,
the/DT
researchers/NNS
said/VBD
  0/-NONE-
  *T*-1/-NONE-
  ./.)
>>> sentences2 = nltk.corpus.treebank.tagged_sents()[7]
>>> print(nltk.ne_chunk(sentences2, binary=True))
(S
  A/DT
  (NE Lorillard/NNP)
spokewoman/NN
said/VBD
  ,/,
  ``/``
  This/DT
```

```
is/VBZ
an/DT
old/JJ
story/NN
  ./.)
>>> print(nltk.ne_chunk(sentences2))
(S
  A/DT
  (ORGANIZATION Lorillard/NNP)
spokewoman/NN
said/VBD
  ,/,
  ``/``
  This/DT
is/VBZ
an/DT
old/JJ
story/NN
  ./.)
```

Consider another example in NLTK that can be used to detect named entities:

```
>>> import nltk
>>> from nltk.corpus import conll2002
>>> for documents in conll2002.chunked_sents('ned.train')[25]:
print(documents)


(PER Vandenbussche/Adj)
('zelf', 'Pron')
('besloot', 'V')
('dat', 'Conj')
('het', 'Art')
('hof', 'N')
('"', 'Punc')
('de', 'Art')
('politieke', 'Adj')
('zeden', 'N')
('uit', 'Prep')
('het', 'Art')
('verleden', 'N')
('"', 'Punc')
('heeft', 'V')
('willen', 'V')
('veroordelen', 'V')
('.', 'Punc')
```

A `chunker` is a program that is used to partition plain text into a sequence of semantically related words. To perform NER in NLTK, default chunkers are used. Default chunkers are chunkers based on classifiers that have been trained on the ACE corpus. Other chunkers have been trained on parsed or chunked NLTK corpora. The languages covered by these NLTK chunkers are as follows:

- Dutch
- Spanish
- Portuguese
- English

Consider another example in NLTK that identifies named entities and categorizes into different named entity classes:

```
>>> import nltk
>>> sentence = "I went to Greece to meet John";
>>> tok=nltk.word_tokenize(sentence)
>>> pos_tag=nltk.pos_tag(tok)
>>> print(nltk.ne_chunk(pos_tag))
(S
  I/PRP
went/VBD
to/TO
  (GPE Greece/NNP)
to/TO
meet/VB
  (PERSON John/NNP))
```

A NER system using Hidden Markov Model

HMM is one of the popular statistical approaches of NER. An HMM is defined as a **Stochastic Finite State Automaton** (**SFSA**) consisting of a finite set of states that are associated with the definite probability distribution. States are unobserved or hidden. HMM generates optimal state sequences as an output. HMM is based on the Markov Chain property. According to the Markov Chain property, the probability of the occurrence of the next state is dependent on the previous tag. It is the simplest approach to implement. The drawback of HMM is that it requires a large amount of training and it cannot be used for large dependencies. HMM consists of the following:

- Set of states, S, where $|S|=N$. Here, N is the total number of states.
- Start state, $S0$.

- Output alphabet, $O; |O| = k$. k is the total number of output alphabets.
- Transition probability, A.
- Emission probability, B.
- Initial state probabilities, π.

HMM is represented by the following tuple—$\lambda = (A, B, \pi)$.

Start probability or initial state probability may be defined as the probability that a particular tag occurs first in a sentence.

Transition probability ($A=aij$) may be defined as the probability of the occurrence of the next tag j in a sentence given the occurrence of the particular tag i at present.

$A=aij$= the number of transitions from state si to sj /the number of transitions from state si

Emission probability ($B=bj(O)$) may be defined as the probability of the occurrence of an output sequence given a state j.

$B=bj(k)$= the number of times in state j and observing the symbol k /the expected number of times in state j.

The Baum Welch algorithm is used to find the maximum likelihood and the posterior mode estimates for HMM parameters. The forward-backward algorithm is used to find the posterior marginals of all the hidden state variables given a sequence of emissions or observations.

There are three steps involved in performing NER using HMM—Annotation, HMM train, and HMM test. The Annotation module converts raw text into annotated or trainable data. During HMM train, we compute HMM parameters—start probability, transition probability, and emission probability. During HMM test, the Viterbi algorithm is used. that finds out the optimal tag sequence.

Consider an example of chunking using the HMM in NLTK. Using chunking, the NP and VP chunks can be obtained. NP chunks can further be processed to obtain proper nouns or named entities:

```
>>> import nltk
>>> nltk.tag.hmm.demo_pos()

HMM POS tagging demo

Training HMM...
Testing...
```

```
Test: the/AT fulton/NP county/NN grand/JJ jury/NN said/VBD friday/
NR an/AT investigation/NN of/IN atlanta's/NP$ recent/JJ primary/NN
election/NN produced/VBD ``/`` no/AT evidence/NN ''/'' that/CS any/DTI
irregularities/NNS took/VBD place/NN ./.

Untagged: the fulton county grand jury said friday an investigation of
atlanta's recent primary election produced `` no evidence '' that any
irregularities took place .

HMM-tagged: the/AT fulton/NP county/NN grand/JJ jury/NN said/
VBD friday/NR an/AT investigation/NN of/IN atlanta's/NP$ recent/
JJ primary/NN election/NN produced/VBD ``/`` no/AT evidence/NN ''/''
that/CS any/DTI irregularities/NNS took/VBD place/NN ./.

Entropy: 18.7331739705

--------------------------------------------------------------
Test: the/AT jury/NN further/RBR said/VBD in/IN term-end/NN
presentments/NNS that/CS the/AT city/NN executive/JJ committee/NN ,/,
which/WDT had/HVD over-all/JJ charge/NN of/IN the/AT election/NN ,/,
``/`` deserves/VBZ the/AT praise/NN and/CC thanks/NNS of/IN the/AT
city/NN of/IN atlanta/NP ''/'' for/IN the/AT manner/NN in/IN which/WDT
the/AT election/NN was/BEDZ conducted/VBN ./.

Untagged: the jury further said in term-end presentments that the
city executive committee , which had over-all charge of the election
, `` deserves the praise and thanks of the city of atlanta '' for the
manner in which the election was conducted .

HMM-tagged: the/AT jury/NN further/RBR said/VBD in/IN term-end/AT
presentments/NN that/CS the/AT city/NN executive/NN committee/NN ,/,
which/WDT had/HVD over-all/VBN charge/NN of/IN the/AT election/NN ,/,
``/`` deserves/VBZ the/AT praise/NN and/CC thanks/NNS of/IN the/AT
city/NN of/IN atlanta/NP ''/'' for/IN the/AT manner/NN in/IN which/WDT
the/AT election/NN was/BEDZ conducted/VBN ./.

Entropy: 27.0708725519

--------------------------------------------------------------
Test: the/AT september-october/NP term/NN jury/NN had/HVD been/BEN
charged/VBN by/IN fulton/NP superior/JJ court/NN judge/NN durwood/
NP pye/NP to/TO investigate/VB reports/NNS of/IN possible/JJ ``/``
irregularities/NNS ''/'' in/IN the/AT hard-fought/JJ primary/NN which/
WDT was/BEDZ won/VBN by/IN mayor-nominate/NN ivan/NP allen/NP jr./NP
./.
```

```
Untagged: the september-october term jury had been charged by fulton
superior court judge durwoodpye to investigate reports of possible ``
irregularities '' in the hard-fought primary which was won by mayor-
nominate ivanallenjr. .

HMM-tagged: the/AT september-october/JJ term/NN jury/NN had/HVD been/
BEN charged/VBN by/IN fulton/NP superior/JJ court/NN judge/NN durwood/
TO pye/VB to/TO investigate/VB reports/NNS of/IN possible/JJ ``/``
irregularities/NNS ''/'' in/IN the/AT hard-fought/JJ primary/NN which/
WDT was/BEDZ won/VBN by/IN mayor-nominate/NP ivan/NP allen/NP jr./NP
./.

Entropy: 33.8281874237

------------------------------------------------------------
Test: ``/`` only/RB a/AT relative/JJ handful/NN of/IN such/JJ reports/
NNS was/BEDZ received/VBN ''/'' ,/, the/AT jury/NN said/VBD ,/, ``/``
considering/IN the/AT widespread/JJ interest/NN in/IN the/AT election/
NN ,/, the/AT number/NN of/IN voters/NNS and/CC the/AT size/NN of/IN
this/DT city/NN ''/'' ./.

Untagged: `` only a relative handful of such reports was received '' ,
the jury said , `` considering the widespread interest in the election
, the number of voters and the size of this city '' .

HMM-tagged: ``/`` only/RB a/AT relative/JJ handful/NN of/IN such/JJ
reports/NNS was/BEDZ received/VBN ''/'' ,/, the/AT jury/NN said/VBD
,/, ``/`` considering/IN the/AT widespread/JJ interest/NN in/IN the/AT
election/NN ,/, the/AT number/NN of/IN voters/NNS and/CC the/AT size/
NN of/IN this/DT city/NN ''/'' ./.

Entropy: 11.4378198596

------------------------------------------------------------
Test: the/AT jury/NN said/VBD it/PPS did/DOD find/VB that/CS many/AP
of/IN georgia's/NP$ registration/NN and/CC election/NN laws/NNS ``/``
are/BER outmoded/JJ or/CC inadequate/JJ and/CC often/RB ambiguous/JJ
''/'' ./.

Untagged: the jury said it did find that many of georgia's
registration and election laws `` are outmoded or inadequate and often
ambiguous '' .

HMM-tagged: the/AT jury/NN said/VBD it/PPS did/DOD find/VB that/CS
many/AP of/IN georgia's/NP$ registration/NN and/CC election/NN laws/
NNS ``/`` are/BER outmoded/VBG or/CC inadequate/JJ and/CC often/RB
ambiguous/VB ''/'' ./.
```

```
Entropy: 20.8163623192

------------------------------------------------------------
Test: it/PPS recommended/VBD that/CS fulton/NP legislators/NNS act/VB
``/`` to/TO have/HV these/DTS laws/NNS studied/VBN and/CC revised/VBN
to/IN the/AT end/NN of/IN modernizing/VBG and/CC improving/VBG them/
PPO ''/'' ./.

Untagged: it recommended that fulton legislators act `` to have these
laws studied and revised to the end of modernizing and improving them
'' .

HMM-tagged: it/PPS recommended/VBD that/CS fulton/NP legislators/
NNS act/VB ``/`` to/TO have/HV these/DTS laws/NNS studied/VBD and/CC
revised/VBD to/IN the/AT end/NN of/IN modernizing/NP and/CC improving/
VBG them/PPO ''/'' ./.

Entropy: 20.3244921203

------------------------------------------------------------
Test: the/AT grand/JJ jury/NN commented/VBD on/IN a/AT number/NN of/
IN other/AP topics/NNS ,/, among/IN them/PPO the/AT atlanta/NP and/
CC fulton/NP county/NN purchasing/VBG departments/NNS which/WDT it/
PPS said/VBD ``/`` are/BER well/QL operated/VBN and/CC follow/VB
generally/RB accepted/VBN practices/NNS which/WDT inure/VB to/IN the/
AT best/JJT interest/NN of/IN both/ABX governments/NNS ''/'' ./.

Untagged: the grand jury commented on a number of other topics ,
among them the atlanta and fulton county purchasing departments which
it said `` are well operated and follow generally accepted practices
which inure to the best interest of both governments '' .

HMM-tagged: the/AT grand/JJ jury/NN commented/VBD on/IN a/AT number/
NN of/IN other/AP topics/NNS ,/, among/IN them/PPO the/AT atlanta/
NP and/CC fulton/NP county/NN purchasing/NN departments/NNS which/WDT
it/PPS said/VBD ``/`` are/BER well/RB operated/VBN and/CC follow/VB
generally/RB accepted/VBN practices/NNS which/WDT inure/VBZ to/IN the/
AT best/JJT interest/NN of/IN both/ABX governments/NNS ''/'' ./.

Entropy: 31.3834231469

------------------------------------------------------------
Test: merger/NN proposed/VBN

Untagged: merger proposed
```

```
HMM-tagged: merger/PPS proposed/VBD

Entropy: 5.6718203946

------------------------------------------------------------
Test: however/WRB ,/, the/AT jury/NN said/VBD it/PPS believes/VBZ
``/`` these/DTS two/CD offices/NNS should/MD be/BE combined/VBN to/TO
achieve/VB greater/JJR efficiency/NN and/CC reduce/VB the/AT cost/NN
of/IN administration/NN ''/'' ./.

Untagged: however , the jury said it believes `` these two offices
should be combined to achieve greater efficiency and reduce the cost
of administration '' .

HMM-tagged: however/WRB ,/, the/AT jury/NN said/VBD it/PPS believes/
VBZ ``/`` these/DTS two/CD offices/NNS should/MD be/BE combined/VBN
to/TO achieve/VB greater/JJR efficiency/NN and/CC reduce/VB the/AT
cost/NN of/IN administration/NN ''/'' ./.

Entropy: 8.27545943909

------------------------------------------------------------
Test: the/AT city/NN purchasing/VBG department/NN ,/, the/AT jury/NN
said/VBD ,/, ``/`` is/BEZ lacking/VBG in/IN experienced/VBN clerical/
JJ personnel/NNS as/CS a/AT result/NN of/IN city/NN personnel/NNS
policies/NNS ''/'' ./.

Untagged: the city purchasing department , the jury said , `` is
lacking in experienced clerical personnel as a result of city
personnel policies '' .

HMM-tagged: the/AT city/NN purchasing/NN department/NN ,/, the/
AT jury/NN said/VBD ,/, ``/`` is/BEZ lacking/VBG in/IN experienced/
AT clerical/JJ personnel/NNS as/CS a/AT result/NN of/IN city/NN
personnel/NNS policies/NNS ''/'' ./.

Entropy: 16.7622537278

------------------------------------------------------------
accuracy over 284 tokens: 92.96
```

The outcome of an NER tagger may be defined as a *response* and an interpretation of human beings as *answer key*. So, we provide the following definitions:

- **Correct**: If the response is exactly the same as answer key
- **Incorrect**: If the response is not same as answer key
- **Missing**: If answer key is found tagged, but response is not tagged
- **Spurious**: If response is found tagged, but answer key is not tagged

Performance of an NER-based system can be judged by using the following parameters:

- **Precision (P)**: It is defined as follows:
  P=Correct/ (Correct+Incorrect+Missing)
- **Recall (R)**: It is defined as follows:
  R=Correct/ (Correct+Incorrect+Spurious)
- **F-Measure**: It is defined as follows:
  *F-Measure = (2*PREC*REC)/(PRE+REC)*

Training NER using Machine Learning Toolkits

NER can be performed using the following approaches:

- Rule-based or Handcrafted approach:
  - List Lookup approach
  - Linguistic approach
- Machine Learning-based approach or Automated approach:
  - Hidden Markov Model
  - Maximum Entropy Markov Model
  - Conditional Random Fields
  - Support Vector Machine
  - Decision Trees

It has been proved experimentally that Machine learning-based approaches outperform Rule-based approaches. Also, if a combination of Rule-based approaches and Machine Learning-based approaches is used, then the performance of NER will increase.

NER using POS tagging

Using POS tagging, NER can be performed. The POS tags that can be used are as follows (they are available at `https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`):

Tag	Description
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun
PRP$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	To
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VBN	Verb, past participle

Tag	Description
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Wh-determiner
WP	Wh-pronoun
WP$	Possessive wh-pronoun
WRB	Wh-adverb

If POS tagging is performed, then using POS information, named entities can be identified. The tokens tagged with the `NNP` tag are Named Entities.

Consider the following example in NLTK in which POS tagging is used to perform NER:

```
>>> import nltk
>>> from nltk import pos_tag, word_tokenize
>>> pos_tag(word_tokenize("John and Smith are going to NY and
Germany"))
[('John', 'NNP'), ('and', 'CC'), ('Smith', 'NNP'), ('are', 'VBP'),
('going', 'VBG'), ('to', 'TO'), ('NY', 'NNP'), ('and', 'CC'),
('Germany', 'NNP')]
```

Here, the named entities are—`John`, `Smith`, `NY`, and `Germany` since they are tagged with the NNP tag.

Let's see another example in which POS tagging is performed in NLTK and the POS tag information is used to detect Named Entities:

```
>>> import nltk
>>> from nltk.corpus import brown
>>> from nltk.tag import UnigramTagger
>>> tagger = UnigramTagger(brown.tagged_sents(categories='news')
[:700])
>>> sentence = ['John','and','Smith','went','to','NY','and','Germany']
>>> for word, tag in tagger.tag(sentence):
print(word,'->',tag)


John -> NP
and -> CC
Smith -> None
went -> VBD
to -> TO
```

```
NY -> None
and -> CC
Germany -> None
```

Here, John has been tagged with the `NP` tag, so it is identified as a named entity. Some of the tokens here are tagged with the `None` tag because these tokens have not been trained.

Generation of the synset id from Wordnet

Wordnet may be defined as an English lexical database. The conceptual dependency between words, such as hypernym, synonym, antonym, and hyponym, can be found using synsets.

Consider the following code in NLTK for the generation of synsets:

```
def all_synsets(self, pos=None):
        """Iterate over all synsets with a given part of speech tag.
        If no pos is specified, all synsets for all parts of speech
        will be loaded.
        """
        if pos is None:
            pos_tags = self._FILEMAP.keys()
        else:
            pos_tags = [pos]

        cache = self._synset_offset_cache
        from_pos_and_line = self._synset_from_pos_and_line

        # generate all synsets for each part of speech
        for pos_tag in pos_tags:
            # Open the file for reading.  Note that we can not re-use
            # the file poitners from self._data_file_map here, because
            # we're defining an iterator, and those file pointers
might
            # be moved while we're not looking.
            if pos_tag == ADJ_SAT:
                pos_tag = ADJ
            fileid = 'data.%s' % self._FILEMAP[pos_tag]
            data_file = self.open(fileid)

            try:
                # generate synsets for each line in the POS file
                offset = data_file.tell()
```

```
                    line = data_file.readline()
                    while line:
                        if not line[0].isspace():
                            if offset in cache[pos_tag]:
                                # See if the synset is cached
                                synset = cache[pos_tag][offset]
                            else:
                                # Otherwise, parse the line
                                synset = from_pos_and_line(pos_tag, line)
                                cache[pos_tag][offset] = synset

                            # adjective satellites are in the same file as
                            # adjectives so only yield the synset if it's
    actually
                            # a satellite
                            if synset._pos == ADJ_SAT:
                                yield synset

                            # for all other POS tags, yield all synsets
    (this means
                            # that adjectives also include adjective
    satellites)
                            else:
                                yield synset
                        offset = data_file.tell()
                        line = data_file.readline()

                # close the extra file handle we opened
                except:
                    data_file.close()
                    raise
                else:
                    data_file.close()
```

Let's see the following code in NLTK, that is used to look up a word using synsets:

```
>>> import nltk
>>> from nltk.corpus import wordnet
>>> from nltk.corpus import wordnet as wn
>>> wn.synsets('cat')
[Synset('cat.n.01'), Synset('guy.n.01'), Synset('cat.n.03'),
Synset('kat.n.01'), Synset('cat-o'-nine-tails.n.01'),
Synset('caterpillar.n.02'), Synset('big_cat.n.01'),
Synset('computerized_tomography.n.01'), Synset('cat.v.01'),
Synset('vomit.v.01')]
```

```
>>> wn.synsets('cat', pos=wn.VERB)
[Synset('cat.v.01'), Synset('vomit.v.01')]
>>> wn.synset('cat.n.01')
Synset('cat.n.01')
```

Here, `cat.n.01` means that `cat` is of the noun category and only one meaning of `cat` exists:

