Natural Language Processing Recipes

Unlocking Text Data with Machine Learning and Deep Learning using Python

Akshay Kulkarni
Adarsha Shivananda

Apress®

Natural Language Processing Recipes

Unlocking Text Data with Machine Learning and Deep Learning using Python

Akshay Kulkarni
Adarsha Shivananda

Apress®

Natural Language Processing Recipes

Akshay Kulkarni
Bangalore, Karnataka, India

Adarsha Shivananda
Bangalore, Karnataka, India

ISBN-13 (pbk): 978-1-4842-4266-7
ISBN-13 (electronic): 978-1-4842-4267-4
https://doi.org/10.1007/978-1-4842-4267-4

Library of Congress Control Number: 2019931849

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Development Editor: Matthew Moodie
Coordinating Editor: Shrikant Vishwakarma

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4266-7. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To our family

Table of Contents

About the Authors

Akshay Kulkarni is an Artificial Intelligence and Machine learning evangelist. Akshay has a rich experience of building and scaling AI and Machine Learning businesses and creating significant client impact. He is currently the Senior Data Scientist at SapientRazorfish's core data science team where he is part of strategy and transformation interventions through AI and works on various Machine Learning, Deep Learning, and Artificial Intelligence engagements by applying state-of-the-art techniques in this space. Previously he was part of Gartner and Accenture, where he scaled the analytics and data science business.

Akshay is a regular speaker at major data science conferences. He is a visiting faculty at few of the top graduate institutes in India. In his spare time, he enjoys reading, writing, coding, and helping aspiring data scientists. He lives in Bangalore with his family.

Adarsha Shivananda is a Senior Data Scientist at Indegene's Product and Technology team where he is working on building Machine Learning and AI capabilities to pharma products. He is aiming to build a pool of exceptional data scientists within and outside of the organization to solve greater problems through brilliant training programs; and

he always wants to stay ahead of the curve. Previously he was working with Tredence Analytics and IQVIA. Adarsha has extensively worked on pharma, health care, retail, and marketing domains.

He lives in Bangalore and loves to read, ride, and teach data science.

About the Technical Reviewers

Dikshant Shahi is a Software Architect with expertise in Search Engines, Semantic Technologies, and Natural Language Processing. He is currently focusing on building semantic search platforms and related enterprise applications. He has been building search engines for more than a decade and is also the author of the book *Apache Solr: A Practical Approach to Enterprise Search* (Apress, 2015).

Dikshant lives in Bangalore, India. When not at work, you can find him backpacking.

Krishnendu Dasgupta is a Senior Consultant with 8 years of experience. He has worked on different cloud platforms and has designed data mining architectures. He is working and contributing toward NLP and Artificial Intelligence through his work. He has worked with major consulting firms and has experience in supply chain and banking domains.

Krishnendu is accredited by the Global Innovation and Entrepreneurship Bootcamp – Class of 2018, held by the Massachusetts Institute of Technology.

Acknowledgments

We are grateful to our mother, father, and loving brother and sister. We thank all of them for their motivation and constant support.

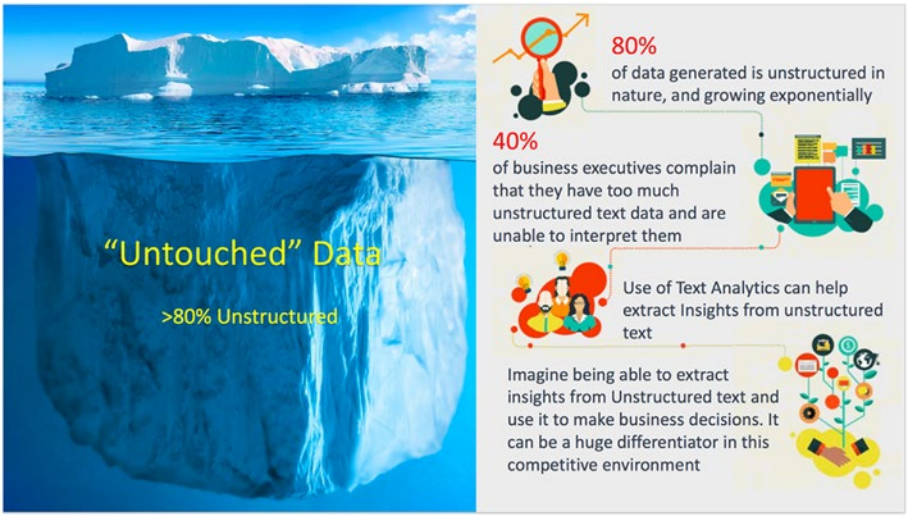
We would like to express our gratitude to mentors and friends for their inputs, inspiration, and support. A special thanks to Anoosh R. Kulkarni, Data Scientist at Awok.com for all his support in writing this book and his technical inputs. Big thanks to the Apress team for their constant support and help.