```
>>> print(wn.synset('cat.n.01').definition())
feline mammal usually having thick soft fur and no ability to roar:
domestic cats; wildcats
>>> len(wn.synset('cat.n.01').examples())
0
>>> wn.synset('cat.n.01').lemmas()
[Lemma('cat.n.01.cat'), Lemma('cat.n.01.true_cat')]
>>> [str(lemma.name()) for lemma in wn.synset('cat.n.01').lemmas()]
['cat', 'true_cat']
>>> wn.lemma('cat.n.01.cat').synset()
Synset('cat.n.01')
```

Let's see the following example in NLTK, that depicts the use of Synsets and Open Multilingual Wordnet using ISO 639 language codes:

```
>>> import nltk
>>> from nltk.corpus import wordnet
>>> from nltk.corpus import wordnet as wn
>>> sorted(wn.langs())
['als', 'arb', 'cat', 'cmn', 'dan', 'eng', 'eus', 'fas', 'fin', 'fra',
'fre', 'glg', 'heb', 'ind', 'ita', 'jpn', 'nno', 'nob', 'pol', 'por',
'spa', 'tha', 'zsm']
>>> wn.synset('cat.n.01').lemma_names('ita')
['gatto']
>>> sorted(wn.synset('cat.n.01').lemmas('dan'))
[Lemma('cat.n.01.kat'), Lemma('cat.n.01.mis'), Lemma('cat.n.01.
missekat')]
>>> sorted(wn.synset('cat.n.01').lemmas('por'))
[Lemma('cat.n.01.Gato-doméstico'), Lemma('cat.n.01.Gato_doméstico'),
Lemma('cat.n.01.gato'), Lemma('cat.n.01.gato')]
>>> len(wordnet.all_lemma_names(pos='n', lang='jpn'))
66027
>>> cat = wn.synset('cat.n.01')
>>> cat.hypernyms()
[Synset('feline.n.01')]
>>> cat.hyponyms()
[Synset('domestic_cat.n.01'), Synset('wildcat.n.03')]
```

```
>>> cat.member_holonyms()
[]
>>> cat.root_hypernyms()
[Synset('entity.n.01')]
>>> wn.synset('cat.n.01').lowest_common_hypernyms(wn.
synset('dog.n.01'))
[Synset('carnivore.n.01')]
```

Disambiguating senses using Wordnet

Disambiguation is the task of distinguishing two or more of the same spellings or the same sounding words on the basis of their sense or meaning.

Following are the implementations of disambiguation or the WSD task using Python technologies:

- Lesk algorithms:
  - Original Lesk
  - Cosine Lesk (use cosines to calculate overlaps instead of using raw counts)
  - Simple Lesk (with definitions, example(s), and hyper+hyponyms)
  - Adapted/extended Lesk
  - Enhanced Lesk
- Maximizing similarity:
  - Information content
  - Path similarity
- Supervised WSD:
  - **It Makes Sense** (**IMS**)
  - SVM WSD
- Vector Space models:
  - Topic Models, LDA
  - LSI/LSA
  - NMF
- Graph-based models:
  - Babelfly
  - UKB

- Baselines:
    - Random sense
    - Highest lemma counts
    - First NLTK sense

Wordnet sense similarity in NLTK involves the following algorithms:

- **Resnik Score**: On comparing two tokens, a score (Least Common Subsumer) is returned that decides the similarity of two tokens
- **Wu-Palmer Similarity**: Defines the similarity between two tokens on the basis of the depth of two senses and Least Common Subsumer
- **Path Distance Similarity**: The similarity of two tokens is determined on the basis of the shortest distance that is computed in the is-a taxonomy
- **Leacock Chodorow Similarity**: A similarity score is returned on the basis of the shortest path and the depth (maximum) in which the senses exist in the taxonomy
- **Lin Similarity**: A similarity score is returned on the basis of the information content of the Least Common Subsumer and two input Synsets
- **Jiang-Conrath Similarity**: A similarity score is returned on the basis of the content information of Least Common Subsumer and two input Synsets

Consider the following example in NLTK, which depicts path similarity:

```
>>> import nltk
>>> from nltk.corpus import wordnet
>>> from nltk.corpus import wordnet as wn
>>> lion = wn.synset('lion.n.01')
>>> cat = wn.synset('cat.n.01')
>>> lion.path_similarity(cat)
0.25
```

Consider the following example in NLTK that depicts Leacock Chodorow Similarity:

```
>>> import nltk
>>> from nltk.corpus import wordnet
>>> from nltk.corpus import wordnet as wn
>>> lion = wn.synset('lion.n.01')
>>> cat = wn.synset('cat.n.01')
>>> lion.lch_similarity(cat)
2.2512917986064953
```

Consider the following example in NLTK that depicts Wu-Palmer Similarity:

```
>>> import nltk
>>> from nltk.corpus import wordnet
>>> from nltk.corpus import wordnet as wn
>>> lion = wn.synset('lion.n.01')
>>> cat = wn.synset('cat.n.01')
>>> lion.wup_similarity(cat)
0.896551724137931
```

Consider the following example in NLTK that depicts Resnik Similarity, Lin Similarity, and Jiang-Conrath Similarity:

```
>>> import nltk
>>> from nltk.corpus import wordnet
>>> from nltk.corpus import wordnet as wn
>>> from nltk.corpus import wordnet_ic
>>> brown_ic = wordnet_ic.ic('ic-brown.dat')
>>> semcor_ic = wordnet_ic.ic('ic-semcor.dat')
>>> from nltk.corpus import genesis
>>> genesis_ic = wn.ic(genesis, False, 0.0)
>>> lion = wn.synset('lion.n.01')
>>> cat = wn.synset('cat.n.01')
>>> lion.res_similarity(cat, brown_ic)
8.663481537685325
>>> lion.res_similarity(cat, genesis_ic)
7.339696591781995
>>> lion.jcn_similarity(cat, brown_ic)
0.36425897775957294
>>> lion.jcn_similarity(cat, genesis_ic)
0.3057800856788946
>>> lion.lin_similarity(cat, semcor_ic)
0.8560734335071154
```

Let's see the following code in NLTK based on Wu-Palmer Similarity and Path Distance Similarity:

```
from nltk.corpus import wordnet as wn
def getSenseSimilarity(worda,wordb):

"""

find similarity betwwn word senses of two words

"""
```

```
wordasynsets = wn.synsets(worda)

wordbsynsets = wn.synsets(wordb)

synsetnamea = [wn.synset(str(syns.name)) for syns in wordasynsets]

    synsetnameb = [wn.synset(str(syns.name)) for syns in wordbsynsets]


for sseta, ssetb in [(sseta,ssetb) for sseta in synsetnamea\

for ssetb in synsetnameb]:

pathsim = sseta.path_similarity(ssetb)

wupsim = sseta.wup_similarity(ssetb)

if pathsim != None:

print "Path Sim Score: ",pathsim," WUP Sim Score: ",wupsim,\

"\t",sseta.definition, "\t", ssetb.definition


if __name__ == "__main__":

#getSenseSimilarity('walk','dog')

getSenseSimilarity('cricket','ball')
```

Consider the following code of a Lesk algorithm in NLTK , which is used to perform the disambiguation task:

```
from nltk.corpus import wordnet


def lesk(context_sentence, ambiguous_word, pos=None, synsets=None):
    """Return a synset for an ambiguous word in a context.

    :param iter context_sentence: The context sentence where the
ambiguous word
    occurs, passed as an iterable of words.
    :param str ambiguous_word: The ambiguous word that requires WSD.
    :param str pos: A specified Part-of-Speech (POS).
```

```
    :param iter synsets: Possible synsets of the ambiguous word.
    :return: ``lesk_sense`` The Synset() object with the highest
signature overlaps.

//    This function is an implementation of the original Lesk
algorithm (1986) [1].

    Usage example::

>>> lesk(['I', 'went', 'to', 'the', 'bank', 'to', 'deposit', 'money',
'.'], 'bank', 'n')
        Synset('savings_bank.n.02')


    context = set(context_sentence)
    if synsets is None:
        synsets = wordnet.synsets(ambiguous_word)

    if pos:
        synsets = [ss for ss in synsets if str(ss.pos()) == pos]

    if not synsets:
        return None

    _, sense = max(
        (len(context.intersection(ss.definition().split())), ss) for
ss in synsets
    )

    return sense
```

Summary

In this chapter, we have discussed Semantic Analysis, which is also one of the phase of Natural Language Processing. We have discussed NER, NER using HMM, NER using Machine Learning Toolkits, Performance Metrics of NER, NER using POS tagging, and WSD using Wordnet and the Generation of Synsets.

In the next chapter, we will discuss sentiment analysis using NER and machine learning approaches. We will also discuss the evaluation of the NER system.

7
Sentiment Analysis – I Am Happy

Sentiment analysis or sentiment generation is one of the tasks in NLP. It is defined as the process of determining the sentiments behind a character sequence. It may be used to determine whether the speaker or the person expressing the textual thoughts is in a happy or sad mood, or it represents a neutral expression.

This chapter will include the following topics:

- Introducing sentiment analysis
- Sentiment analysis using NER
- Sentiment analysis using machine learning
- Evaluation of the NER system

Introducing sentiment analysis

Sentiment analysis may be defined as a task performed on natural languages. Here, computations are performed on the sentences or words expressed in natural language to determine whether they express a positive, negative, or a neutral sentiment. Sentiment analysis is a subjective task, since it provides the information about the text being expressed. Sentiment analysis may be defined as a classification problem in which classification may be of two types—binary categorization (positive or negative) and multi-class categorization (positive, negative, or neutral). Sentiment analysis is also referred to as **text sentiment analysis**. It is a text mining approach in which we determine the sentiments or the emotions behind the text. When we combine sentiment analysis with topic mining, then it is referred to as **topic-sentiment analysis**. Sentiment analysis can be performed using a lexicon. The lexicon could be domain-specific or of a general purpose nature. Lexicon may contain a list of positive expressions, negative expressions, neutral expressions, and stop words. When a testing sentence appears, then a simple look up operation can be performed through this lexicon.

One example of the word list is—**Affective Norms for English Words** (**ANEW)**. It is an English word list found at the University of Florida. It consists of 1034 words for dominance, valence, and arousal. It was formed by Bradley and Lang. This word list was constructed for academic purposes and not for research purposes. Other variants are **DANEW** (Dutch ANEW) and **SPANEW** (Spanish ANEW).

AFINN consists of 2477 words (earlier 1468 words). This word list was formed by Finn Arup Nielson. The main purpose for creating this word list was to perform sentiment analysis for Twitter texts. A valence value ranging from *-5* to *+5* is allotted to each word.

The **Balance Affective** word list consists of 277 English words. The valence code ranges from 1 to 4. 1 means positive, 2 means negative, 3 means anxious, and 4 means neutral.

Berlin Affective Word List (**BAWL)** consists of 2,200 words in German. Another version of BAWL is **Berlin Affective Word List Reloaded** (**BAWL-R**) that comprises of additional arousal for words.

Bilingual Finnish Affective Norms comprises 210 British English as well as Finnish nouns. It also comprises taboo words.

Compass DeRose Guide to Emotion Words consists of emotional words in English. This was formed by Steve J. DeRose. Words were classified, but there was no valence and arousal.

Dictionary of Affect in Language (**DAL**) comprises emotional words that can be used for sentiment analysis. It was formed by Cynthia M. Whissell. So, it is also referred to as **Whissell's Dictionary of Affect in Language** (**WDAL**).

General Inquirer consists of many dictionaries. In this, the positive list comprises 1915 words and the negative list comprises 2291 words.

Hu-Liu opinion Lexicon (**HL**) comprises a list of 6800 words (positive and negative).

Leipzig Affective Norms for German (**LANG**) is a list that consists of 1000 nouns in German, and the rating has been done based on valence, concreteness, and arousal.

Loughran and McDonald Financial Sentiment Dictionaries were created by Tim Loughran and Bill McDonald. These dictionaries consist of words for financial documents, which are positive, negative, or modal words.

Moors consist of a list of words in Dutch related to dominance, arousal, and valence.

NRC Emotion Lexicon comprises of a list of words developed through Amazon Mechanical Turk by Saif M. Mohammad.

OpinionFinder's **Subjectivity Lexicon** comprises a list of 8221 words (positive or negative).

SentiSense comprises 2,190 synsets and 5,496 words based on 14 emotional categories.

Warringer comprises 13,915 words in English collected from Amazon Mechanical Turk that are related to dominance, arousal, and valence.

labMT is a word list consisting of 10,000 words.

Let's consider the following example in NLTK, which performs sentiment analysis for movie reviews:

```
import nltk
import random
from nltk.corpus import movie_reviews
docs = [(list(movie_reviews.words(fid)), cat)
        for cat in movie_reviews.categories()
        for fid in movie_reviews.fileids(cat)]
random.shuffle(docs)

all_tokens = nltk.FreqDist(x.lower() for x in movie_reviews.words())
token_features = all_tokens.keys()[:2000]
print token_features[:100]
```

```
    [',', 'the', '.', 'a', 'and', 'of', 'to', "'", 'is', 'in', 's',
'"', 'it', 'that', '-', ')', '(', 'as', 'with', 'for', 'his', 'this',
'film', 'i', 'he', 'but', 'on', 'are', 't', 'by', 'be', 'one',
'movie', 'an', 'who', 'not', 'you', 'from', 'at', 'was', 'have',
'they', 'has', 'her', 'all', '?', 'there', 'like', 'so', 'out',
'about', 'up', 'more', 'what', 'when', 'which', 'or', 'she', 'their',
':', 'some', 'just', 'can', 'if', 'we', 'him', 'into', 'even', 'only',
'than', 'no', 'good', 'time', 'most', 'its', 'will', 'story', 'would',
'been', 'much', 'character', 'also', 'get', 'other', 'do', 'two',
'well', 'them', 'very', 'characters', ';', 'first', '--', 'after',
'see', '!', 'way', 'because', 'make', 'life']

def doc_features(doc):
    doc_words = set(doc)
    features = {}
    for word in token_features:
        features['contains(%s)' % word] = (word in doc_words)
    return features

print doc_features(movie_reviews.words('pos/cv957_8737.txt
feature_sets = [(doc_features(d), c) for (d,c) in doc]
train_sets, test_sets = feature_sets[100:], feature_sets[:100]
classifiers = nltk.NaiveBayesClassifier.train(train_sets)
print nltk.classify.accuracy(classifiers, test_sets)

    0.86

classifier.show_most_informative_features(5)

     Most Informative Features
contains(damon) = True              pos : neg    =     11.2 : 1.0
contains(outstanding) = True        pos : neg    =     10.6 : 1.0
contains(mulan) = True              pos : neg    =      8.8 : 1.0
contains(seagal) = True             neg : pos    =      8.4 : 1.0
contains(wonderfully) = True        pos : neg    =      7.4 : 1.0
```

Here, it is checked whether the informative features are present in the document or not.

Consider another example of semantic analysis. First, the preprocessing of text is performed. In this, individual sentences are identified in a given text. Then, tokens are identified in the sentences. Each token further comprises three entities, namely, word, lemma, and tag.

Let's see the following code in NLTK for the preprocessing of text:

```
importnltk

class Splitter(object):
def __init__(self):
self.nltk_splitter = nltk.data.load('tokenizers/punkt/english.pickle')
self.nltk_tokenizer = nltk.tokenize.TreebankWordTokenizer()

def split(self, text):
sentences = self.nltk_splitter.tokenize(text)
tokenized_sentences = [self.nltk_tokenizer.tokenize(sent) for sent in
sentences]
return tokenized_sentences
classPOSTagger(object):
def __init__(self):
pass

def pos_tag(self, sentences):

pos = [nltk.pos_tag(sentence) for sentence in sentences]
pos = [[(word, word, [postag]) for (word, postag) in sentence] for
sentence in pos]
returnpos
```

The lemmas generated will be same as the word forms. Tags are the POS tags. Consider the following code, which generates three tuples for each token, that is, word, lemma, and the POS tag:

```
text = """Why are you looking disappointed. We will go to restaurant
for dinner."""
splitter = Splitter()
postagger = POSTagger()
splitted_sentences = splitter.split(text)
print splitted_sentences
[['Why','are','you','looking','disappointed','.'], ['We','will','go','
to','restaurant','for','dinner','.']]

pos_tagged_sentences = postagger.pos_tag(splitted_sentences)

print pos_tagged_sentences
[[('Why','Why',['WP']),('are','are',['VBZ']),('you','you',['PRP']
),('looking','looking',['VB']),('disappointed','disappointed',['
VB']),('.','.',['.'])],[('We','We',['PRP']),('will','will',['VBZ']),('
go','go',['VB']),('to','to',['TO']),('restaurant','restaurant',['NN'])
,('for','for',['IN']),('dinner','dinner',['NN']),('.','.',['.'])]]
```

We can construct two kinds of dictionary consisting of positive and negative expressions. We can then perform tagging on our processed text using dictionaries.

Let's consider the following NLTK code for tagging using dictionaries:

```
classDictionaryTagger(object):
def __init__(self, dictionary_paths):
files = [open(path, 'r') for path in dictionary_paths]
dictionaries = [yaml.load(dict_file) for dict_file in files]
map(lambda x: x.close(), files)
self.dictionary = {}
self.max_key_size = 0
forcurr_dict in dictionaries:
for key in curr_dict:
if key in self.dictionary:
self.dictionary[key].extend(curr_dict[key])
else:
self.dictionary[key] = curr_dict[key]
self.max_key_size = max(self.max_key_size, len(key))

def tag(self, postagged_sentences):
return [self.tag_sentence(sentence) for sentence in postagged_
sentences]

def tag_sentence(self, sentence, tag_with_lemmas=False):
tag_sentence = []
        N = len(sentence)
ifself.max_key_size == 0:
self.max_key_size = N
i = 0
while (i< N):
j = min(i + self.max_key_size, N) #avoid overflow
tagged = False
while (j >i):
expression_form = ' '.join([word[0] for word in sentence[i:j]]).
lower()
expression_lemma = ' '.join([word[1] for word in sentence[i:j]]).
lower()
iftag_with_lemmas:
literal = expression_lemma
else:
literal = expression_form
if literal in self.dictionary:
    is_single_token = j - i == 1
original_position = i
i = j
taggings = [tag for tag in self.dictionary[literal]]
```

```
tagged_expression = (expression_form, expression_lemma, taggings)
ifis_single_token: #if the tagged literal is a single token, conserve
its previous taggings:
original_token_tagging = sentence[original_position][2]
tagged_expression[2].extend(original_token_tagging)
tag_sentence.append(tagged_expression)
tagged = True
else:
                    j = j - 1
if not tagged:
tag_sentence.append(sentence[i])
i += 1
return tag_sentence
```

Here, words in the preprocessed text are tagged as positive or negative with the help of dictionaries.

Let's see the following code in NLTK, which can be used to compute the number of positive expressions and negative expressions:

```
def value_of(sentiment):
if sentiment == 'positive': return 1
if sentiment == 'negative': return -1
return 0
def sentiment_score(review):
return sum ([value_of(tag) for sentence in dict_tagged_sentences for
token in sentence for tag in token[2]])
```

The `nltk.sentiment.util` module is used in NLTK to perform sentiment analysis using Hu-Liu lexicon. This module counts the number of positive, negative, and neutral expressions, with the help of the lexicon, and then decides on the basis of majority counts whether the text consist of a positive, negative, or neutral sentiment. The words which are not available in the lexicon are considered neutral.

Sentiment analysis using NER

NER is the process of finding named entities and then categorizing named entities into different named entity classes. NER can be performed using different techniques, such as the Rule-based approach, List look up approach, and Statistical approaches (Hidden Markov Model, Maximum Entropy Markov Model, Support Vector Machine, Conditional Random Fields, and Decision Trees).

If named entities are identified in the list, then they may be removed or filtered out from the sentences. Similarly, stop words may also be removed. Now, sentiment analysis may be performed on the remaining words, since named entities are words that do not contribute to sentiment analysis.

Sentiment analysis using machine learning

The `nltk.sentiment.sentiment_analyzer` module in NLTK is used to perform sentiment analysis. It is based on machine learning techniques.

Let's see the following code of the `nltk.sentiment.sentiment_analyzer` module in NLTK:

```
from __future__ import print_function
from collections import defaultdict

from nltk.classify.util import apply_features, accuracy as eval_
accuracy
from nltk.collocations import BigramCollocationFinder
from nltk.metrics import (BigramAssocMeasures, precision as eval_
precision,
    recall as eval_recall, f_measure as eval_f_measure)

from nltk.probability import FreqDist

from nltk.sentiment.util import save_file, timer
class SentimentAnalyzer(object):
    """
    A tool for Sentiment Analysis which is based on machine learning
techniques.
    """
    def __init__(self, classifier=None):
        self.feat_extractors = defaultdict(list)
        self.classifier = classifier
```

Consider the following code, which will return all the words (duplicates) from a text:

```
    def all_words(self, documents, labeled=None):
        all_words = []
        if labeled is None:
            labeled = documents and isinstance(documents[0], tuple)
        if labeled == True:
            for words, sentiment in documents:
                all_words.extend(words)
        elif labeled == False:
```

```
            for words in documents:
                all_words.extend(words)
        return all_words
```

Consider the following code, which will apply the feature extraction function to the text:

```
def apply_features(self, documents, labeled=None):

        return apply_features(self.extract_features, documents,
labeled)
```

Consider the following code, which will return the word's features:

```
def unigram_word_feats(self, words, top_n=None, min_freq=0):
        unigram_feats_freqs = FreqDist(word for word in words)
        return [w for w, f in unigram_feats_freqs.most_common(top_n)
                if unigram_feats_freqs[w] > min_freq]
```

The following code returns the bigram features:

```
def bigram_collocation_feats(self, documents, top_n=None, min_freq=3,
                                 assoc_measure=BigramAssocMeasures.
pmi):
        finder = BigramCollocationFinder.from_documents(documents)
        finder.apply_freq_filter(min_freq)
        return finder.nbest(assoc_measure, top_n)
```

Let's see the following code, which can be used to classify a given instance using the available feature set:

```
def classify(self, instance):
        instance_feats = self.apply_features([instance],
labeled=False)
        return self.classifier.classify(instance_feats[0])
```

Let's see the following code, which can be used for the extraction of features from the text:

```
def add_feat_extractor(self, function, **kwargs):
        self.feat_extractors[function].append(kwargs)

def extract_features(self, document):
        all_features = {}
        for extractor in self.feat_extractors:
            for param_set in self.feat_extractors[extractor]:
                feats = extractor(document, **param_set)
            all_features.update(feats)
        return all_features
```

Let's see the following code that can be used to perform training on the training file. `Save_classifier` is used to save the output in a file:

```
def train(self, trainer, training_set, save_classifier=None,
**kwargs):
        print("Training classifier")
        self.classifier = trainer(training_set, **kwargs)
        if save_classifier:
            save_file(self.classifier, save_classifier)

        return self.classifier
```

Let's see the following code that can be used to perform testing and performance evaluation of our classifier using test data:

```
def evaluate(self, test_set, classifier=None, accuracy=True, f_
measure=True,
                 precision=True, recall=True, verbose=False):
        if classifier is None:
            classifier = self.classifier
        print("Evaluating {0} results...".format(type(classifier).__
name__))
        metrics_results = {}
        if accuracy == True:
            accuracy_score = eval_accuracy(classifier, test_set)
            metrics_results['Accuracy'] = accuracy_score

        gold_results = defaultdict(set)
        test_results = defaultdict(set)
        labels = set()
        for i, (feats, label) in enumerate(test_set):
            labels.add(label)
            gold_results[label].add(i)
            observed = classifier.classify(feats)
            test_results[observed].add(i)

        for label in labels:
            if precision == True:
                precision_score = eval_precision(gold_results[label],
                    test_results[label])
                metrics_results['Precision [{0}]'.format(label)] =
precision_score
            if recall == True:
                recall_score = eval_recall(gold_results[label],
                    test_results[label])
```

```
                metrics_results['Recall [{0}]'.format(label)] =
recall_score
            if f_measure == True:
                f_measure_score = eval_f_measure(gold_results[label],
                    test_results[label])
                metrics_results['F-measure [{0}]'.format(label)] = f_
measure_score

        if verbose == True:
            for result in sorted(metrics_results):
                print('{0}: {1}'.format(result, metrics_
results[result]))

        return metrics_results
```

Twitter can be considered as one of the most popular blogging services that is used to create messages referred to as *tweets*. These tweets comprise words that are either related to positive, negative, or neutral sentiments.

For performing sentiment analysis, we can use machine learning classifiers, statistical classifiers, or automated classifiers, such as the Naive Bayes Classifier, Maximum Entropy Classifier, Support Vector Machine Classifier, and so on.

These machine learning classifiers or automated classifiers are used to perform supervised classification, since they require training data for classification.

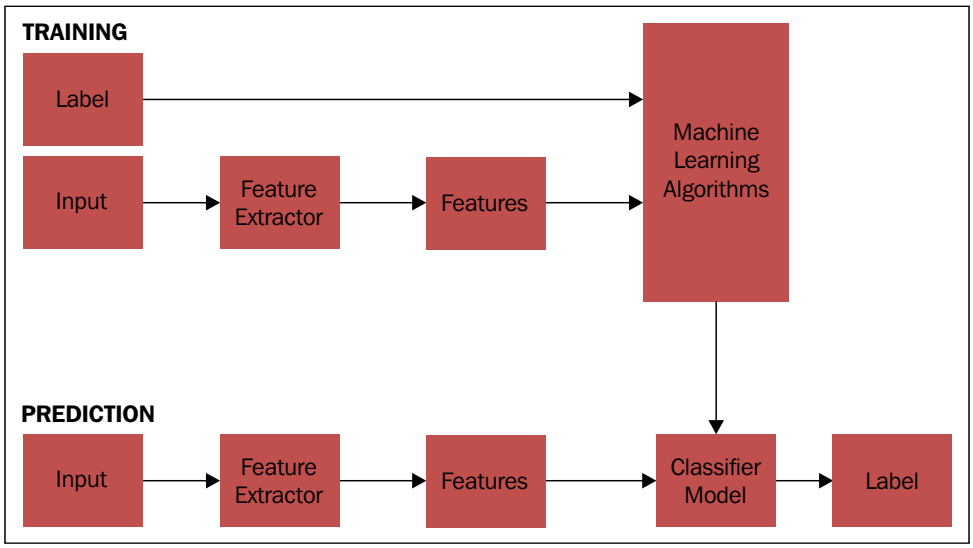
Let's see the following code in NLTK for feature extraction:

```
stopWords = []

#If there is occurrence of two or more same character, then replace it
with the character itself.
def replaceTwoOrMore(s):
    pattern = re.compile(r"(.)\1{1,}", re.DOTALL)
    return pattern.sub(r"\1\1", s)
def getStopWordList(stopWordListFileName):
    # This function will read the stopwords from a file and builds a
list.
    stopWords = []
    stopWords.append('AT_USER')
    stopWords.append('URL')

    fp = open(stopWordListFileName, 'r')
    line = fp.readline()
    while line:
```

```
            word = line.strip()
            stopWords.append(word)
            line = fp.readline()
        fp.close()
        return stopWords

    def getFeatureVector(tweet):
        featureVector = []
        #Tweets are  firstly split into words
        words = tweet.split()
        for w in words:
            #replace two or more with two occurrences
            w = replaceTwoOrMore(w)
            #strip punctuation
            w = w.strip('\'"?,.')
            #Words begin with alphabet is checked.
            val = re.search(r"^[a-zA-Z][a-zA-Z0-9]*$", w)
            #If there is a stop word, then it is ignored.
            if(w in stopWords or val is None):
                continue
            else:
                featureVector.append(w.lower())
        return featureVector
    #end

    #Tweets are read one by one and then processed.
    fp = open('data/sampleTweets.txt', 'r')
    line = fp.readline()

    st = open('data/feature_list/stopwords.txt', 'r')
    stopWords = getStopWordList('data/feature_list/stopwords.txt')

    while line:
        processedTweet = processTweet(line)
        featureVector = getFeatureVector(processedTweet)
        print featureVector
        line = fp.readline()
    #end loop
    fp.close()

    #Tweets are read one by one and then processed.
    inpTweets = csv.reader(open('data/sampleTweets.csv', 'rb'),
    delimiter=',', quotechar='|')
    tweets = []
```

```
for row in inpTweets:
    sentiment = row[0]
    tweet = row[1]
    processedTweet = processTweet(tweet)
    featureVector = getFeatureVector(processedTweet, stopWords)
    tweets.append((featureVector, sentiment));

#Features Extraction takes place using following method
def extract_features(tweet):
    tweet_words = set(tweet)
    features = {}
    for word in featureList:
        features['contains(%s)' % word] = (word in tweet_words)
    return features
```

During the training of a classifier, the input to the machine learning algorithm is a label and features. Features are obtained from the feature extractor when the input is given to the feature extractor. During prediction, a label is provided as an output of a classifier model and the input of the classifier model is the features that are obtained using the feature extractor. Let's have a look at a diagram explaining the same process:



Now, let's see the following code that can be used to perform sentiment analysis using the Naive Bayes Classifier:

```
NaiveBClassifier = nltk.NaiveBayesClassifier.train(training_set)
# Testing the classifiertestTweet = 'I liked this book on Sentiment
Analysis a lot.'
processedTestTweet = processTweet(testTweet)
print NaiveBClassifier.classify(extract_features(getFeatureVector(proc
essedTestTweet)))
testTweet = 'I am so badly hurt'
processedTestTweet = processTweet(testTweet)
print NBClassifier.classify(extract_features(getFeatureVector(process
edTestTweet)))
```

Let's see the following code on sentiment analysis using maximum entropy:

```
MaxEntClassifier = nltk.classify.maxent.MaxentClassifier.
train(training_set, 'GIS', trace=3, \
                    encoding=None, labels=None, sparse=True, gaussian_
prior_sigma=0, max_iter = 10)
testTweet = 'I liked the book on sentiment analysis a lot'
processedTestTweet = processTweet(testTweet)
print MaxEntClassifier.classify(extract_features(getFeatureVector(proc
essedTestTweet)))
print MaxEntClassifier.show_most_informative_features(10)
```

Evaluation of the NER system

Performance metrics or evaluation helps to show the performance of an NER system. The outcome of an NER tagger may be defined as the *response* and the interpretation of human beings as the *answer key*. So, we will provide the following definitions:

- **Correct**: If the response is exactly the same as the answer key
- **Incorrect**: If the response is not the same as the answer key
- **Missing**: If the answer key is found tagged, but the response is not tagged
- **Spurious**: If the response is found tagged, but the answer key is not tagged

The performance of an NER-based system can be judged by using the following parameters:

- **Precision (P)**: *P=Correct/(Correct+Incorrect+Missing)*
- **Recall (R)**: *R=Correct/(Correct+Incorrect+Spurious)*
- **F-Measure**: *F-Measure = (2*P*R)/(P+R)*

Let's see the code for NER using the HMM:

```
#*******   Function to find all tags in corpus  **********

def find_tag_set(tra_lines):
global tag_set

tag_set = [ ]

for line in tra_lines:
tok = line.split()
for t in tok:
wd = t.split("/")
if not wd[1] in tag_set:
tag_set.append(wd[1])

return

#*******   Function to find frequency of each tag in tagged corpus
**********

defcnt_tag(tr_ln):
global start_li
global li
global tag_set
global c
global line_cnt
global lines

lines = tr_ln

start_li = [ ]   # list of starting tags

find_tag_set(tr_ln)


line_cnt  = 0
for line in lines:
tok = line.split()
x = tok[0].split("/")
if  not x[1] in start_li:
start_li.append(x[1])
line_cnt = line_cnt + 1
```

```
find_freq_tag()

find_freq_srttag()

return

def find_freq_tag():
global tag_cnt
global tag_set
tag_cnt={}
i = 0
for w in tag_set:
cal_freq_tag(tag_set[i])
i = i + 1
tag_cnt.update({w:freq_tg})
return


defcal_freq_tag(tg):
global freq_tg
global lines
freq_tg = 0

for line in lines:
freq_tg = freq_tg + line.count(tg)

return

#*******   Function to find frequency of each starting tag in tagged
corpus  **********


def find_freq_srttag():
global lst
lst = {}        # start probability

i  = 0
for w in start_li:
        cc = freq_srt_tag(start_li[i])
prob = cc / line_cnt

lst.update({start_li[i]:prob})
i = i + 1
return
```

```
def freq_srt_tag(stg):
global lines
freq_srt_tg = 0

for line in lines:
tok = line.split()
ifstg in tok[0]:
freq_srt_tg = freq_srt_tg + 1

return(freq_srt_tg)

import tkinter as tk
import vit
import random
import cal_start_p
import calle_prob
import trans_mat
import time
import trans
import dict5
from tkinter import *
from tkinter import ttk
from tkinter.filedialog import askopenfilename
from tkinter.messagebox import showerror
import languagedetect1
import languagedetect3
e_dict = dict()
t_dict = dict()

def calculate1(*args):
import listbox1
def calculate2(*args):
import listbox2
def calculate3(*args):
import listbox3


def dispdlg():
global file_name
root = tk.Tk()
root.withdraw()
file_name = askopenfilename()
return
```

```
def tranhmm():
ttk.Style().configure("TButton", padding=6, relief="flat",background="
Pink",foreground="Red")
ttk.Button(mainframe, text="BROWSE", command=find_train_corpus).
grid(column=7, row=5, sticky=W)

# The following code will be used to display or accept the testing
corpus from the user.
def testhmm():
ttk.Button(mainframe, text="Develop a new testing Corpus",
command=calculate3).grid(column=9, row=5, sticky=E)

ttk.Button(mainframe, text="BROWSE", command=find_obs).grid(column=9,
row=7, sticky=E)


#In HMM, We require parameters such as Start Probability, Transition
Probability and Emission Probability. The following code is used to
calculate emission probability matrix

def cal_emit_mat():
global emission_probability
global corpus
global tlines

calle_prob.m_prg(e_dict,corpus,tlines)

emission_probability = e_dict

return

# to calculate states


def cal_states():
global states
global tlines

cal_start_p.cnt_tag(tlines)

states = cal_start_p.tag_set

return
```

```
# to take observations


def find_obs():
global observations
global test_lines
global tra
global w4
global co
global tra
global wo1
global wo2
global testl
global wo3
global te
global definitionText
global definitionScroll
global dt2
global ds2
global dt11
global ds11

wo3=[ ]
woo=[ ]
wo1=[ ]
wo2=[ ]
    co=0
w4=[ ]
if(flag2!=0):
definitionText11.pack_forget()
definitionScroll11.pack_forget()
dt1.pack_forget()
ds1.pack_forget()
dispdlg()
f = open(file_name,"r+",encoding = 'utf-8')
test_lines = f.readlines()
f.close()
fname="C:/Python32/file_name1"




for x in states:
if not x in start_probability:
```

```
start_probability.update({x:0.0})
for line in test_lines:
ob = line.split()
observations = ( ob )



fe=open("C:\Python32\output3_file","w+",encoding = 'utf-8')
fe.write("")
fe.close()
ff=open("C:\Python32\output4_file","w+",encoding = 'utf-8')


ff.write("")
ff.close()
ff7=open("C:\Python32\output5_file","w+",encoding = 'utf-8')
ff7.write("")
ff7.close()
ff8=open("C:\Python32\output6_file","w+",encoding = 'utf-8')
ff8.write("")
ff8.close()
ff81=open("C:\Python32\output7_file","w+",encoding = 'utf-8')
ff81.write("")
ff81.close()
dict5.search_obs_train_corpus(file1,fname,tlines,test_
lines,observations, states, start_probability, transition_probability,
emission_probability)

f20 = open("C:\Python32\output5_file","r+",encoding = 'utf-8')
te = f20.readlines()
tee=f20.read()
f = open(fname,"r+",encoding = 'utf-8')
train_llines = f.readlines()



ds11 = Scrollbar(root)
dt11 = Text(root, width=10, height=20,fg='black',bg='pink',yscrollcom
mand=ds11.set)
ds11.config(command=dt11.yview)
dt11.insert("1.0",train_llines)
dt11.insert("1.0","\n")
dt11.insert("1.0","\n")
```

```
dt11.insert("1.0","******TRAINING SENTENCES******")


     # an example of how to add new text to the text area
dt11.pack(padx=10,pady=150)
ds11.pack(padx=10,pady=150)

ds11.pack(side=LEFT, fill=BOTH)
dt11.pack(side=LEFT, fill=BOTH, expand=True)

ds2 = Scrollbar(root)
dt2 = Text(root, width=10, height=10,fg='black',bg='pink',yscrollcomm
and=ds2.set)
ds2.config(command=dt2.yview)
dt2.insert("1.0",test_lines)
dt2.insert("1.0","\n")
dt2.insert("1.0","\n")
dt2.insert("1.0","*********TESTING SENTENCES*********")



     # an example of how to add new text to the text area
dt2.pack(padx=10,pady=150)
ds2.pack(padx=10,pady=150)

ds2.pack(side=LEFT, fill=BOTH)
dt2.pack(side=LEFT, fill=BOTH, expand=True)


definitionScroll = Scrollbar(root)
definitionText = Text(root, width=10, height=10,fg='black',bg='pink',y
scrollcommand=definitionScroll.set)
definitionScroll.config(command=definitionText.yview)
definitionText.insert("1.0",te)
definitionText.insert("1.0","\n")
definitionText.insert("1.0","\n")
definitionText.insert("1.0","*********OUTPUT*********")



     # an example of how to add new text to the text area
definitionText.pack(padx=10,pady=150)
definitionScroll.pack(padx=10,pady=150)
```

```
definitionScroll.pack(side=LEFT, fill=BOTH)
definitionText.pack(side=LEFT, fill=BOTH, expand=True)

l = tk.Label(root, text="NOTE:*****The Entities which are not tagged
in Output are not Named Entities*****" , fg='black', bg='pink')
l.place(x = 500, y = 650, width=500, height=25)



    #ttk.Button(mainframe, text="View Parameters", command=parame).
grid(column=11, row=10, sticky=E)
    #definitionText.place(x= 19, y = 200,height=25)

f20.close()

f14 = open("C:\Python32\output2_file","r+",encoding = 'utf-8')
testl = f14.readlines()
for lines in testl:
toke = lines.split()
for t in toke:
w4.append(t)
f14.close()
f12 = open("C:\Python32\output_file","w+",encoding = 'utf-8')
f12.write("")
f12.close()


ttk.Button(mainframe, text="SAVE OUTPUT", command=save_output).
grid(column=11, row=7, sticky=E)
ttk.Button(mainframe, text="NER EVALUATION", command=evaluate).
grid(column=13, row=7, sticky=E)
ttk.Button(mainframe, text="REFRESH", command=ref).grid(column=15,
row=7, sticky=E)

return
def ref():
root.destroy()
import new1
return
```

Let's see the following code in Python, which will be used to evaluate the output produced by NER using HMM:

```
def evaluate():
global wDict
```

```
global woe
global woe1
global woe2
woe1=[ ]
woe=[ ]
woe2=[ ]
ws=[ ]
wDict = {}
i=0
    j=0
    k=0
sp=0
f141 = open("C:\Python32\output1_file","r+",encoding = 'utf-8')
tesl = f141.readlines()
for lines in tesl:
toke = lines.split()
for t in toke:
ws.append(t)
if t in wDict: wDict[t] += 1
else: wDict[t] = 1
for line in tlines:
tok = line.split()

for t in tok:
wd = t.split("/")
if(wd[1]!='OTHER'):
if t in wDict: wDict[t] += 1
else: wDict[t] = 1
print ("words  in train corpus ",wDict)
for key  in wDict:
i=i+1
print("total words in Dictionary are:",i)
for line in train_lines:
toe=line.split()
for t1 in toe:
if '/' not in t1:
sp=sp+1
woe2.append(t1)
print("Spurious words are")
for w in woe2:
print(w)
print("Total spurious words are:",sp)
for l in te:
to=l.split()
```