Finally, we would like to thank you, the reader, for showing an interest in this book and believe that you can make your natural language processing journey more interesting and exciting.

Note that the views expressed in this book are the authors' personal ones.

Introduction

According to industry estimates, more than 80% of the data being generated is in an unstructured format, maybe in the form of text, image, audio, video, etc. Data is getting generated as we speak, as we write, as we tweet, as we use social media platforms, as we send messages on various messaging platforms, as we use e-commerce for shopping and in various other activities. The majority of this data exists in the textual form.



So, what is unstructured data? Unstructured data is the information that doesn't reside in a traditional relational database. Examples include documents, blogs, social media feeds, pictures, and videos.

Most of the insight is locked within different types of unstructured data. Unlocking all these unstructured data plays a vital role in every organization to make improved and better decisions. In this book, let us unlock the potential of text data.

Text data is most common and covers more than 50% of the unstructured data. A few examples include – tweets/posts on social media, chat conversations, news, blogs and articles, product or services reviews, and patient records in the health care sector. A few more recent ones include voice-driven bots like Siri, Alexa, etc.

In order to produce significant and actionable insights from text data, to unlock the potential of text data, we use Natural Language Processing coupled with machine learning and deep learning.

But what is Natural Language Processing - popularly known as NLP? We all know that machines/algorithms cannot understand texts or characters, so it is very important to convert these text data into machine understandable format (like numbers or binary) to perform any kind of analysis on text data. The ability to make machines understand and interpret the human language (text data) is termed as natural language processing.

So, if you want to use the power of unstructured text, this book is the right starting point. This book unearths the concepts and implementation of natural language processing and its applications in the real world. Natural Language Processing (NLP) offers unbounded opportunities for solving interesting problems in artificial intelligence, making it the latest frontier for developing intelligent, deep learning-based applications.

What This Book Covers

Natural Language Processing Recipes is your handy problem-solution reference for learning and implementing NLP solutions using Python. The book is packed with thousands of code and approaches that help you to quickly learn and implement the basic and advanced Natural Language Processing techniques. You will learn how to efficiently use a wide range of NLP packages and implement text classification, identify parts of speech, topic modeling, text summarization, text generation, sentiment analysis, and many more applications of NLP.

能学到什么：

This book starts off by ways of extracting text data along with web scraping. You will also learn how to clean and preprocess text data and ways to analyze them with advanced algorithms. During the course of the book, you will explore the semantic as well as syntactic analysis of the text. We will be covering complex NLP solutions that will involve text normalization, various advanced preprocessing methods, POS tagging, text similarity, text summarization, sentiment analysis, topic modeling, NER, word2vec, seq2seq, and much more. In this book, we will cover the various fundamentals necessary for applications of machine learning and deep learning in natural language processing, and the other state-of-the-art techniques. Finally, we close it with some of the advanced industrial applications of NLP with the solution approach and implementation, also leveraging the power of deep learning techniques for Natural Language Processing and Natural Language Generation problems. Employing state-of-the-art advanced RNNs, like long short-term memory, to solve complex text generation tasks. Also, we explore word embeddings.

Each chapter includes several code examples and illustrations.

By the end of the book, the reader will have a clear understanding of implementing natural language processing and will have worked on multiple examples that implement NLP techniques in the real world. The reader will be comfortable with various NLP techniques coupled with machine learning and deep learning and its industrial applications, which make the NLP journey much more interesting and will definitely help improve Python coding skills as well. You will learn about all the ingredients that you need to, to become successful in the NLP space.

Who This Book Is For

Fundamental Python skills are assumed, as well as some knowledge of machine learning. If you are an NLP or machine learning enthusiast and an intermediate Python programmer who wants to quickly master natural

language processing, then this learning path will do you a lot of good. All you need are the basics of machine learning and Python to enjoy this book.

What you will learn:

1) Core concepts implementation of NLP and various approaches to natural language processing, NLP using Python libraries such as NLTK, TextBlob, SpaCy, Stanford CoreNLP, and so on.

2) Learn about implementing text preprocessing and feature engineering in NLP, along with advanced methods of feature engineering like word embeddings.

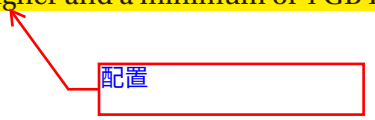
3) Understand and implement the concepts of information retrieval, text summarization, sentiment analysis, text classification, text generation, and other advanced NLP techniques solved by leveraging machine learning and deep learning.

4) After reading this book, the reader should get a good hold of the problems faced by different industries and how to implement them using NLP techniques.

5) Implementing an end-to-end pipeline of the NLP life cycle, which includes framing the problem, finding the data, collecting, preprocessing the data, and solving it using state-of-the-art techniques.

What You Need For This Book

To perform all the recipes of this book successfully, you will need Python 3.x or higher running on any Windows- or Unix-based operating system with a processor of 2.0 GHz or higher and a minimum of 4 GB RAM. You



can download Python from Anaconda and leverage Jupyter notebook for all coding purposes. This book assumes you know Keras's basics and how to install the basic libraries of machine learning and deep learning.

Please make sure you upgrade or install the latest version of all the libraries.

Python is the most popular and widely used tool for building NLP applications. It has a huge number of sophisticated libraries to perform NLP tasks starting from basic preprocessing to advanced techniques.

To install any library in Python Jupyter notebook. use "!" before the pip install.

NLTK: Natural language toolkit and commonly called the mother of all NLP libraries. It is one of the mature primary resources when it comes to Python and NLP.

```
!pip install nltk
nltk.download()
```

SpaCy: SpaCy is recently a trending library, as it comes with the added flavors of a deep learning framework. While SpaCy doesn't cover all of the NLP functionalities, the things that it does do, it does really well.

```
!pip install spacy
#if above doesn't work, try this in your terminal/ command
prompt
conda install spacy
python -m spacy.en.download all
#then load model via
spacy.load('en')
```

TextBlob: This is one of the data scientist's favorite library when it comes to implementing NLP tasks. It is based on both NLTK and Pattern. However, TextBlob certainly isn't the fastest or most complete library.

```
!pip install textblob
```

CoreNLP: It is a Python wrapper for Stanford CoreNLP. The toolkit provides very robust, accurate, and optimized techniques for tagging, parsing, and analyzing text in various languages.

```
!pip install CoreNLP
```

These are not the only ones; there are hundreds of NLP libraries. But we have covered widely used and important ones.

Motivation: There is an immense number of industrial applications of NLP that are leveraged to uncover insights. By the end of the book, you will have implemented most of these use cases end to end, right from framing the business problem to building applications and drawing business insights.

- Sentiment analysis: Customer's emotions toward products offered by the business.
- Topic modeling: Extract the unique topics from the group of documents.
- Complaint classifications/Email classifications/ E-commerce product classification, etc.
- Document categorization/management using different clustering techniques.
- Resume shortlisting and job description matching using similarity methods.
- Advanced feature engineering techniques (word2vec and fastText) to capture context.
- Information/Document Retrieval Systems, for example, search engine.
- Chatbot, Q & A, and Voice-to-Text applications like Siri and Alexa.

- Language detection and translation using neural networks.
- Text summarization using graph methods and advanced techniques.
- Text generation/predicting the next sequence of words using deep learning algorithms.

CHAPTER 1

Extracting the Data

In this chapter, we are going to cover various sources of text data and ways to extract it, which can act as information or insights for businesses.