```
for t1 in to:
if '/' in t1:
                #print(t1)
if t1 in ws or t1 in wDict:
woe.append(t1)
                    j=j+1
if t1 not in wDict:
wdd=t1.split("/")
ifwdd[0] not in woe2:
woe1.append(t1)
                        k=k+1
print("Word found in Dict are:")
for w in woe:
print(w)
print("Word not found in Dict are:")
for w in woe1:
print(w)
print("Total correctly tagged words are:",j)
print("Total incorrectly tagged words are:",k)
pr=(j)/(j+k)
re=(j)/(j+k+sp)
f141.close()
root=Tk()
root.title("NER EVALUATION")
root.geometry("1000x1000")

ds21 = Scrollbar(root)
dt21 = Text(root, width=10, height=10,fg='black',bg='pink',yscrollcom
mand=ds21.set)
ds21.config(command=dt21.yview)
dt21.insert("1.0",(2*pr*re)/(pr+re))
dt21.insert("1.0","\n")
dt21.insert("1.0","F-MEASURE=")
dt21.insert("1.0","\n")
dt21.insert("1.0","F-MEASURE=(2*PRECISION*RECALL)/(PRECISION+RECALL)")
dt21.insert("1.0","\n")
dt21.insert("1.0","\n")
dt21.insert("1.0",re)
dt21.insert("1.0","RECALL=")
dt21.insert("1.0","\n")
dt21.insert("1.0","RECALL= CORRECT/(CORRECT +INCORRECT +SPURIOUS)")
dt21.insert("1.0","\n")
dt21.insert("1.0","\n")
dt21.insert("1.0",pr)
```

```
dt21.insert("1.0","PRECISION=")
dt21.insert("1.0","\n")
dt21.insert("1.0","PRECISION= CORRECT/(CORRECT +INCORRECT +MISSING)")
dt21.insert("1.0","\n")
dt21.insert("1.0","\n")
dt21.insert("1.0","Total No. of Missing words are: 0")
dt21.insert("1.0","\n")
dt21.insert("1.0","\n")
dt21.insert("1.0",sp)
dt21.insert("1.0","Total No. of Spurious Words are:")
dt21.insert("1.0","\n")
for w in woe2:
dt21.insert("1.0",w)
dt21.insert("1.0"," ")
dt21.insert("1.0","Total Spurious Words are:")
dt21.insert("1.0","\n")
dt21.insert("1.0","\n")
dt21.insert("1.0",k)
dt21.insert("1.0","Total No. of Incorrectly tagged words are:")
dt21.insert("1.0","\n")
for w in woe1:
dt21.insert("1.0",w)
dt21.insert("1.0"," ")
dt21.insert("1.0","Total Incorrectly tagged words are:")
dt21.insert("1.0","\n")
dt21.insert("1.0","\n")
dt21.insert("1.0",j)
dt21.insert("1.0","Total No. of Correctly tagged words are:")
dt21.insert("1.0","\n")
for w in woe:
dt21.insert("1.0",w)
dt21.insert("1.0"," ")
dt21.insert("1.0","Total Correctly tagged words are:")
dt21.insert("1.0","\n")
dt21.insert("1.0","\n")
dt21.insert("1.0","***************PERFORMANCE EVALUATION OF
NERHMM***************")



     # an example of how to add new text to the text area
dt21.pack(padx=5,pady=5)
ds21.pack(padx=5,pady=5)
```

```
ds21.pack(side=LEFT, fill=BOTH)
dt21.pack(side=LEFT, fill=BOTH, expand=True)
root.mainloop()
return
def save_output():
    #dispdlg()
f = open("C:\Python32\save","w+",encoding = 'utf-8')
f20 = open("C:\Python32\output5_file","r+",encoding = 'utf-8')
te = f20.readlines()
for t in te:
f.write(t)
f.close()
f20.close()

# to calculate start probability matrix

def cal_srt_prob():
global start_probability

start_probability = cal_start_p.lst

return

# to print vitarbi parameter if required

def pr_param():
l1 = tk.Label(root, text="HMM Training is going on.....Don't Click any
Button!!",fg='black',bg='pink')
l1.place(x = 300, y = 150,height=25)

print("states")
print(states)
print(" ")
print(" ")
print("start probability")
print(start_probability)
print(" ")
print(" ")
print("transition probability")
print(transition_probability)
print(" ")
print(" ")
print("emission probability")
print(emission_probability)
```

```
l1 = tk.Label(root, text="
")
l1.place(x = 300, y = 150,height=25)
global flag1
    flag1=0
global flag2
    flag2=0
ttk.Button(mainframe, text="View Parameters", command=parame).
grid(column=7, row=5, sticky=W)
return

def parame():
global flag2
    flag2=flag1+1
global definitionText11
global definitionScroll11
definitionScroll11 = Scrollbar(root)
definitionText11 = Text(root, width=10, height=10,fg='black',bg='pink'
,yscrollcommand=definitionScroll11.set)

    #definitionText.place(x= 19, y = 200,height=25)
definitionScroll11.config(command=definitionText11.yview)

definitionText11.delete("1.0", END)   # an example of how to delete
all current text
definitionText11.insert("1.0",emission_probability )
definitionText11.insert("1.0","\n")
definitionText11.insert("1.0","Emission Probability")
definitionText11.insert("1.0","\n")
definitionText11.insert("1.0",transition_probability)
definitionText11.insert("1.0","Transition Probability")
definitionText11.insert("1.0","\n")
definitionText11.insert("1.0",start_probability)
definitionText11.insert("1.0","Start Probability")


     # an example of how to add new text to the text area
definitionText11.pack(padx=10,pady=175)
definitionScroll11.pack(padx=10,pady=175)

definitionScroll11.pack(side=LEFT, fill=BOTH)
definitionText11.pack(side=LEFT, fill=BOTH, expand=True)

return
```

```
# to calculate transition probability matrix


def cat_trans_prob():
global transition_probability
global corpus
global tlines

trans_mat.main_prg(t_dict,corpus,tlines)

transition_probability = t_dict
return

def find_train_corpus():
global train_lines
global tlines
global c
global corpus
global words1
global w1
global train1
global fname
global file1
global ds1
global dt1
global w21
words1=[ ]
    c=0
w1=[ ]
w21=[ ]
f11 = open("C:\Python32\output1_file","w+",encoding='utf-8')
f11.write("")
f11.close()
fr = open("C:\Python32\output_file","w+",encoding='utf-8')
fr.write("")
fr.close()
fgl=open("C:\Python32\ladetect1","w+",encoding = 'utf-8')
fgl.write("")
fgl.close()

fgl=open("C:\Python32\ladetect","w+",encoding = 'utf-8')
fgl.write("")
fgl.close()
dispdlg()
```

```
f = open(file_name,"r+",encoding = 'utf-8')
train_lines = f.readlines()

ds1 = Scrollbar(root)
dt1 = Text(root, width=10, height=10,fg='black',bg='pink',yscrollcomm
and=ds1.set)
ds1.config(command=dt1.yview)
dt1.insert("1.0",train_lines)
dt1.insert("1.0","\n")
dt1.insert("1.0","\n")
dt1.insert("1.0","*********TRAINING SENTENCES*********")


     # an example of how to add new text to the text area
dt1.pack(padx=10,pady=175)
ds1.pack(padx=10,pady=175)

ds1.pack(side=LEFT, fill=BOTH)
dt1.pack(side=LEFT, fill=BOTH, expand=True)
fname="C:/Python32/file_name1"
f = open(file_name,"r+",encoding = 'utf-8')
    file1=file_name
p = open(fname,"w+",encoding = 'utf-8')

corpus = f.read()
for line in train_lines:
tok = line.split()
for t in tok:
n=t.split()

le=len(t)
i=0
            j=0
for n1 in n:
while(j<le):

if(n1[j]!='/'):
i=i+1
                        j=j+1
else:
                        j=j+1
if(i==le):
p.write(t)
p.write("/OTHER ")          #Handling Spurious words
```

```
else:
p.write(t)
p.write(" ")

p.write("\n")



p.close()
fname="C:/Python32/file_name1"
f00 = open(fname,"r+",encoding = 'utf-8')
tlines = f00.readlines()
for line in tlines:
tok = line.split()
for t in tok:
wd = t.split("/")
if(wd[1]!='OTHER'):
if not wd[0] in words1:
words1.append(wd[0])
w1.append(wd[1])
f00.close()

f157 = open("C:\Python32\input_file","w+",encoding='utf-8')
f157.write("")
f157.close()
f1 = open("C:\Python32\input_file","w+",encoding='utf-8')   #input_
file has list of Named Entities of training file
for w in words1:
f1.write(w)
f1.write("\n")
f1.close()
fr=open("C:\Python32\detect","w+",encoding = 'utf-8')
fr.write("")
fr.close()


f.close()
f.close()


cal_states()
cal_emit_mat()
cal_srt_prob()
cat_trans_prob()
```

```
pr_param()


return

root=Tk()
root.title("NAMED ENTITY RECOGNITION IN NATURAL LANGUAGES USING HIDDEN
MARKOV MODEL")
root.geometry("1000x1000")

mainframe = ttk.Frame(root, padding="20 20 12 12")
mainframe.grid(column=0, row=0, sticky=(N, W, E, S))

b=StringVar()
a=StringVar()

ttk.Style().configure("TButton", padding=6, relief="flat",background="
Pink", foreground="Red")
ttk.Button(mainframe, text="ANNOTATION", command=calculate1).
grid(column=5, row=3, sticky=W)

ttk.Button(mainframe, text="TRAIN HMM", command=tranhmm).
grid(column=7, row=3, sticky=E)

ttk.Button(mainframe, text="TEST HMM", command=testhmm).grid(column=9,
row=3, sticky=E)

ttk.Button(mainframe, text="HELP", command=hmmhelp).grid(column=11,
row=3, sticky=E)


# To call viterbi for particular observations find in find_obs

def call_vitar():
global test_lines
global train_lines
global corpus
global observations
global states
global start_probability
global transition_probability
global emission_probability

find_train_corpus()
```

```
cal_states()
find_obs()
cal_emit_mat()
cal_srt_prob()
cat_trans_prob()

   # print("Vitarbi Parameters are for selected corpus")
   # pr_param()

    # -----------------To add all states not in start probability ---
-------------

for x in states:
if not x in start_probability:
start_probability.update({x:0.0})


for line in test_lines:

ob = line.split()
observations = ( ob )
print(" ")
print(" ")
print(line)
print("**************************")
print(vit.viterbi(observations, states, start_probability, transition_
probability, emission_probability),bg='Pink',fg='Red')
return



root.mainloop()
```

The preceding code in Python shows how NER is performed using the HMM, and how an NER system is evaluated using performance metrics (Precision, Recall and F-Measure).

Summary

In this chapter, we have discussed sentiment analysis using NER and machine learning techniques. We have also discussed the evaluation of an NER-based system..

In the next chapter, we'll discuss information retrieval, text summarization, stop word removal, question-answering system, and more.

8
Information Retrieval – Accessing Information

Information retrieval is one of the many applications of natural language processing. Information retrieval may be defined as the process of retrieving information (for example, the number of times the word *Ganga* has appeared in the document) corresponding to a query that has been made by the user.

This chapter will include the following topics:

- Introducing information retrieval
- Stop word removal
- Information retrieval using a vector space model
- Vector space scoring and query operator interactions
- Developing an IR system using latent semantic indexing
- Text summarization
- Question-answering system

Introducing information retrieval

Information retrieval may be defined as the process of retrieving the most suitable information as a response to the query being made by the user. In information retrieval, the search is performed based on metadata or context-based indexing. One example of information retrieval is Google Search in which, corresponding to each user query, a response is provided on the basis of the information retrieval algorithm being used. An indexing mechanism is used by the information retrieval algorithm. The indexing mechanism used is known as an inverted index. An IR system builds an index postlist to perform the information retrieval task.

Boolean retrieval is an information retrieval task in which a Boolean operation is applied to the postlist in order to retrieve relevant information.

The accuracy of an information retrieval task is measured in terms of precision and recall.

Suppose that a given IR system returns *X* documents when a query is fired. But the actual or gold set of documents that needs to be returned is *Y*.

Recall may be defined as the fraction of gold documents that a system finds. It may be defined as the ratio of true positives to the combination of true positives and false negatives.

Recall (R) = (X ∩ Y) / Y

Precision may be defined as the fraction of documents that an IR system detects and are correct.

Precision (P) = (X ∩ Y) / X

F-Measure may be defined as the harmonic mean of precision and recall.

*F-Measure = 2 * (X ∩ Y) / (X + Y)*

Stop word removal

While performing information retrieval , it is important to detect the stop words in a document and eliminate them.

Let's see the following code that can be used to provide the collection of stop words that can be detected in the English text in NLTK:

```
>>> import nltk
>>> fromnltk.corpus import stopwords
>>> stopwords.words('english')
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you',
'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his',
'himself', 'she', 'her', 'hers', 'herself', 'it', 'its', 'itself',
'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which',
'who', 'whom', 'this', 'that', 'these', 'those', 'am', 'is', 'are',
'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having',
'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if',
'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for',
'with', 'about', 'against', 'between', 'into', 'through', 'during',
'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in',
'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then',
'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any',
```

```
'both', 'each', 'few', 'more', 'most', 'other', 'some', 'such', 'no',
'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's',
't', 'can', 'will', 'just', 'don', 'should', 'now']
```

NLTK consists of stop word corpus that comprises of 2,400 stop words from 11 different languages.

Let's see the following code in NLTK that can be used to find the fraction of words in a text that are not stop words:

```
>>> def not_stopwords(text):
    stopwords = nltk.corpus.stopwords.words('english')
    content = [w for w in text if w.lower() not in stopwords]
    return len(content) / len(text)

>>> not_stopwords(nltk.corpus.reuters.words())
0.7364374824583169
```

Let's see the following code in NLTK that can be used to remove the stop words from a given text. Here, the `lower()` function is used prior to the elimination of stop words so that the stop words in capital letters, such as A, are first converted into lower case letters and then eliminated:

```
import nltk
from collections import Counter
import string
fromnltk.corpus import stopwords

def get_tokens():
    with open('/home/d/TRY/NLTK/STOP.txt') as stopl:
        tokens = nltk.word_tokenize(stopl.read().lower().
translate(None, string.punctuation))
    return tokens

if __name__ == "__main__":

    tokens = get_tokens()
    print("tokens[:20]=%s") %(tokens[:20])

    count1 = Counter(tokens)
    print("before: len(count1) = %s") %(len(count1))

    filtered1 = [w for w in tokens if not w in stopwords.
words('english')]
```

```
print("filtered1 tokens[:20]=%s") %(filtered1[:20])

count1 = Counter(filtered1)
print("after: len(count1) = %s") %(len(count1))

print("most_common = %s") %(count.most_common(10))

tagged1 = nltk.pos_tag(filtered1)
print("tagged1[:20]=%s") %(tagged1[:20])
```

Information retrieval using a vector space model

In a vector space model, documents are represented as vectors. One of the methods of representing documents as vectors is using **TF-IDF** (**Term Frequency-Inverse Document Frequency**).

Term frequency may be defined as the total number of times a given token exists in a document divided by the total number of tokens. It may also be defined as the frequency of the occurrence of certain terms in a given document.

The formula for term frequency (TF) is given as follows:

$TF(t,d) = 0.5 + (0.5 * f(t,d)) / max \{f(w,d) : w \in d\}$

IDF may be defined as the inverse of document frequency. It is also defined as the document count that lies in the corpus in which a given term coexists.

IDF can be computed by finding the logarithm of the total number of documents present in a given corpus divided by the number of documents in which a particular token exists.

The formula for $IDF(t,d)$ may be stated as follows:

$IDF(t,D) = log(N/\{d \in D : t \in d\})$

The TF-IDF score can be obtained by multiplying both scores. This is written as follows:

$TF\text{-}IDF(t, d, D) = TF(t,d) * IDF(t,D)$

TF-IDF provides the estimate of the frequency of a term as present in the given document and how much it is being spread across the corpus.

In order to compute TF-IDF for a given document, the following steps are required:

- Tokenization of documents
- Computation of vector space model
- Computation of TF-IDF for each document

The process of tokenization involves tokenizing the text into sentences first. The individual sentences are then tokenized into words. The words, which are of no significance during information retrieval, also known as stop words, can then be removed.

Let's see the following code that can be used for performing tokenization on each document in a corpus:

```
authen = OAuthHandler(CLIENT_ID, CLIENT_SECRET, CALLBACK)
authen.set_access_token(ACCESS_TOKEN)
ap = API(authen)


venue = ap.venues(id='4bd47eeb5631c9b69672a230')
stopwords = nltk.corpus.stopwords.words('english')
tokenizer = RegexpTokenizer("[\w']+", flags=re.UNICODE)


def freq(word, tokens):
return tokens.count(word)


#Compute the frequency for each term.
vocabulary = []
docs = {}
all_tips = []
for tip in (venue.tips()):
tokens = tokenizer.tokenize(tip.text)

bitokens = bigrams(tokens)
tritokens = trigrams(tokens)
tokens = [token.lower() for token in tokens if len(token) > 2]
tokens = [token for token in tokens if token not in stopwords]

bitokens = [' '.join(token).lower() for token in bitokens]
bitokens = [token for token in bitokens if token not in stopwords]

tritokens = [' '.join(token).lower() for token in tritokens]
```

```
tritokens = [token for token in tritokens if token not in stopwords]

ftokens = []
ftokens.extend(tokens)
ftokens.extend(bitokens)
ftokens.extend(tritokens)
docs[tip.text] = {'freq': {}}

for token in ftokens:
docs[tip.text]['freq'][token] = freq(token, ftokens)

print docs
```

The next step performed after tokenization is the normalization of the `tf` vector. Let's see the following code that performs the normalization of the `tf` vector:

```
authen = OAuthHandler(CLIENT_ID, CLIENT_SECRET, CALLBACK)
authen.set_access_token(ACCESS_TOKEN)
ap = API(auth)


venue = ap.venues(id='4bd47eeb5631c9b69672a230')
stopwords = nltk.corpus.stopwords.words('english')
tokenizer = RegexpTokenizer("[\w']+", flags=re.UNICODE)


def freq(word, tokens):
return tokens.count(word)


def word_count(tokens):
return len(tokens)


def tf(word, tokens):
return (freq(word, tokens) / float(word_count(tokens)))

#Compute the frequency for each term.
vocabulary = []
docs = {}
all_tips = []
for tip in (venue.tips()):
tokens = tokenizer.tokenize(tip.text)

bitokens = bigrams(tokens)
```

```
tritokens = trigrams(tokens)
tokens = [token.lower() for token in tokens if len(token) > 2]
tokens = [token for token in tokens if token not in stopwords]

bitokens = [' '.join(token).lower() for token in bitokens]
bitokens = [token for token in bitokens if token not in stopwords]

tritokens = [' '.join(token).lower() for token in tritokens]
tritokens = [token for token in tritokens if token not in stopwords]

ftokens = []
ftokens.extend(tokens)
ftokens.extend(bitokens)
ftokens.extend(tritokens)
docs[tip.text] = {'freq': {}, 'tf': {}}

for token in ftokens:
        #The Computed  Frequency
docs[tip.text]['freq'][token] = freq(token, ftokens)
        # Normalized Frequency
docs[tip.text]['tf'][token] = tf(token, ftokens)


print docs
```

Let's see the following code for computing the TF-IDF:

```
authen = OAuthHandler(CLIENT_ID, CLIENT_SECRET, CALLBACK)
authen.set_access_token(ACCESS_TOKEN)
ap = API(authen)


venue = ap.venues(id='4bd47eeb5631c9b69672a230')
stopwords = nltk.corpus.stopwords.words('english')
tokenizer = RegexpTokenizer("[\w']+", flags=re.UNICODE)


def freq(word, doc):
return doc.count(word)


def word_count(doc):
return len(doc)


def tf(word, doc):
```

```
return (freq(word, doc) / float(word_count(doc)))


def num_docs_containing(word, list_of_docs):
count = 0
for document in list_of_docs:
if freq(word, document) > 0:
count += 1
return 1 + count


def idf(word, list_of_docs):
return math.log(len(list_of_docs) /
float(num_docs_containing(word, list_of_docs)))