Recipe 1. Text data collection using APIs

Recipe 2. Reading PDF file in Python

Recipe 3. Reading word document

Recipe 4. Reading JSON object

Recipe 5. Reading HTML page and HTML parsing

Recipe 6. Regular expressions

Recipe 7. String handling

Recipe 8. Web scraping

Introduction

Before getting into details of the book, let's see the different possible data sources available in general. We need to identify potential data sources for a business's benefit.

A. Kulkarni and A. Shivananda, *Natural Language Processing Recipes*,
https://doi.org/10.1007/978-1-4842-4267-4_1

Client Data For any problem statement, one of the sources is their own data that is already present. But it depends on the business where they store it. Data storage depends on the type of business, amount of data, and cost associated with different sources.

- SQL databases
- Hadoop clusters
- Cloud storage
- Flat files

Free source A huge amount of data is freely available over the internet. We just need to streamline the problem and start exploring multiple free data sources.

- Free APIs like Twitter
- Wikipedia
- Government data (e.g. `http://data.gov`)
- Census data (e.g. `http://www.census.gov/data.html`)
- Health care claim data (e.g. `https://www.healthdata.gov/`)

Web scraping Extracting the content/data from websites, blogs, forums, and retail websites for reviews with the permission from the respective sources using web scraping packages in Python.

There are a lot of other sources like crime data, accident data, and economic data that can also be leveraged for analysis based on the problem statement.

Recipe 1-1. Collecting Data

As discussed, there are a lot of free APIs through which we can collect data and use it to solve problems. We will discuss the Twitter API in particular (it can be used in other scenarios as well).

Problem

You want to collect text data using Twitter APIs.

Solution

Twitter has a gigantic amount of data with a lot of value in it. Social media marketers are making their living from it. There is an enormous amount of tweets every day, and every tweet has some story to tell. When all of this data is collected and analyzed, it gives a tremendous amount of insights to a business about their company, product, service, etc.

Let's see how to pull the data in this recipe and then explore how to leverage it in coming chapters.

How It Works

Step 1-1 Log in to the Twitter developer portal

Create your own app in the Twitter developer portal, and get the keys mentioned below. Once you have these credentials, you can start pulling data. Keys needed:

- consumer key: Key associated with the application (Twitter, Facebook, etc.).
- consumer secret: Password used to authenticate with the authentication server (Twitter, Facebook, etc.).

- access token: Key given to the client after successful authentication of above keys.
- access token secret: Password for the access key.

Step 1-2 Execute below query in Python

Once all the credentials are in place, use the code below to fetch the data.

```
# Install tweepy
!pip install tweepy

# Import the libraries

import numpy as np
import tweepy
import json
import pandas as pd
from tweepy import OAuthHandler

# credentials

consumer_key = "adjbiejfaaoeh"
consumer_secret = "had73haf78af"
access_token = "jnsfby5u4yuawhafjeh"
access_token_secret = "jhdfgay768476r"

# calling API

auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)
api = tweepy.API(auth)

# Provide the query you want to pull the data. For example,
pulling data for the mobile phone ABC

query ="ABC"
```

```
# Fetching tweets

Tweets = api.search(query, count = 10,lang='en',
exclude='retweets',tweet_mode='extended')
```

The query above will pull the top 10 tweets when the product ABC is searched. The API will pull English tweets since the language given is 'en' and it will exclude retweets.

Recipe 1-2. Collecting Data from PDFs

Most of the time your data will be stored as PDF files. We need to extract text from these files and store it for further analysis.

Problem

You want to read a PDF file.

Solution

The simplest way to do this is by using the PyPDF2 library.

How It Works

Let's follow the steps in this section to extract data from PDF files.

Step 2-1 Install and import all the necessary libraries

Here are the first lines of code:

```
!pip install PyPDF2
import PyPDF2
from PyPDF2 import PdfFileReader
```

Note You can download any PDF file from the web and place it in the location where you are running this Jupyter notebook or Python script.

Step 2-2 Extracting text from PDF file

Now we extract the text.

```
#Creating a pdf file object

pdf = open("file.pdf","rb")

#creating pdf reader object

pdf_reader = PyPDF2.PdfFileReader(pdf)

#checking number of pages in a pdf file

print(pdf_reader.numPages)

#creating a page object

page = pdf_reader.getPage(0)

#finally extracting text from the page

print(page.extractText())

#closing the pdf file

pdf.close()
```

Please note that the function above doesn't work for scanned PDFs.