#Compute the frequency for each term.
vocabulary = []
docs = {}
all_tips = []
for tip in (venue.tips()):
tokens = tokenizer.tokenize(tip.text)

bitokens = bigrams(tokens)
tritokens = trigrams(tokens)
tokens = [token.lower() for token in tokens if len(token) > 2]
tokens = [token for token in tokens if token not in stopwords]

bitokens = [' '.join(token).lower() for token in bitokens]
bitokens = [token for token in bitokens if token not in stopwords]

tritokens = [' '.join(token).lower() for token in tritokens]
tritokens = [token for token in tritokens if token not in stopwords]

ftokens = []
ftokens.extend(tokens)
ftokens.extend(bitokens)
ftokens.extend(tritokens)
docs[tip.text] = {'freq': {}, 'tf': {}, 'idf': {}}

for token in ftokens:
        #The frequency computed for each tip
docs[tip.text]['freq'][token] = freq(token, ftokens)
        #The term-frequency (Normalized Frequency)
```

```
docs[tip.text]['tf'][token] = tf(token, ftokens)

vocabulary.append(ftokens)

for doc in docs:
for token in docs[doc]['tf']:
        #The Inverse-Document-Frequency
docs[doc]['idf'][token] = idf(token, vocabulary)

print docs
```

TF-IDF is computed by finding the product of TF and IDF. The large value of TF-IDF is computed when there is an occurrence of high term frequency and low document frequency.

Let's see the following code for computing the TF-IDF for each term in a document:

```
authen = OAuthHandler(CLIENT_ID, CLIENT_SECRET, CALLBACK)
authen.set_access_token(ACCESS_TOKEN)
ap = API(authen)


venue = ap.venues(id='4bd47eeb5631c9b69672a230')
stopwords = nltk.corpus.stopwords.words('english')
tokenizer = RegexpTokenizer("[\w']+", flags=re.UNICODE)


def freq(word, doc):
return doc.count(word)


def word_count(doc):
return len(doc)


def tf(word, doc):
return (freq(word, doc) / float(word_count(doc)))


def num_docs_containing(word, list_of_docs):
count = 0
for document in list_of_docs:
if freq(word, document) > 0:
count += 1
```

```
    return 1 + count


    def idf(word, list_of_docs):
    return math.log(len(list_of_docs) /
    float(num_docs_containing(word, list_of_docs)))


    def tf_idf(word, doc, list_of_docs):
    return (tf(word, doc) * idf(word, list_of_docs))

    #Compute the frequency for each term.
    vocabulary = []
    docs = {}
    all_tips = []
    for tip in (venue.tips()):
    tokens = tokenizer.tokenize(tip.text)

    bitokens = bigrams(tokens)
    tritokens = trigrams(tokens)
    tokens = [token.lower() for token in tokens if len(token) > 2]
    tokens = [token for token in tokens if token not in stopwords]

    bitokens = [' '.join(token).lower() for token in bitokens]
    bitokens = [token for token in bitokens if token not in stopwords]

    tritokens = [' '.join(token).lower() for token in tritokens]
    tritokens = [token for token in tritokens if token not in stopwords]

    ftokens = []
    ftokens.extend(tokens)
    ftokens.extend(bitokens)
    ftokens.extend(tritokens)
    docs[tip.text] = {'freq': {}, 'tf': {}, 'idf': {},
                          'tf-idf': {}, 'tokens': []}

    for token in ftokens:
            #The frequency computed for each tip
    docs[tip.text]['freq'][token] = freq(token, ftokens)
            #The term-frequency (Normalized Frequency)
    docs[tip.text]['tf'][token] = tf(token, ftokens)
    docs[tip.text]['tokens'] = ftokens
```

```
vocabulary.append(ftokens)

for doc in docs:
for token in docs[doc]['tf']:
        #The Inverse-Document-Frequency
docs[doc]['idf'][token] = idf(token, vocabulary)
        #The tf-idf
docs[doc]['tf-idf'][token] = tf_idf(token, docs[doc]['tokens'],
vocabulary)

#Now let's find out the most relevant words by tf-idf.
words = {}
for doc in docs:
for token in docs[doc]['tf-idf']:
if token not in words:
words[token] = docs[doc]['tf-idf'][token]
else:
if docs[doc]['tf-idf'][token] > words[token]:
words[token] = docs[doc]['tf-idf'][token]

for item in sorted(words.items(), key=lambda x: x[1], reverse=True):
print "%f <= %s" % (item[1], item[0])
```

Let's see the following code for mapping keywords to the vector's dimension:

```
>>> def getVectkeyIndex(self,documentList):
    vocabString=" ".join(documentList)
    vocabList=self.parser.tokenise(vocabString)
    vocabList=self.parser.removeStopWords(vocabList)
    uniquevocabList=util.removeDuplicates(vocabList)
    vectorIndex={}
    offset=0


for word in uniquevocabList:
        vectorIndex[word]=offset
        offset+=1
return vectorIndex
```

Let's see the following code for mapping document strings to vectors:

```
>>> def makeVect(self,wordString):
    vector=[0]*len(self.vectorkeywordIndex)
    wordList=self.parser.tokenise(wordString)
    wordList=self.parser.removeStopWords(wordList)
    for word in wordList:
        vector[self.vectorkeywordIndex[word]]+=1;
    return vector
```

Vector space scoring and query operator interaction

Vector space model is used for the representation of meanings in the form of vectors of lexical items. A vector space model can easily be modeled using linear algebra. So the similarity between vectors can be computed easily.

Vector size is used to represent the size of the vector being used that represents a particular context. The window-based method and dependency-based method are used for the modeling context. In the window-based method, the context is determined using the occurrence of words within the window of a particular size. In a dependency-based method, the context is determined when there is an occurrence of a word in a particular syntactic relation with the corresponding target word. Features or contextual words are stemmed and lemmatized. Similarity metrics can be used to compute the similarity between the two vectors.

Let's see the following list of similarity metrics:

Measure	Definition
Euclidean	$\frac{1}{1+\sqrt{\sum_{i=1}^{n}(u_i-v_i)^2}}$
Cityblock	$\frac{1}{1+\sum_{i=1}^{n}\lvert u_i-v_i\rvert}$
Chebyshev	$\frac{1}{1+\max_i\lvert u_i-v_i\rvert}$
Cosine	$\frac{u.v}{\lvert u\rvert\lvert v\rvert}$
Correlation	$\frac{(u-\mu_u).(v-\mu_v)}{\lvert u\rvert\lvert v\rvert}$
Dice	$\frac{2\sum_{i=0}^{n} min(u_i,v_i)}{\sum_{i=0}^{n} u_i+v_i}$
Jaccard	$\frac{u.v}{\sum_{i=0}^{n} u_i+v_i}$
Jaccard2	$\frac{\sum_{i=0}^{n} min(u_i,v_i)}{\sum_{i=0}^{n} max(u_i,v_i)}$
Lin	$\frac{\sum_{i=0}^{n} u_i+v_i}{\lvert u\rvert+\lvert v\rvert}$

Tanimoto	$\frac{u.v}{\lvert u\rvert+\lvert v\rvert-u.v}$
Jensen-Shannon Div	$1-\frac{\frac{1}{2}\left(D\left(u\middle\|\frac{u+v}{2}\right)+D\left(v\middle\|\frac{u+v}{2}\right)\right)}{\sqrt{2\log 2}}$
α-skew	$1-\frac{D\left(u\middle\|\alpha v+(1-\alpha)u\right)}{\sqrt{2\log 2}}$

Weighting scheme is another term that is very important as it provides information that the given context is more related to the target word.

Let's see the list of weighting schemes that can be considered:

Scheme	Definition
None	$w_{ij}=f_{ij}$
TF-IDF	$w_{ij}=\log\left(f_{ij}\right)\times\log\left(\frac{N}{n_j}\right)$
TF-ICF	$w_{ij}=\log\left(f_{ij}\right)\times\log\left(\frac{N}{f_j}\right)$
Okapi BM25	$w_{ij}=\frac{f_{ij}}{0.5+1.5\times\frac{f_j}{\frac{f_j}{j}}+f_{ij}}\log\frac{N-n_j+0.5}{f_{ij}+0.5}$
ATC	$w_{ij}=\frac{\left(0.5+0.5\times\frac{f_{ij}}{\max_f}\right)\log\left(\frac{N}{n_j}\right)}{\sqrt{\sum_{i=1}^{N}\left[\left(0.5+0.5\times\frac{f_{ij}}{\max_f}\right)\log\left(\frac{N}{n_j}\right)\right]^2}}$
LTU	$w_{ij}=\frac{\left(\log\left(f_{ij}\right)+1.0\right)\log\left(\frac{N}{n_j}\right)}{0.8+0.2\times f_j\times\frac{j}{f_j}}$
MI	$w_{ij}=\log\frac{P\left(t_{ij}\mid c_j\right)}{P\left(t_{ij}\right)P\left(c_j\right)}$

PosMI	$\max(0, MI)$
T-Test	$w_{ij} = \frac{p(t_{ij} \mid c_j) - P(t_{ij})P(c_j)}{\sqrt{P(t_{ij})P(c_j)}}$
χ^2	$see(Curran, 2004, p.83)$
Lin98a	$w_{ij} = \frac{f_{ij} \times f}{f_i \times f_j}$
Lin98b	$w_{ij} = -1 \times \log \frac{n_j}{N}$
Gref94	$w_{ij} = \frac{\log f_{ij} + 1}{\log n_i + 1}$

Developing an IR system using latent semantic indexing

Latent semantic indexing can be used for performing categorization with the help of minimum training.

Latent semantic indexing is a technique that can be used for processing text. It can perform the following:

- Automatic categorization of text
- Conceptual information retrieval
- Cross-lingual information retrieval

Latent semantic method may be defined as an information retrieval and indexing method. It makes use of a mathematical technique known as **Singular Value Decomposition** (**SVD**). SVD is used for the detection of patterns having a certain relation with the concepts contained in a given unstructured text.

Some of the applications of latent semantic indexing include the following:

- Information discovery
- Automated document classification text summarization[20] (eDiscovery, Publishing)
- Relationship discovery
- Automatic generation of the link charts of individuals and organizations

- Matching technical papers and grants with reviewers
- Online customer support
- Determining document authorship
- Automatic keyword annotation of images
- Understanding software source code
- Filtering spam
- Information visualization
- Essay scoring
- Literature-based discovery

Text summarization

Text summarization is the process of generating summaries from a given long text. Based on the Luhn work, *The Automatic Creation of Literature Abstracts* (1958), a naïve summarization approach known as NaiveSumm is developed. It makes use of a word's frequencies for the computation and extraction of sentences that consist of the most frequent words. Using this approach, text summarization can be performed by extracting a few specific sentences.

Let's see the following code in NLTK that can be used for performing text summarization:

```
from nltk.tokenize import sent_tokenize,word_tokenize
from nltk.corpus import stopwords
from collections import defaultdict
from string import punctuation
from heapq import nlargest

class Summarize_Frequency:
  def __init__(self, cut_min=0.2, cut_max=0.8):
    """
     Initilize the text summarizer.
     Words that have a frequency term lower than cut_min
     or higer than cut_max will be ignored.
    """
    self._cut_min = cut_min
    self._cut_max = cut_max
    self._stopwords = set(stopwords.words('english') +
list(punctuation))

  def _compute_frequencies(self, word_sent):
```

```
        """
          Compute the frequency of each of word.
          Input:
           word_sent, a list of sentences already tokenized.
          Output:
           freq, a dictionary where freq[w] is the frequency of w.
        """
        freq = defaultdict(int)
        for s in word_sent:
          for word in s:
            if word not in self._stopwords:
              freq[word] += 1
        # frequencies normalization and fitering
        m = float(max(freq.values()))
        for w in freq.keys():
          freq[w] = freq[w]/m
          if freq[w] >= self._cut_max or freq[w] <= self._cut_min:
            del freq[w]
        return freq

      def summarize(self, text, n):
        """
    list of (n) sentences are returned.
    summary of text is returned.
        """
        sents = sent_tokenize(text)
        assert n <= len(sents)
        word_sent = [word_tokenize(s.lower()) for s in sents]
        self._freq = self._compute_frequencies(word_sent)
        ranking = defaultdict(int)
        for i,sent in enumerate(word_sent):
          for w in sent:
            if w in self._freq:
              ranking[i] += self._freq[w]
        sents_idx = self._rank(ranking, n)
        return [sents[j] for j in sents_idx]

      def _rank(self, ranking, n):
        """ return the first n sentences with highest ranking """
        return nlargest(n, ranking, key=ranking.get)
```

The preceding code computes the term frequency for each word and then the most frequent words, such as determiners, may be eliminated as they are not of much use while performing information retrieval tasks.

Question-answering system

Question-answering systems are referred to as intelligent systems that can be used to provide responses for the questions being asked by the user based on certain facts or rules stored in the knowledge base. So the accuracy of a question-answering system to provide a correct response depends on the rules or facts stored in the knowledge base.

One of the many issues involved in a question-answering system is how the responses and questions would be represented in the system. Responses may be retrieved and then represented using text summarization or parsing. Another issue involved in the question-answering system is how the questions and the corresponding answers are represented in a knowledge base.

To build a question-answering system, various approaches, such as the named entity recognition, information retrieval, information extraction, and so on, can be applied.

A question-answering system involves three phases:

- Extraction of facts
- Understanding of questions
- Generation of answers

Extraction of facts is performed in order to understand domain-specific data and generate a response for a given query.

Extraction of facts can be performed in two ways using: extraction of entity and extraction of relation. The process of extraction of entity or extraction of proper nouns is referred to as NER. The process of extraction of relation is based on the extraction of semantic information from the text.

Understanding of questions involves the generation of a parse tree from a given text.

The generation of answers involves obtaining the most likely response for a given query that can be understood by the user.

Let's see the following code in NLTK that can be used to accept a query from a user user. This query can be processed by removing stop words from it so that information retrieval can be performed post processing:

```
import nltk
from nltk import *
import string
print "Enter your question"
ques=raw input()
```