Recipe 1-3. Collecting Data from Word Files

Next, let us look at another small recipe by reading **Word** files in Python.

Problem

You want to read Word files.

Solution

The simplest way to do this is by using the docx library.

How It Works

Let's follow the steps in this section to extract data from the Word file.

Step 3-1 Install and import all the necessary libraries

Here are the first lines of code:

```
#Install docx
!pip install docx

#Import library
from docx import Document
```

Note You can download any Word file from the web and place it in the location where you are running this Jupyter notebook or Python script.

Step 3-2 Extracting text from word file

Now we get the text:

```
#Creating a word file object

doc = open("file.docx","rb")

#creating word reader object

document = docx.Document(doc)

# create an empty string and call this document. This document
variable store each paragraph in the Word document.We then
create a for loop that goes through each paragraph in the Word
document and appends the paragraph.

docu=""
for para in document.paragraphs:
       docu += para.text

#to see the output call docu
print(docu)
```

Recipe 1-4. Collecting Data from JSON

Reading a JSON file/object.

Problem

You want to read a JSON file/object.

Solution

The simplest way to do this is by using requests and the JSON library.

How It Works

Let's follow the steps in this section to extract data from the JSON.

Step 4-1 Install and import all the necessary libraries

Here is the code for importing the libraries.

```
import requests
import json
```

Step 4-2 Extracting text from JSON file

Now we extract the text.

```
#json from "https://quotes.rest/qod.json"
r = requests.get("https://quotes.rest/qod.json")
res = r.json()
print(json.dumps(res, indent = 4))
```

```
#output
{
    "success": {
        "total": 1
    },
    "contents": {
        "quotes": [
            {
                "quote": "Where there is ruin, there is hope
                         for a treasure.",
                "length": "50",
                "author": "Rumi",
                "tags": [
                    "failure",
```

```
                    "inspire",
                    "learning-from-failure"
                ],
                "category": "inspire",
                "date": "2018-09-29",
                "permalink": "https://theysaidso.com/quote/
                dPKsui4sQnQqgMnXHLKtfweF/rumi-where-there-is-
                ruin-there-is-hope-for-a-treasure",
                "title": "Inspiring Quote of the day",
                "background": "https://theysaidso.com/img/bgs/
                man_on_the_mountain.jpg",
                "id": "dPKsui4sQnQqgMnXHLKtfweF"
            }
        ],
        "copyright": "2017-19 theysaidso.com"
    }
}

#extract contents
q = res['contents']['quotes'][0]
q

#output

{'author': 'Rumi',
 'background': 'https://theysaidso.com/img/bgs/man_on_the_
 mountain.jpg',
 'category': 'inspire',
 'date': '2018-09-29',
 'id': 'dPKsui4sQnQqgMnXHLKtfweF',
 'length': '50',
```

```
 'permalink': 'https://theysaidso.com/quote/
 dPKsui4sQnQqgMnXHLKtfweF/rumi-where-there-is-ruin-there-is-
 hope-for-a-treasure',
 'quote': 'Where there is ruin, there is hope for a treasure.',
 'tags': ['failure', 'inspire', 'learning-from-failure'],
 'title': 'Inspiring Quote of the day'}

#extract only quote
print(q['quote'], '\n--', q['author'])

#output
It wasn't raining when Noah built the ark....
-- Howard Ruff
```

Recipe 1-5. Collecting Data from HTML

In this recipe, let us look at reading HTML pages.

Problem

You want to read parse/read HTML pages.

Solution

The simplest way to do this is by using the bs4 library.

How It Works

Let's follow the steps in this section to extract data from the web.

Step 5-1 Install and import all the necessary libraries

Let's import the libraries:

```
!pip install bs4
import urllib.request as urllib2
from bs4 import BeautifulSoup
```

Step 5-2 Fetch the HTML file

Pick any website from the web that you want to extract. Let's pick Wikipedia for this example.

```
response = urllib2.urlopen('https://en.wikipedia.org/wiki/
Natural_language_processing')
html_doc = response.read()
```

Step 5-3 Parse the HTML file

Now we get the data:

```
#Parsing
soup = BeautifulSoup(html_doc, 'html.parser')
# Formating the parsed html file
strhtm = soup.prettify()

# Print few lines
print (strhtm[:1000])

#output

<!DOCTYPE html>
<html class="client-nojs" dir="ltr" lang="en">
 <head>
  <meta charset="utf-8"/>
  <title>
```

```
 Natural language processing - Wikipedia
</title>
<script>
 document.documentElement.className = document.
 documentElement.className.replace( /(^|\s)client-nojs(\
 s|$)/, "$1client-js$2" );
</script>
<script>
 (window.RLQ=window.RLQ||[]).push(function(){mw.config.
 set({"wgCanonicalNamespace":"","wgCanonicalSpecialPageName":
 false,"wgNamespaceNumber":0,"wgPageName":"Natural_language_
 processing","wgTitle":"Natural language processing",
 "wgCurRevisionId":860741853,"wgRevisionId":860741853,"wgArticle
 Id":21652,"wgIsArticle":true,"wgIsRedirect":false,"wgAction":
 "view","wgUserName":null,"wgUserGroups":["*"],"wgCategories":
 ["Webarchive template wayback links","All accuracy disputes",
 "Articles with disputed statements from June 2018",
 "Wikipedia articles with NDL identifiers","Natural language
 processing","Computational linguistics","Speech recognition",
 "Computational fields of stud
```

Step 5-4 Extracting tag value

We can extract a tag value from the first instance of the tag using the following code.

```
print(soup.title)
print(soup.title.string)
print(soup.a.string)
print(soup.b.string)
```

```
#output
 <title>Natural language processing - Wikipedia</title>
Natural language processing - Wikipedia
None
Natural language processing
```

Step 5-5 Extracting all instances of a particular tag

Here we get all the instances of a tag that we are interested in:

```
for x in soup.find_all('a'): print(x.string)
```

```
#sample output
 None
Jump to navigation
Jump to search
Language processing in the brain
None
None
automated online assistant
customer service
[1]
computer science
artificial intelligence
natural language
speech recognition
natural language understanding
natural language generation
```

Step 5-6 Extracting all text of a particular tag

Finally, we get the text:

```
for x in soup.find_all('p'): print(x.text)
```

```
#sample output
Natural language processing (NLP) is an area of computer
science and artificial intelligence concerned with the
interactions between computers and human (natural) languages,
in particular how to program computers to process and analyze
large amounts of natural language data.

Challenges in natural language processing frequently involve
speech recognition, natural language understanding, and natural
language generation.

The history of natural language processing generally started in
the 1950s, although work can be found from earlier periods.
In 1950, Alan Turing published an article titled "Intelligence"
which proposed what is now called the Turing test as a
criterion of intelligence.
```

If you observe here, using the 'p' tag extracted most of the text present in the page.

Recipe 1-6. Parsing Text Using Regular Expressions

In this recipe, we are going to discuss how regular expressions are helpful when dealing with text data. This is very much required when dealing with raw data from the web, which would contain HTML tags, long text, and repeated text. During the process of developing your application, as well as in output, we don't need such data.

We can do all sort of basic and advanced data cleaning using regular expressions.

Problem

You want to parse text data using regular expressions.

Solution

The best way to do this is by using the "re" library in Python.

How It Works

Let's look at some of the ways we can use regular expressions for our tasks. Basic flags: the basic flags are I, L, M, S, U, X:

- `re.I`: This flag is used for ignoring casing.
- `re.L`: This flag is used to find a local dependent.
- `re.M`: This flag is useful if you want to find patterns throughout multiple lines.
- `re.S`: This flag is used to find dot matches.
- `re.U`: This flag is used to work for unicode data.
- `re.X`: This flag is used for writing regex in a more readable format.