```
ques=ques.lower()
stopwords=nltk.corpus.stopwords.words('english')
cont=nltk.word_tokenize(question)
analysis_keywords=list( set(cont) -set(stopwords) )
```

Summary

In this chapter, we have discussed information retrieval. We have mainly learned about stop words removal. Stop words are eliminated so that information retrieval and text summarization tasks become faster. We have also discussed the implementation of text summarization, question-answering systems, and vector space models.

In the next chapter, we'll study the concepts of discourse analysis and anaphora resolution.

9
Discourse Analysis – Knowing Is Believing

Discourse analysis is another one of the applications of Natural Language Processing. Discourse analysis may be defined as the process of determining contextual information that is useful for performing other tasks, such as **anaphora resolution (AR)** (we will cover this section later in this chapter), NER, and so on.

This chapter will include the following topics:

- Introducing discourse analysis
- Discourse analysis using Centering Theory
- Anaphora resolution

Introducing discourse analysis

The word *discourse* in linguistic terms means language in use. Discourse analysis may be defined as the process of performing text or language analysis, which involves text interpretation and knowing the social interactions. Discourse analysis may involve dealing with morphemes, n-grams, tenses, verbal aspects, page layouts, and so on. Discourse may be defined as the sequential set of sentences.

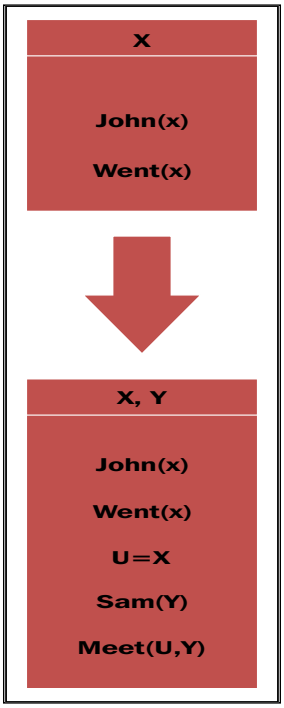
In most cases, we can interpret the meaning of the sentence on the basis of the preceding sentences.

Consider a discourse *John went to the club on Saturday. He met Sam*." Here, *He* refers to John.

Discourse Representation Theory (**DRT**) has been developed to provide a means for performing AR. A **Discourse Representation Structure** (**DRS**) has been developed that provides the meaning of discourse with the help of discourse referents and conditions. Discourse referents refer to variables used in first-order logic and things under consideration in a discourse. A discourse representation structure's conditions refer to the atomic formulas used in first-order predicate logic.

First Order Predicate Logic (**FOPL**) was developed to extend the idea of propositional logic. FOPL involves the use of functions, arguments, and quantifiers. Two types of quantifiers are used to represent the general sentences, namely, universal quantifiers and existential quantifiers. In FOPL, connectives, constants, and variables are also used. For instance, `Robin is a bird` can be represented in FOPL as `bird (robin)`.

Let's see an example of the discourse representation structure:

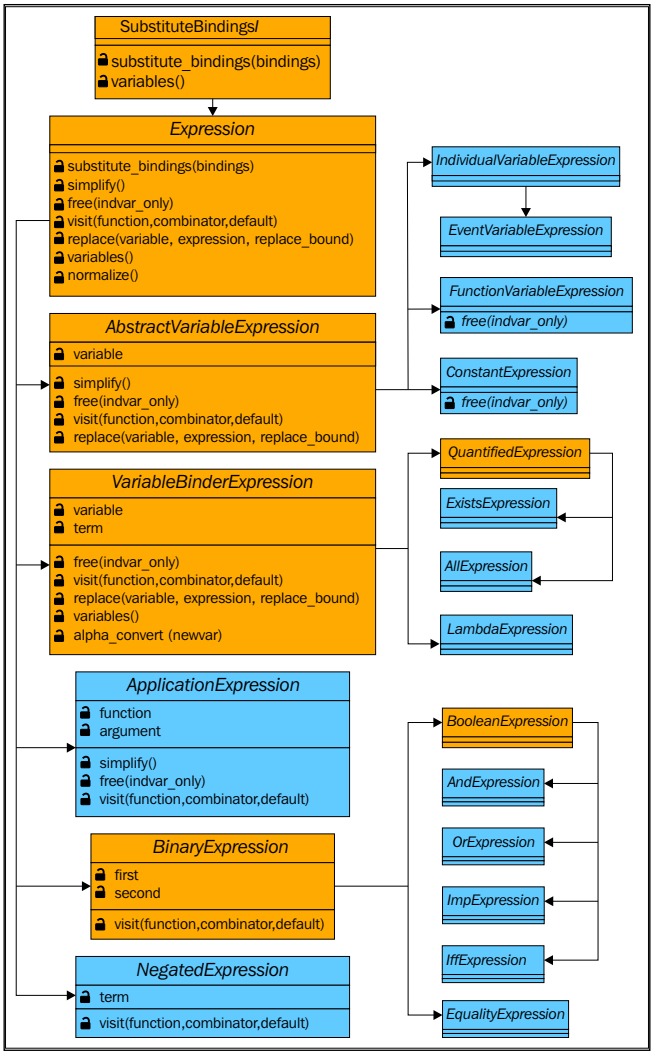


The preceding diagram is a representation of the following sentences:

1. John went to a club
2. John went to a club. He met Sam.

Here, the discourse consists of two sentences. Discourse Structure Representation may represent the entire text. For computationally processing DRS, it needs to be converted into a linear format.

The NLTK module that can be used to provide first order predicate logic implementation is `nltk.sem.logic`. Its UML diagram is shown here:



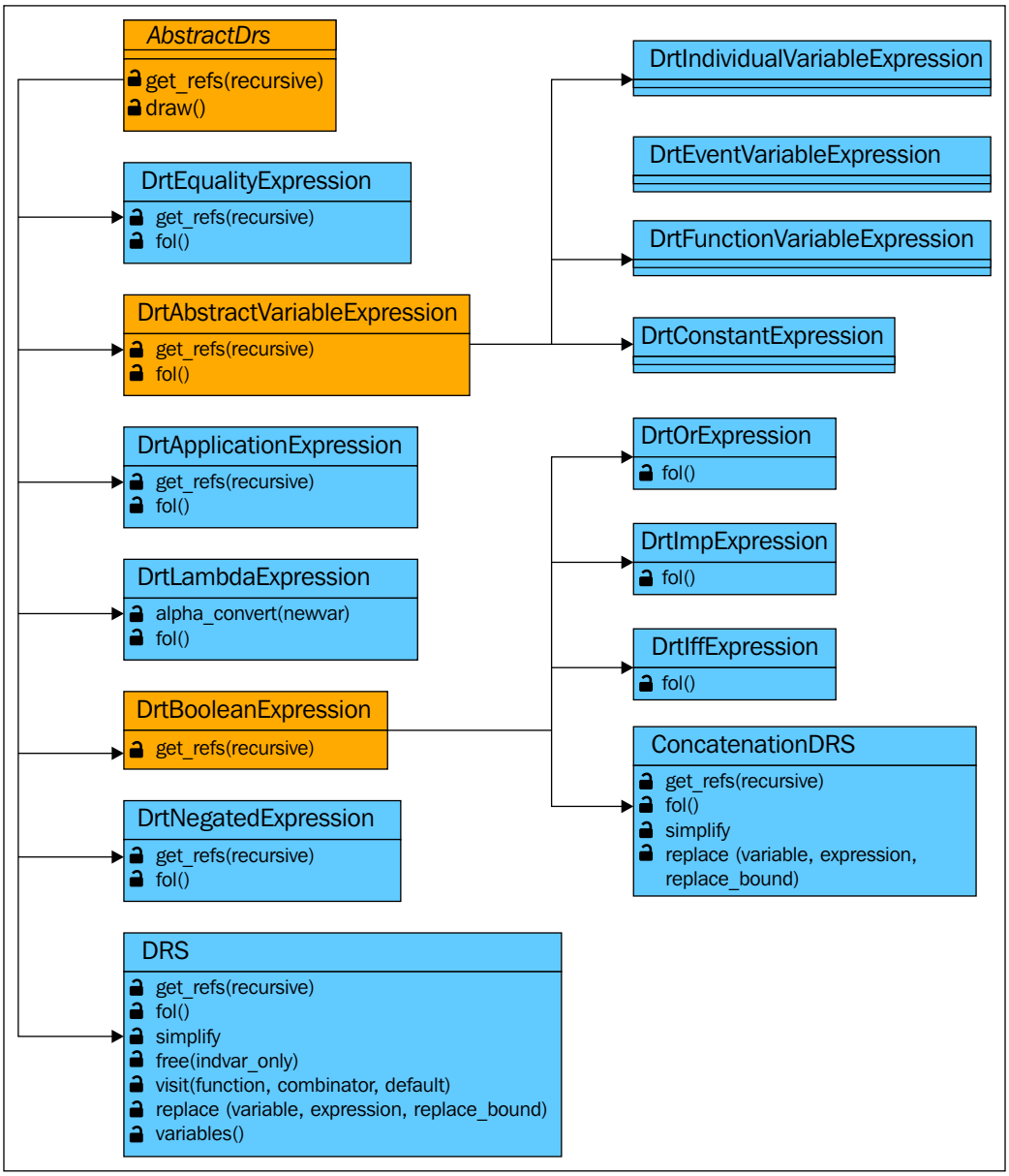
The `nltk.sem.logic` module is used to define the expressions of first order predicate logic. Its UML diagram is comprised of various classes that are required for the representation of objects in first order predicate logic as well as their methods. The methods that are included are as follows:

- `substitute_bindings(bindings)`: Here, binding represents variable-to-expression mapping. It replaces variables present in the expression with a specific value.
- `Variables()`: This comprises a set of all the variables that need to be replaced. It consists of constants as well as free variables.
- `replace(variable, expression, replace_bound)`: This is used for substituting the expression for a variable instance; `replace_bound` is used to specify whether we need to replace bound variables or not.
- `Normalize()`: This is used to rename the autogenerated unique variables.
- `Visit(self,function,combinatory,default)`: This is used to visit subexpression calling functions; results are passed to the combinator that begins with a default value. Results of the combination are returned.
- `free(indvar_only)`: This is used to return the set of all the free variables of the object. Individual variables are returned if `indvar_only` is set to `True`.
- `Simplify()`: This is used to simplify the expression that represents an object.

The NLTK module that provides a base for the discourse representation theory is `nltk.sem.drt`. It is built on top of `nltk.sem.logic`. Its UML class diagram comprises classes that are inherited from the `nltk.sem.logic` module. The following are the methods described in this module:

- The `get_refs(recursive)`: This method obtains the referents for the current discourse.
- The `fol()`: This method is used for the conversion of DRS into first order predicate logic.
- The `draw()`: This method is used for drawing DRS with the help of the Tkinter graphics library.

Let's see the UML class diagram of the `nltk.sem.drt` module:



The NLTK module that provides access to WordNet 3.0 is `nltk.corpus.reader.wordnet`.

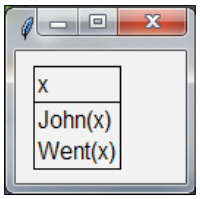
Linear format comprises discourse referents and DRS conditions, for example:

([x], [John(x), Went(x)])

Let's see the following code in NLTK, which can be used for the implementation of DRS:

```
>>> import nltk
>>> expr_read = nltk.sem.DrtExpression.from string
>>> expr1 = expr_read('([x], [John(x), Went(x)])')
>>> print(expr1)
([x],[John(x), Went(x)])
>>> expr1.draw()
>>> print(expr1.fol())
exists x.(John(x) & Went(x))
```

The preceding code of NLTK will draw the following image:

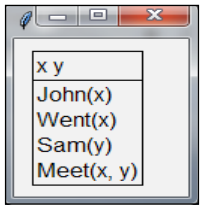


Here, the expression is converted into FOPL using the `fol()` method.

Let's see the following code in NLTK for the other expression:

```
>>> import nltk
>>> expr_read = nltk.sem.DrtExpression.from string
>>> expr2 = expr_read('([x,y], [John(x), Went(x),Sam(y),Meet(x,y)])')
>>> print(expr2)
([x,y],[John(x), Went(x), Sam(y), Meet(x,y)])
>>> expr2.draw()
>>> print(expr2.fol())
exists x y.(John(x) & Went(x) & Sam(y) & Meet(x,y))
```

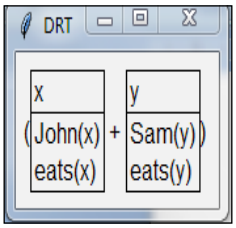
The `fol()` function is used to obtain the first order predicate logic equivalent of the expression. The preceding code displays the following image:



We can perform the concatenation of two DRS using the DRS concatenation operator (+). Let's see the following code in NLTK that can be used to perform the concatenation of two DRS:

```
>>> import nltk
>>> expr_read = nltk.sem.DrtExpression.from_string
>>> expr3 = expr_read('([x], [John(x), eats(x)])+
([y],[Sam(y),eats(y)])')
>>> print(expr3)
(([x],[John(x), eats(x)]) + ([y],[Sam(y), eats(y)]))
>>> print(expr3.simplify())
([x,y],[John(x), eats(x), Sam(y), eats(y)])
>>> expr3.draw()
```

The preceding code draws the following image:



Here, `simplify()` is used to simplify the expression.

Let's see the following code in NLTK, which can be used to embed one DRS into another:

```
>>> import nltk
>>> expr_read = nltk.sem.DrtExpression.from string
>>> expr4 = expr_read('([],[(([x],[student(x)])-
>([y],[book(y),read(x,y)]))])')
>>> print(expr4.fol())
all x.(student(x) -> exists y.(book(y) & read(x,y)))
```

Let's see another example that can be used to combine two sentences. Here, PRO has been used and `resolve_anaphora()` is used to perform AR:

```
>>> import nltk
>>> expr_read = nltk.sem.DrtExpression.from string
>>> expr5 = expr_read('([x,y],[ram(x),food(y),eats(x,y)])')
>>> expr6 = expr_read('([u,z],[PRO(u),coffee(z),drinks(u,z)])')
>>> expr7=expr5+expr6
>>> print(expr7.simplify())
([u,x,y,z],[ram(x), food(y), eats(x,y), PRO(u), coffee(z),
drinks(u,z)])
>>> print(expr7.simplify().resolve_anaphora())
([u,x,y,z],[ram(x), food(y), eats(x,y), (u = [x,y,z]), coffee(z),
drinks(u,z)])
```

Discourse analysis using Centering Theory

Discourse analysis using Centering Theory is the first step toward corpus annotation. It also involves the task of AR. In Centering Theory, we perform the task of segmenting discourse into various units for analysis.

Centering Theory involves the following:

- Interaction between purposes or intentions of discourse participants and discourse
- Attention of participants
- Discourse structure

Centering is related to participants attention and how the local as well as global structures affect expressions and the coherence of discourse.

Anaphora resolution

AR may be defined as the process by which a pronoun or a noun phrase used in the sentence is resolved and refers to a specific entity on the basis of discourse knowledge.

For example:

```
John helped Sara. He was kind.
```

Here, `He` refers to `John`.

AR is of three types, namely:

- **Pronominal**: Here, the referent is referred to by a pronoun. For example, `Sam` found the love of his life. Here, 'his' refers to 'Sam'.
- **Definite noun phrase**: Here, the antecedent may be referred to by the phrase of the form, `<the><noun phrase>`. For example, `The relationship could not last long`. Here, `The relationship` refers to `the love` in the previous sentence.
- **Quantifier/ordinal**: The quantifier, such as one, and the ordinal, such as first, are also examples of AR. For example, `He began a new one`.
  Here, `one` refers to `the relationship`.

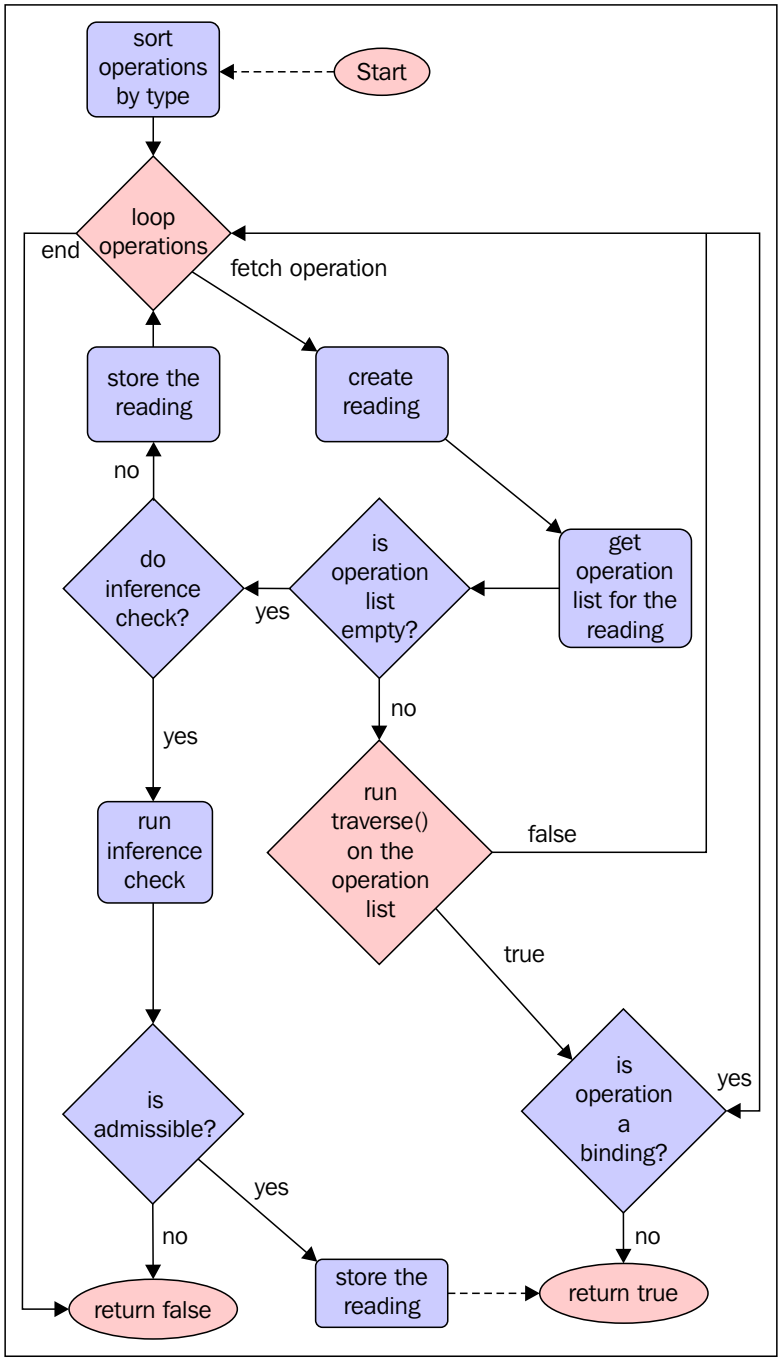
In cataphora, the referent precedes the antecedent. For example, `After his class, Sam will go home`. Here, `his` refers to `Sam`.

For integrating some extensions in a NLTK architecture, a new module is developed on top of the existing modules, `nltk.sem.logic` and `nltk.sem.drt`. The new module acts like a replacement for the `nltk.sem.drt` module. There is a replacement of all the classes with the enhanced classes.

A method called `resolve()` can be called indirectly and directly from a class called `AbstractDRS()`. It then provides a list consisting of resolved copies of a particular object. An object that needs to be resolved must override the `readings()` method. The `resolve()` method is used to generate readings using the `traverse()` function. The `traverse()` function is used to perform sorting on the list of operations.
A priority order list includes the following:

- Binding operations
- Local accommodation operations
- Intermediate accommodation operations
- Global accommodation operations

Let's see the flow diagram of the `traverse()` function:



After the priority order of operations is generated, the following takes place:

- Readings are generated from the operation with the help of the `deepcopy()` method. The current operation is taken as an argument.
- When the `readings()` method runs, a list of operations are performed.
- Till the list of operations is not empty, run is performed on those operations.
- If there are no operations left to be performed, admissibility check will be run on the final reading; if the check is successful, it will be stored.

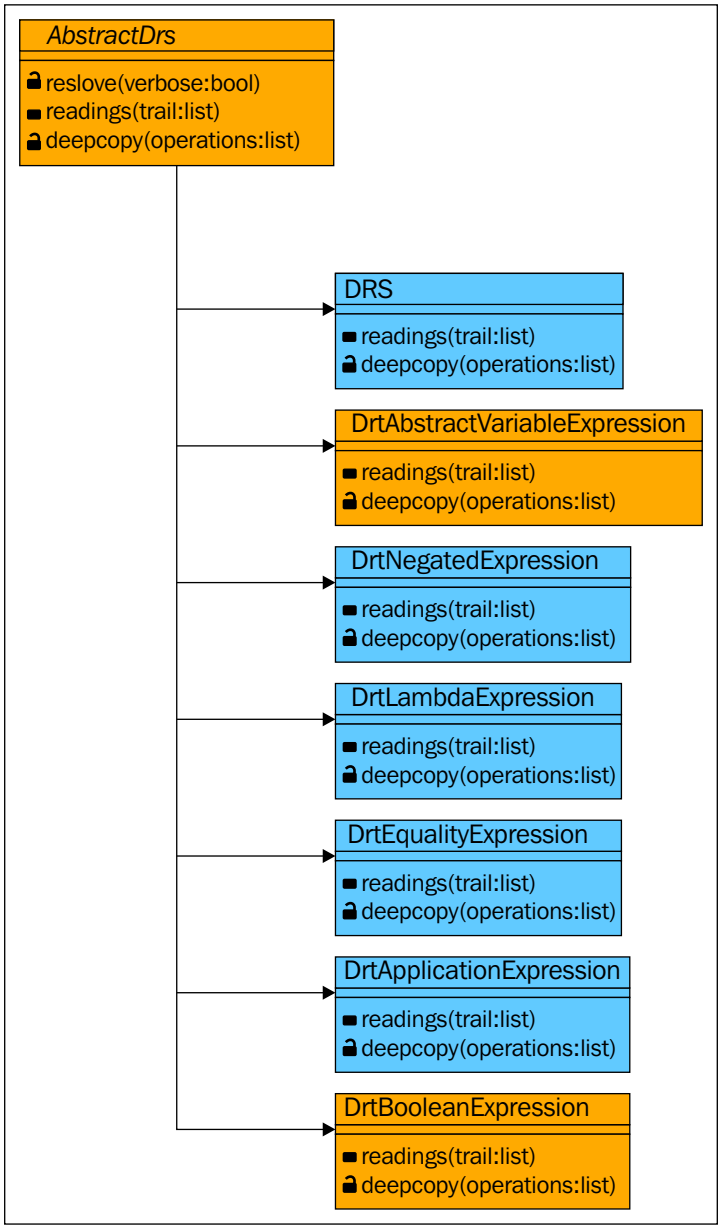
In `AbstractDRS()`, the `resolve()` method is defined. It is defined as follows:

def resolve(self, verbose=False)

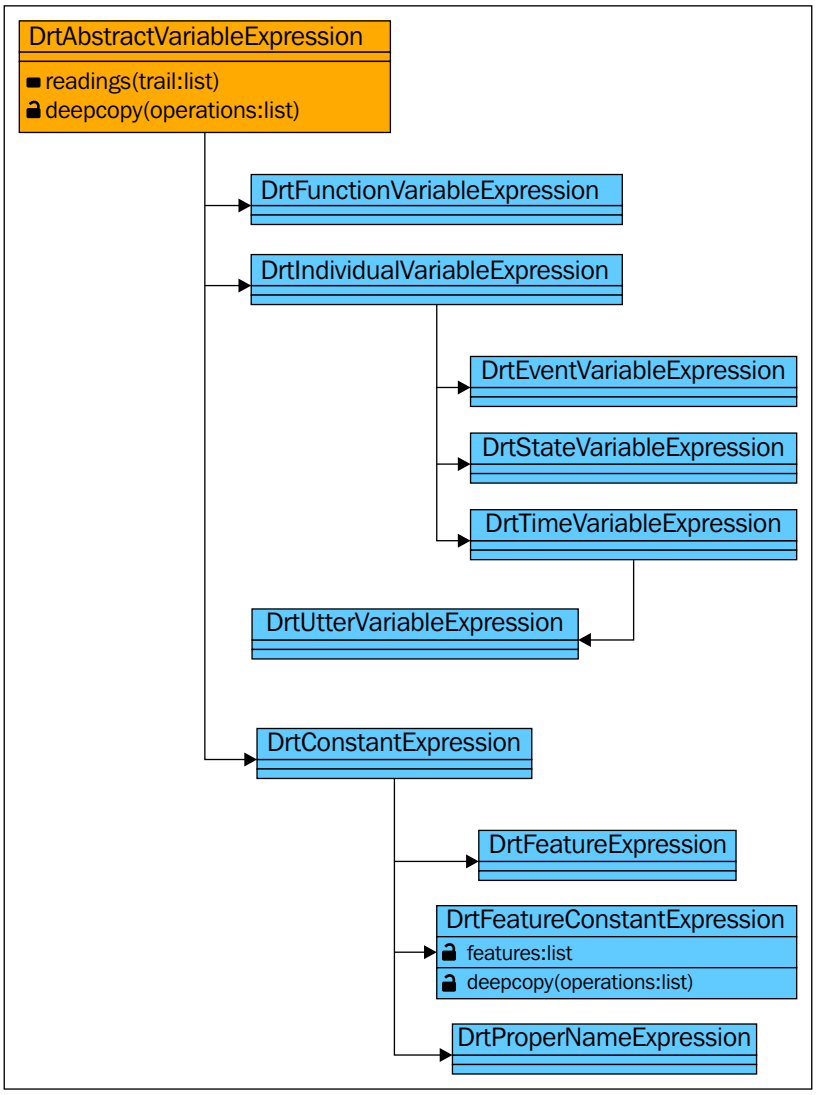
The `PresuppositionDRS` class includes the following methods:

- `find_bindings(drs_list, collect_event_data)`: Bindings are found from the list of DRS instances using the `is_possible_binding` method. Collection of participation information is done if `collect_event_data` is set to `True`.
- `is_possible_binding(cond)`: This finds out whether the condition is a binding candidate and makes sure that it is an unary predicate with the features that match the trigger conditions.
- `is_presupposition.cond(cond)`: This is used to identify a trigger condition among all the conditions.
- `presupposition_readings(trail)`: This is like readings in the subclasses of `PresuppositionDRS`.

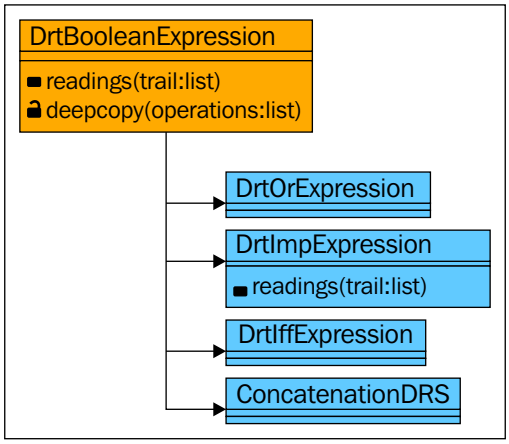
Let's see the classes that are inherited from `AbstractDRS`:



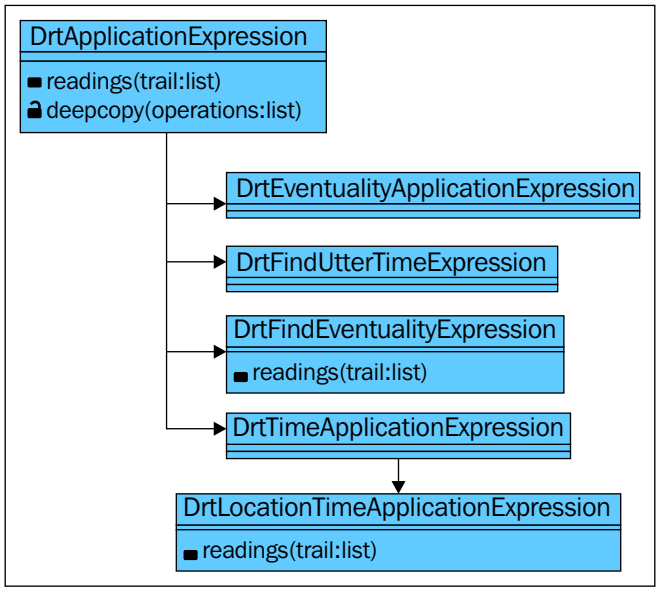
Let's see the classes that are inherited in `DRTAbstractVariableExpression`:



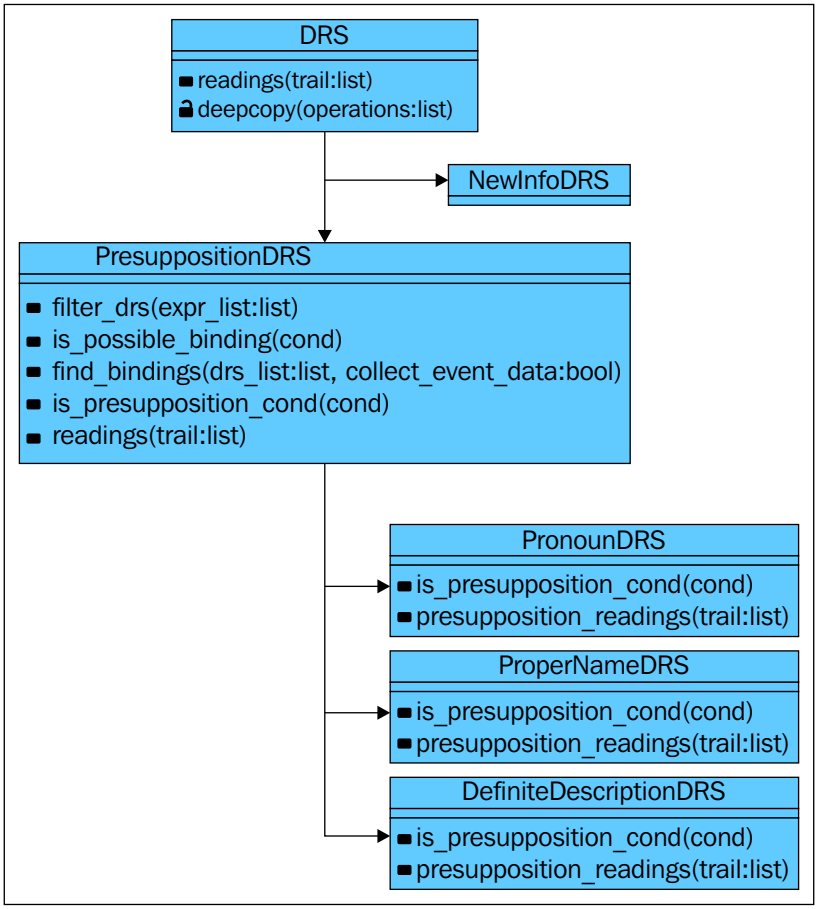
Let's see the classes inherited from `DrtBooleanExpression`:



Let's see the classes inherited from `DrtApplicationExpression`:



Let's see the classes inherited from DRS:



Summary

In this chapter, we have discussed discourse analysis, discourse analysis using Centering Theory, and anaphora resolution. We have discussed the discourse representation structure that is built using first order predicate logic. We have also discussed how NLTK can be used to implement first order predicate logic using UML diagrams.

In the next chapter, we will discuss the evaluation of NLP Tools. We will also discuss various metrics for error identification, lexical matching, syntactic matching, and shallow semantic matching.

10
Evaluation of NLP Systems – Analyzing Performance

The evaluation of NLP systems is performed so that we can analyze whether a given NLP system produces the desired result or not and the desired performance is achieved or not. Evaluation may be performed automatically using predefined metrics, or it may be performed manually by comparing human output with the output obtained by an NLP system.

This chapter will include the following topics:

- The need for the evaluation of NLP systems
- Evaluation of NLP tools (POS Taggers, Stemmers, and Morphological Analyzers)
- Parser evaluation using gold data
- The evaluation of an IR system
- Metrics for error identification
- Metrics based on lexical matching
- Metrics based on syntactic matching
- Metrics using shallow semantic matching

The need for evaluation of NLP systems

Evaluation of NLP systems is done so as to analyze whether the output given by the NLP systems is similar to the one expected from the human output. If errors in the module are identified at an early stage, then the cost of correcting the NLP system is reduced to quite an extent.

Suppose we want to evaluate a tagger. We can do this by comparing the output of the tagger with the human output. Many a times, we do not have access to an impartial or expert human. So we can construct a gold standard test data to perform the evaluation of our tagger. This is a corpus, which has been tagged manually and is considered as a standard corpus that can be used for the evaluation of our tagger. The tagger is considered as correct if the output in the form of a tag given by the tagger is the same as that provided by the gold standard test data.

Creation of a gold standard annotated corpus is a major task and is also very expensive. It is performed by manually tagging a given test data. The tags chosen in this manner are taken as standard tags that can be used to represent a wide range of information.

Evaluation of NLP tools (POS taggers, stemmers, and morphological analyzers)

We can perform the evaluation of NLP systems, such as POS taggers, stemmers, morphological analyzers, NER-based systems, machine translators, and so on. Consider the following code in NLTK that can be used to train a unigram tagger. Sentence tagging is performed and then an evaluation is done to check whether the output given by the tagger is the same as the gold standard test data:

```
>>> import nltk
>>>from nltk.corpus import brown
>>> sentences=brown.tagged_sents(categories='news')
>>> sent=brown.sents(categories='news')
>>> unigram_sent=nltk.UnigramTagger(sentences)
>>> unigram_sent.tag(sent[2008])
[('Others', 'NNS'), (',', ','), ('which', 'WDT'), ('are', 'BER'),
('reached', 'VBN'), ('by', 'IN'), ('walking', 'VBG'), ('up', 'RP'),
('a', 'AT'), ('single', 'AP'), ('flight', 'NN'), ('of', 'IN'),
('stairs', 'NNS'), (',', ','), ('have', 'HV'), ('balconies', 'NNS'),
('.', '.')]
>>> unigram_sent.evaluate(sentences)
0.9349006503968017
```

Consider the following code in NLTK in which the training and testing of Unigram tagger is performed on separate data. A given data is split into 80% training data and 20% testing data:

```
>>> import nltk
>>> from nltk.corpus import brown
>>> sentences=brown.tagged_sents(categories='news')
>>> sz=int(len(sentences)*0.8)
```

```
>>> sz
3698
>>> training_sents = sentences[:sz]
>>> testing_sents=sentences[sz:]
>>> unigram_tagger=nltk.UnigramTagger(training_sents)
>>> unigram_tagger.evaluate(testing_sents)
0.8028325063827737
```

Consider the following code in NLTK that demonstrates the use of N-Gram tagger. Here, Training corpus consists of tagged data. Also, in the following example, we have used a special case of n-gram tagger, that is, bigram tagger:

```
>>> import nltk
>>> from nltk.corpus import brown
>>> sentences=brown.tagged_sents(categories='news')
>>> sz=int(len(sentences)*0.8)
>>> training_sents = sentences[:sz]
>>> testing_sents=sentences[sz:]
>>> bigram_tagger=nltk.UnigramTagger(training_sents)
>>> bigram_tagger=nltk.BigramTagger(training_sents)
>>> bigram_tagger.tag(sentences[2008])
[(('Others', 'NNS'), None), ((',', ','), None), (('which', 'WDT'),
None), (('are', 'BER'), None), (('reached', 'VBN'), None), (('by',
'IN'), None), (('walking', 'VBG'), None), (('up', 'IN'), None), (('a',
'AT'), None), (('single', 'AP'), None), (('flight', 'NN'), None),
(('of', 'IN'), None), (('stairs', 'NNS'), None), ((',', ','), None),
(('have', 'HV'), None), (('balconies', 'NNS'), None), (('.', '.'),
None)]
>>> un_sent=sentences[4203]
>>> bigram_tagger.tag(un_sent)
[(('The', 'AT'), None), (('population', 'NN'), None), (('of', 'IN'),
None), (('the', 'AT'), None), (('Congo', 'NP'), None), (('is', 'BEZ'),
None), (('13.5', 'CD'), None), (('million', 'CD'), None), ((',',
','), None), (('divided', 'VBN'), None), (('into', 'IN'), None),
(('at', 'IN'), None), (('least', 'AP'), None), (('seven', 'CD'),
None), (('major', 'JJ'), None), (('``', '``'), None), (('culture',
'NN'), None), (('clusters', 'NNS'), None), (("''", "''"), None),
(('and', 'CC'), None), (('innumerable', 'JJ'), None), (('tribes',
'NNS'), None), (('speaking', 'VBG'), None), (('400', 'CD'), None),
(('separate', 'JJ'), None), (('dialects', 'NNS'), None), (('.', '.'),
None)]
>>> bigram_tagger.evaluate(testing_sents)
0.09181559805385615
```

Another way of tagging can be performed by means of bootstrapping different methods. In this approach, tagging can be performed using a bigram `Tagger`. If the tag is not found using the bigram `Tagger`, then a back-off method involving a unigram `Tagger` can be used. Also, if a tag is not found using a unigram `Tagger`, then a back-off method involving a default tagger can be used.

Let's see the following code in NLTK that implements combined `Tagger`:

```
>>> import nltk
>>> from nltk.corpus import brown
>>> sentences=brown.tagged_sents(categories='news')
>>> sz=int(len(sentences)*0.8)
>>> training_sents = sentences[:sz]
>>> testing_sents=sentences[sz:]
>>> s0=nltk.DefaultTagger('NNP')
>>> s1=nltk.UnigramTagger(training_sents,backoff=s0)
>>> s2=nltk.BigramTagger(training_sents,backoff=s1)
>>> s2.evaluate(testing_sents)
0.8122260224480948
```

The linguists use the following clues to determine the category of a word:

- Morphological clues
- Syntactic clues
- Semantic clues

Morphological clues are those in which prefix, suffix, infix, and affix information are used to determine the category of a word. For example, *ment* is a suffix that combines with a verb to form a noun, such as *establish* + *ment* = *establishment* and *achieve* + *ment* = *achievement*.

Syntactic clues can be useful in determining the category of a word. For example, let's assume that nouns are already known. Now, adjectives can be determined. Adjectives can occur either after a noun or after a word, such as very, in a sentence.

Semantic information can also be used to determine the category of a word. If the meaning of a word is known, then its category can easily be known.

Let's see the following code in NLTK that can be used for the evaluation of a chunk parser:

```
>>> import nltk
>>> chunkparser = nltk.RegexpParser("")
>>> print(nltk.chunk.accuracy(chunkparser, nltk.corpus.conll2000.
chunked_sents('train.txt', chunk_types=('NP',))))
0.44084599507856814
```

Let's see another code in NLTK that is based on the evaluation of a naïve chunk parser that looks for tags, such as CD, JJ, and so on:

```
>>> import nltk
>>> grammar = r"NP: {<[CDJNP].*>+}"
>>> cp = nltk.RegexpParser(grammar)
>>> print(nltk.chunk.accuracy(cp, nltk.corpus.conll2000.chunked_
sents('train.txt', chunk_types=('NP',))))
0.8744798726662164
```

The following code in NLTK is used to compute the conditional frequency distribution for chunked data:

```
def chunk_tags(train):
    """Generate a following tags list that appears inside chunks"""
    cfreqdist = nltk.ConditionalFreqDist()
    for t in train:
        for word, tag, chunktag in nltk.chunk.tree2conlltags(t):
            if chtag == "O":
                cfreqdist[tag].inc(False)
            else:
                cfreqdist[tag].inc(True)
    return [tag for tag in cfreqdist.conditions() if cfreqdist[tag].
max() == True]
>>> training_sents = nltk.corpus.conll2000.chunked_sents('train.txt',
chunk_types=('NP',))
>>> print chunked_tags(train_sents)
['PRP$', 'WDT', 'JJ', 'WP', 'DT', '#', '$', 'NN', 'FW', 'POS',
'PRP', 'NNS', 'NNP', 'PDT', 'RBS', 'EX', 'WP$', 'CD', 'NNPS', 'JJS',
'JJR']
```

Let's see the following code for performing the evaluation of `chunker` in NLTK. Here, two entities, namely `guessed` and `correct`, are used. Guessed entities are those that are returned by a chunk parser. Correct entities are those set of chunks that are defined in the test corpus:

```
>>> import nltk
>>> correct = nltk.chunk.tagstr2tree(
"[ the/DT little/JJ cat/NN ] sat/VBD on/IN [ the/DT mat/NN ]")
>>> print(correct.flatten())
(S the/DT little/JJ cat/NN sat/VBD on/IN the/DT mat/NN)
>>> grammar = r"NP: {<[CDJNP].*>+}"
>>> cp = nltk.RegexpParser(grammar)
>>> grammar = r"NP: {<PRP|DT|POS|JJ|CD|N.*>+}"
```

```
>>> chunk_parser = nltk.RegexpParser(grammar)
>>> tagged_tok = [("the", "DT"), ("little", "JJ"), ("cat",
"NN"),("sat", "VBD"), ("on", "IN"), ("the", "DT"), ("mat", "NN")]
>>> chunkscore = nltk.chunk.ChunkScore()
>>> guessed = cp.parse(correct.flatten())
>>> chunkscore.score(correct, guessed)
>>> print(chunkscore)
ChunkParse score:
    IOB Accuracy: 100.0%
    Precision:    100.0%
    Recall:       100.0%
    F-Measure:    100.0%
```

Let's see the following code in NLTK that can be used for the evaluation of unigram `chunker` and bigram `chunker`:

```
>>>chunker_data = [[(t,c) for w,t,c in nltk.chunk.
tree2conlltags(chtree)]
>>>              for chtree in nltk.corpus.conll2000.chunked_
sents('train.txt')]
>>> unigram_chunk = nltk.UnigramTagger(chunker_data)
>>> print nltk.tag.accuracy(unigram_chunk, chunker_data)
0.781378851068
>>> bigram_chunk = nltk.BigramTagger(chunker_data, backoff=unigram_
chunker)
>>> print nltk.tag.accuracy(bigram_chunk, chunker_data)
0.893220987404
```

Consider the following code in which the suffix of a word is used to determine the part of a speech tag. A classifier is trained to provide a list of informative suffixes. A feature extractor function has been used that checks the suffixes that are present in a given word:

```
>>> from nltk.corpus import brown
>>> suffix_freqdist = nltk.FreqDist()
>>> for wrd in brown.words():
...     wrd = wrd.lower()
...     suffix_freqdist[wrd[-1:]] += 1
...     suffix_fdist[wrd[-2:]] += 1
...     suffix_fdist[wrd[-3:]] += 1
>>> common_suffixes = [suffix for (suffix, count) in suffix_freqdist.
most_common(100)]
>>> print(common_suffixes)
['e', ',', '.', 's', 'd', 't', 'he', 'n', 'a', 'of', 'the',
 'y', 'r', 'to', 'in', 'f', 'o', 'ed', 'nd', 'is', 'on', 'l',
```

```
 'g', 'and', 'ng', 'er', 'as', 'ing', 'h', 'at', 'es', 'or',
 're', 'it', '``', 'an', "''", 'm', ';', 'i', 'ly', 'ion', ...]


>>> def pos_feature(wrd):
...     feature = {}
...     for suffix in common_suffixes:
...         feature['endswith({})'.format(suffix)] = wrd.lower().
endswith(suffix)
...     return feature
>>> tagged_wrds = brown.tagged_wrds(categories='news')
>>> featureset = [(pos_feature(n), g) for (n,g) in tagged_wrds]
>>> size = int(len(featureset) * 0.1)
>>> train_set, test_set = featureset[size:], featureset[:size]
>>> classifier1 = nltk.DecisionTreeClassifier.train(train_set)
>>> nltk.classify.accuracy(classifier1, test_set)
0.62705121829935351


>>> classifier.classify(pos_features('cats'))
'NNS'

>>> print(classifier.pseudocode(depth=4))
if endswith(,) == True: return ','
if endswith(,) == False:
  if endswith(the) == True: return 'AT'
  if endswith(the) == False:
    if endswith(s) == True:
      if endswith(is) == True: return 'BEZ'
      if endswith(is) == False: return 'VBZ'
    if endswith(s) == False:
      if endswith(.) == True: return '.'
      if endswith(.) == False: return 'NN'
```

Consider the following code in NLTK for building a regular expression tagger. Here, tags are assigned on the basis of matching patterns:

```
>>> import nltk
>>> from nltk.corpus import brown
>>> sentences = brown.tagged_sents(categories='news')
>>> sent = brown.sents(categories='news')
>>> pattern = [
(r'.*ing$', 'VBG'),               # for gerunds
(r'.*ed$', 'VBD'),                # for simple past
```

```
    (r'.*es$', 'VBZ'),                # for 3rd singular present
    (r'.*ould$', 'MD'),               # for modals
    (r'.*\'s$', 'NN$'),               # for possessive nouns
    (r'.*s$', 'NNS'),                 # for plural nouns
    (r'^-?[0-9]+(.[0-9]+)?$', 'CD'),  #  for cardinal numbers
     (r'.*', 'NN')                     # for nouns (default)
]
>>> regexpr_tagger = nltk.RegexpTagger(pattern)
>>> regexpr_tagger.tag(sent[3])
 [('``', 'NN'), ('Only', 'NN'), ('a', 'NN'), ('relative', 'NN'),
('handful', 'NN'), ('of', 'NN'), ('such', 'NN'), ('reports', 'NNS'),
('was', 'NNS'), ('received', 'VBD'), ("''", 'NN'), (',', 'NN'),
('the', 'NN'), ('jury', 'NN'), ('said', 'NN'), (',', 'NN'), ('``',
'NN'), ('considering', 'VBG'), ('the', 'NN'), ('widespread', 'NN'),
('interest', 'NN'), ('in', 'NN'), ('the', 'NN'), ('election', 'NN'),
(',', 'NN'), ('the', 'NN'), ('number', 'NN'), ('of', 'NN'), ('voters',
'NNS'), ('and', 'NN'), ('the', 'NN'), ('size', 'NN'), ('of', 'NN'),
('this', 'NNS'), ('city', 'NN'), ("''", 'NN'), ('.', 'NN')]
>>> regexp_tagger.evaluate(sentences)
0.20326391789486245
```

Consider the following code to build a lookup tagger. In building up a lookup tagger, a list of frequently used words is maintained along with their tag information. Some of the words have been assigned the `None` tag because they do not exist among the list of the most frequently occurring words:

```
>>> import nltk
>>> from nltk.corpus import brown
>>> freqd = nltk.FreqDist(brown.words(categories='news'))
>>> cfreqd = nltk.ConditionalFreqDist(brown.tagged_
words(categories='news'))
>>> mostfreq_words = freqd.most_common(100)
>>> likelytags = dict((word, cfreqd[word].max()) for (word, _) in
mostfreq_words)
>>> baselinetagger = nltk.UnigramTagger(model=likelytags)
>>> baselinetagger.evaluate(brown_tagged_sents)
0.45578495136941344
>>> sent = brown.sents(categories='news')[3]
>>> baselinetagger.tag(sent)
[('``', '``'), ('Only', None), ('a', 'AT'), ('relative', None),
('handful', None), ('of', 'IN'), ('such', None), ('reports', None),
('was', 'BEDZ'), ('received', None), ("''", "''"), (',', ','),
('the', 'AT'), ('jury', None), ('said', 'VBD'), (',', ','),
```

```
('``', '``'), ('considering', None), ('the', 'AT'), ('widespread',
None),
('interest', None), ('in', 'IN'), ('the', 'AT'), ('election', None),
(',', ','), ('the', 'AT'), ('number', None), ('of', 'IN'),
('voters', None), ('and', 'CC'), ('the', 'AT'), ('size', None),
('of', 'IN'), ('this', 'DT'), ('city', None), ("''", "''"), ('.',
'.')]
>>> baselinetagger = nltk.UnigramTagger(model=likely_tags,
...                                     backoff=nltk.
DefaultTagger('NN'))
def performance(cfreqd, wordlist):
    lt = dict((word, cfreqd[word].max()) for word in wordlist)
    baseline_tagger = nltk.UnigramTagger(model=lt, backoff=nltk.
DefaultTagger('NN'))
    return baseline_tagger.evaluate(brown.tagged_
sents(categories='news'))

def display():
    import pylab
    word_freqs = nltk.FreqDist(brown.words(categories='news')).most_
common()
    words_by_freq = [w for (w, _) in word_freqs]
    cfd = nltk.ConditionalFreqDist(brown.tagged_
words(categories='news'))
    sizes = 2 ** pylab.arange(15)
    perfs = [performance(cfd, words_by_freq[:size]) for size in sizes]
    pylab.plot(sizes, perfs, '-bo')
    pylab.title('Lookup Tagger Performance with Varying Model Size')
    pylab.xlabel('Model Size')
    pylab.ylabel('Performance')
    pylab.show()
display()
```

Let's see the following stemming code in NLTK using `lancasterstemmer`. The evaluation of such a `stemmer` can be done using gold test data:

```
>>> import nltk
>>> from nltk.stem.lancaster import LancasterStemmer
>>> stri=LancasterStemmer()
>>> stri.stem('achievement')
'achiev'
```

Consider the following code in NLTK that can be used for designing a classifier-based `chunker`. It makes use of the Maximum Entropy classifier:

```
class ConseNPChunkTagger(nltk.TaggerI):

    def __init__(self, train_sents):
        train_set = []
        for tagsent in train_sents:
            untagsent = nltk.tag.untag(tagsent)
            history = []
            for i, (word, tag) in enumerate(tagsent):
                featureset = npchunk_features(untagsent, i, history)
                train_set.append( (featureset, tag) )
                history.append(tag)
        self.classifier = nltk.MaxentClassifier.train(
            train_set, algorithm='megam', trace=0)

    def tag(self, sentence):
        history = []
        for i, word in enumerate(sentence):
            featureset = npchunk_features(sentence, i, history)
            tag = self.classifier.classify(featureset)
            history.append(tag)
        return zip(sentence, history)

class ConseNPChunker(nltk.ChunkParserI): [4]
    def __init__(self, train_sents):
        tagsent = [[((w,t),c) for (w,t,c) in
                        nltk.chunk.tree2conlltags(sent)]
                       for sent in train_sents]
       self.tagger = ConseNPChunkTagger(tagsent)

    def parse(self, sentence):
        tagsent = self.tagger.tag(sentence)
        conlltags = [(w,t,c) for ((w,t),c) in tagsent]
        return nltk.chunk.conlltags2tree(conlltags)
```

In the following code, the evaluation of `chunker` is performed with the use of a feature extractor. The resultant `chunker` is similar to the unigram `chunker`:

```
>>> def npchunk_features(sentence, i, history):
...     word, pos = sentence[i]
...     return {"pos": pos}
>>> chunker = ConseNPChunker(train_sents)
```

```
>>> print(chunker.evaluate(test_sents))
ChunkParse score:
    IOB Accuracy:  92.9%
    Precision:     79.9%
    Recall:        86.7%
    F-Measure:     83.2%
```

In the following code, the features of the previous part of the speech tag are also added. This involves the interaction between tags. So the resultant `chunker` is similar to the bigram `chunker`:

```
>>> def npchunk_features(sentence, i, history):
...     word, pos = sentence[i]
...     if i == 0:
...         previword, previpos = "<START>", "<START>"
...     else:
...         previword, previpos = sentence[i-1]
...     return {"pos": pos, "previpos": previpos}
>>> chunker = ConseNPChunker(train_sents)
>>> print(chunker.evaluate(test_sents))
ChunkParse score:
    IOB Accuracy:  93.6%
    Precision:     81.9%
    Recall:        87.2%
    F-Measure:     84.5%
```

Consider the following code for `chunker` in which features for the current word are added to improve the performance of a `chunker`:

```
>>> def npchunk_features(sentence, i, history):
...     word, pos = sentence[i]
...     if i == 0:
...         previword, previpos = "<START>", "<START>"
...     else:
...         previword, previpos = sentence[i-1]
...     return {"pos": pos, "word": word, "previpos": previpos}
>>> chunker = ConseNPChunker(train_sents)
>>> print(chunker.evaluate(test_sents))
ChunkParse score:
    IOB Accuracy:  94.5%
    Precision:     84.2%
    Recall:        89.4%
    F-Measure:     86.7%
```

Let's consider the code in NLTK in which the collection of features, such as paired features, lookahead features, complex contextual features, and so on, are added to enhance the performance of a `chunker`:

```
>>> def npchunk_features(sentence, i, history):
...     word, pos = sentence[i]
...     if i == 0:
...         previword, previpos = "<START>", "<START>"
...     else:
...         previword, previpos = sentence[i-1]
...     if i == len(sentence)-1:
...         nextword, nextpos = "<END>", "<END>"
...     else:
...         nextword, nextpos = sentence[i+1]
...     return {"pos": pos,
...             "word": word,
...             "previpos": previpos,
...             "nextpos": nextpos,
...             "previpos+pos": "%s+%s" % (previpos, pos),
...             "pos+nextpos": "%s+%s" % (pos, nextpos),
...             "tags-since-dt": tags_since_dt(sentence, i)}
>>> def tags_since_dt(sentence, i):
...     tags = set()
...     for word, pos in sentence[:i]:
...         if pos == 'DT':
...             tags = set()
...         else:
...             tags.add(pos)
...     return '+'.join(sorted(tags))


>>> chunker = ConsecutiveNPChunker(train_sents)
>>> print(chunker.evaluate(test_sents))
ChunkParse score:
    IOB Accuracy:  96.0%
    Precision:     88.6%
    Recall:        91.0%
    F-Measure:     89.8%
```

The evaluation of Morphological Analyzer can also be performed using gold data. The human expected output is already stored to form a gold set and then the output of the morphological analyzer is compared with the gold data.

Parser evaluation using gold data

Parser evaluation can be done using the gold data or the standard data against which the output of the parser is matched.

Firstly, training of parser model is performed on the training data. Then parsing is done on the unseen data or testing data.

The following two measures can be used to evaluate the performance of a parser:

- **Labelled Attachment Score** (**LAS**)
- **Labelled Exact Match** (**LEM**)

In both cases, parser's output is compared with testing data. A good parsing algorithm is one that gives the highest LAS and LEM scores. The training and testing data that we use for parsing may consist of parts of speech tags that are gold standard tags, since they have been assigned manually. Parser evaluation can be done using metrics, such as Recall, Precision, and F-Measure.

Here, precision may be defined as the number of correct entities produced by parser divided by the total number of entities produced by parser.

Recall may be defined as the number of correct entities produced by parser divided by the total number of entities in the gold standard parse trees.

F-Score may be defined as the harmonic mean of recall and precision.

Evaluation of IR system

IR is also one of the applications of Natural Language Processing.

Following are the aspects that can be considered while performing the evaluation of the IR system:

- Resources required
- Presentation of documents
- Market evaluation or appealing to the user
- Retrieval speed
- Assistance in constituting queries
- Ability to find required documents

Evaluation is usually done by comparing one system with another.

IR systems can be compared on the basis of a set of documents, set of queries, techniques used, and so on. Metrics used for performance evaluation are Precision, Recall, and F-Measure. Let's learn a bit more about them:

- **Precision**: It is defined as the proportion of a retrieved set that is relevant.

  $Precision = |relevant \cap retrieved| \div |retrieved| = P(relevant \mid retrieved)$

- **Recall**: It is defined as the proportion of all the relevant documents in the collection included in the retrieved set.

  $Recall = |relevant \cap retrieved| \div |relevant| = P(retrieved \mid relevant)$

- **F-Measure**: It can be obtained using Precision and Recall as follows:

  *F-Measure = (2*Precision*Recall) / (Precision + Recall)*

Metrics for error identification

Error identification is a very important aspect that affects the performance of an NLP system. Searching tasks may involve the following terminologies:

- **True Positive** (**TP**): This may be defined as the set of relevant documents that is correctly identified as the relevant document.
- **True Negative** (**TN**): This may be defined as the set of irrelevant documents that is correctly identified as the irrelevant document.
- **False Positive** (**FP**): This is also referred to as Type I error and is the set of irrelevant documents that is incorrectly identified as the relevant document.
- **False Negative** (**FN**): This is also referred to as Type II error and is the set of relevant documents that is incorrectly identified as the irrelevant document.

On the basis of the previously mentioned terminologies, we have the following metrics:

- *Precision (P) - TP/(TP+FP)*
- *Recall (R) - TP/(TP+FN)*
- *F-Measure – 2*P*R/(P+R)*

Metrics based on lexical matching

We can also perform the analysis of performance at word level or lexical level.

Consider the following code in NLTK in which movie reviews have been taken and marked as either positive or negative. A feature extractor is constructed that checks whether a given word is present in a document or not:

```
>>> from nltk.corpus import movie_reviews
>>> docs = [(list(movie_reviews.words(fileid)), category)
...              for category in movie_reviews.categories()
...              for fileid in movie_reviews.fileids(category)]
>>> random.shuffle(docs)
all_wrds = nltk.FreqDist(w.lower() for w in movie_reviews.words())
word_features = list(all_wrds)[:2000]

def doc_features(doc):
    doc_words = set(doc)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in doc_words)
    return features
>>> print(doc_features(movie_reviews.words('pos/cv957_8737.txt')))
{'contains(waste)': False, 'contains(lot)': False, ...}
featuresets = [(doc_features(d), c) for (d,c) in docs]
train_set, test_set = featuresets[100:], featuresets[:100]
classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print(nltk.classify.accuracy(classifier, test_set))
0.81
>>> classifier.show_most_informative_features(5)
Most Informative Features
   contains(outstanding) = True              pos : neg    =     11.1 :
1.0
        contains(seagal) = True              neg : pos    =      7.7 :
1.0
   contains(wonderfully) = True              pos : neg    =      6.8 :
1.0
         contains(damon) = True              pos : neg    =      5.9 :
1.0
        contains(wasted) = True              neg : pos    =      5.8 :
1.0
```

Consider the following code in NLTK that describes `nltk.metrics.distance`, which provides metrics to determine whether a given output is the same as the expected output:

```
from __future__ import print_function
from __future__ import division
def _edit_dist_init(len1, len2):
    lev = []
    for i in range(len1):
        lev.append([0] * len2)  # initialization of 2D array to zero
    for i in range(len1):
        lev[i][0] = i           # column 0: 0,1,2,3,4,...
    for j in range(len2):
        lev[0][j] = j           # row 0: 0,1,2,3,4,...
    return lev

def _edit_dist_step(lev, i, j, s1, s2, transpositions=False):
    c1 = s1[i - 1]
    c2 = s2[j - 1]

    # skipping a character in s1
    a = lev[i - 1][j] + 1
    # skipping a character in s2
    b = lev[i][j - 1] + 1
    # substitution
    c = lev[i - 1][j - 1] + (c1 != c2)

    # transposition
    d = c + 1  # never picked by default
    if transpositions and i > 1 and j > 1:
        if s1[i - 2] == c2 and s2[j - 2] == c1:
            d = lev[i - 2][j - 2] + 1

    # pick the cheapest
    lev[i][j] = min(a, b, c, d)


def edit_distance(s1, s2, transpositions=False):

    # set up a 2-D array
    len1 = len(s1)
    len2 = len(s2)
    lev = _edit_dist_init(len1 + 1, len2 + 1)
```

```
    # iterate over the array
    for i in range(len1):
        for j in range(len2):
            _edit_dist_step(lev, i + 1, j + 1, s1, s2,
transpositions=transpositions)
    return lev[len1][len2]


def binary_distance(label1, label2):
    """Simple equality test.

    0.0 if the labels are identical, 1.0 if they are different.

>>> from nltk.metrics import binary_distance
>>> binary_distance(1,1)
    0.0

>>> binary_distance(1,3)
    1.0
    """

    return 0.0 if label1 == label2 else 1.0


def jaccard_distance(label1, label2):
    """Distance metric comparing set-similarity.
    """
    return (len(label1.union(label2)) - len(label1.
intersection(label2)))/len(label1.union(label2))

def masi_distance(label1, label2)

    len_intersection = len(label1.intersection(label2))
    len_union = len(label1.union(label2))
    len_label1 = len(label1)
    len_label2 = len(label2)
    if len_label1 == len_label2 and len_label1 == len_intersection:
        m = 1
    elif len_intersection == min(len_label1, len_label2):
        m = 0.67
    elif len_intersection > 0:
        m = 0.33
    else:
        m = 0
```

```
        return 1 - (len_intersection / len_union) * m


    def interval_distance(label1,label2):

        try:
            return pow(label1 - label2, 2)
    #        return pow(list(label1)[0]-list(label2)[0],2)
        except:
            print("non-numeric labels not supported with interval
    distance")


    def presence(label):

        return lambda x, y: 1.0 * ((label in x) == (label in y))


    def fractional_presence(label):
        return lambda x, y:\
            abs(((1.0 / len(x)) - (1.0 / len(y)))) * (label in x and label
    in y) \
            or 0.0 * (label not in x and label not in y) \
            or abs((1.0 / len(x))) * (label in x and label not in y) \
            or ((1.0 / len(y))) * (label not in x and label in y)


    def custom_distance(file):
        data = {}
        with open(file, 'r') as infile:
            for l in infile:
                labelA, labelB, dist = l.strip().split("\t")
                labelA = frozenset([labelA])
                labelB = frozenset([labelB])
                data[frozenset([labelA,labelB])] = float(dist)
        return lambda x,y:data[frozenset([x,y])]


    def demo():
        edit_distance_examples = [
            ("rain", "shine"), ("abcdef", "acbdef"), ("language",
    "lnaguaeg"),
            ("language", "lnaugage"), ("language", "lngauage")]
        for s1, s2 in edit_distance_examples:
```

```
        print("Edit distance between '%s' and '%s':" % (s1, s2), edit_
distance(s1, s2))
    for s1, s2 in edit_distance_examples:
        print("Edit distance with transpositions between '%s' and
'%s':" % (s1, s2), edit_distance(s1, s2, transpositions=True))

    s1 = set([1, 2, 3, 4])
    s2 = set([3, 4, 5])
    print("s1:", s1)
    print("s2:", s2)
    print("Binary distance:", binary_distance(s1, s2))
    print("Jaccard distance:", jaccard_distance(s1, s2))
    print("MASI distance:", masi_distance(s1, s2))

if __name__ == '__main__':
    demo()
```

Metrics based on syntactic matching

Syntactic matching can be done by performing the task of chunking. In NLTK, a module called `nltk.chunk.api` is provided that helps to identify chunks and returns a parse tree for a given chunk sequence.

The module called `nltk.chunk.named_entity` is used to identify a list of named entities and also to generate a parse structure. Consider the following code in NLTK based on syntactic matching:

```
>>> import nltk
>>> from nltk.tree import Tree
>>> print(Tree(1,[2,Tree(3,[4]),5]))
(1 2 (3 4) 5)
>>> ct=Tree('VP',[Tree('V',['gave']),Tree('NP',['her'])])
>>> sent=Tree('S',[Tree('NP',['I']),ct])
>>> print(sent)
(S (NP I) (VP (V gave) (NP her)))
>>> print(sent[1])
(VP (V gave) (NP her))
>>> print(sent[1,1])
(NP her)
>>> t1=Tree.from_string("(S(NP I) (VP (V gave) (NP her)))")
>>> sent==t1
True
>>> t1[1][1].set_label('X')
>>> t1[1][1].label()
```

```
'X'
>>> print(t1)
(S (NP I) (VP (V gave) (X her)))
>>> t1[0],t1[1,1]=t1[1,1],t1[0]
>>> print(t1)
(S (X her) (VP (V gave) (NP I)))
>>> len(t1)
2
```

Metrics using shallow semantic matching

WordNet Similarity is used to perform semantic matching. In this, a similarity of a given text is computed against the hypothesis. The Natural Language Toolkit can be used to compute: path distance, Leacock-Chodorow Similarity, Wu-Palmer Similarity, Resnik Similarity, Jiang-Conrath Similarity, and Lin Similarity between words present in the text and the hypothesis. In these metrics, we compare the similarity between word senses rather than words.

During Shallow Semantic analysis, NER and coreference resolution are also performed.

Consider the following code in NLTK that computes `wordnet` similarity:

```
>>> wordnet.N['dog'][0].path_similarity(wordnet.N['cat'][0])
0.20000000000000001
>>> wordnet.V['run'][0].path_similarity(wordnet.V['walk'][0])
0.25
```

Summary

In this chapter, we discussed the evaluation of NLP systems (POS tagger, stemmer, and morphological analyzer). You learned about various metrics used for performing the evaluation of NLP systems based on error identification, lexical matching, syntactic matching, and shallow semantic matching. We also discussed parser evaluation performed using gold data. Evaluation can be done using three metrics, namely Precision, Recall, and F-Measure. You also learned about the evaluation of IR system.

Index

P

Q

R

S

T

U

V

W

Z