Regular expressions' functionality:

- Find the single occurrence of character a and b:

  ```
  Regex: [ab]
  ```

- Find characters except for a and b:

  ```
  Regex: [^ab]
  ```

- Find the character range of a to z:

  ```
  Regex: [a-z]
  ```

- Find a range except to z:

  ```
  Regex: [^a-z]
  ```

- Find all the characters a to z as well as A to Z:

  ```
  Regex: [a-zA-Z]
  ```

- Any single character:

  ```
  Regex:
  ```

- Any whitespace character:

  ```
  Regex: \s
  ```

- Any non-whitespace character:

  ```
  Regex: \S
  ```

- Any digit:

  ```
  Regex: \d
  ```

- Any non-digit:

  ```
  Regex: \D
  ```

- Any non-words:

  ```
  Regex: \W
  ```

- Any words:

  ```
  Regex: \w
  ```

- Either match a or b:

  ```
  Regex: (a|b)
  ```

- The occurrence of a is either zero or one:
  - Matches zero or one occurrence but not more than one occurrence

    ```
    Regex: a? ; ?
    ```

  - The occurrence of a is zero times or more than that:

    ```
    Regex: a* ; * matches zero or more than that
    ```

  - The occurrence of a is one time or more than that:

    ```
    Regex: a+ ; + matches occurrences one or more that
    one time
    ```

- Exactly match three occurrences of a:

  ```
  Regex: a{3}
  ```

- Match simultaneous occurrences of a with 3 or more than 3:

  ```
  Regex: a{3,}
  ```

- Match simultaneous occurrences of a between 3 to 6:

  ```
  Regex: a{3,6}
  ```

- Starting of the string:

  ```
  Regex: ^
  ```

- Ending of the string:

  ```
  Regex: $
  ```

- Match word boundary:

  `Regex: \b`

- Non-word boundary:

  `Regex: \B`

`re.match()` and `re.search()` functions are used to find the patterns and then can be processed according to the requirements of the application.

Let's look at the differences between `re.match()` and `re.search()`:

- `re.match()`: This checks for a match of the string only at the beginning of the string. So, if it finds the pattern at the beginning of the input string, then it returns the matched pattern; otherwise; it returns a noun.
- `re.search()`: This checks for a match of the string anywhere in the string. It finds all the occurrences of the pattern in the given input string or data.

Now let's look at a few of the examples using these regular expressions.

Tokenizing

You want to split the sentence into words – tokenize. One of the ways to do this is by using `re.split`.

```
# Import library

import re

#run the split query

re.split('\s+','I like this book.')

['I', 'like', 'this', 'book.']
```

For an explanation of regex, please refer to the main recipe.

Extracing email IDs

The simplest way to do this is by using `re.findall.`

1. Read/create the document or sentences

```
doc = "For more details please mail us at: xyz@abc.com,
pqr@mno.com"
```

2. Execute the re.findall function

```
addresses = re.findall(r'[\w\.-]+@[\w\.-]+', doc)
for address in addresses:
    print(address)

#Output
xyz@abc.com
pqr@mno.com
```

Replacing email IDs

Here we replace email ids from the sentences or documents with another email id. The simplest way to do this is by using `re.sub`.

1. Read/create the document or sentences

```
doc = "For more details please mail us at xyz@abc.com"
```

2. Execute the `re.sub` function

```
new_email_address = re.sub(r'([\w\.-]+)@([\w\.-]+)',
r'pqr@mno.com', doc)
print(new_email_address)

#Output
For more details please mail us at pqr@mno.com
```

For an explanation of regex, please refer to Recipe 1-6.

Extract data from the ebook and perform regex

Let's solve this case study by using the techniques learned so far.

1. Extract the content from the book

```
# Import library

import re
import requests

#url you want to extract
url = 'https://www.gutenberg.org/files/2638/2638-0.txt'

#function to extract
def get_book(url):
 # Sends a http request to get the text from project
Gutenberg
 raw = requests.get(url).text
 # Discards the metadata from the beginning of the book
 start = re.search(r"\*\*\* START OF THIS PROJECT
 GUTENBERG EBOOK .* \*\*\*",raw ).end()
 # Discards the metadata from the end of the book
 stop = re.search(r"II", raw).start()
 # Keeps the relevant text
 text = raw[start:stop]
 return text

# processing
def preprocess(sentence):
 return re.sub('[^A-Za-z0-9.]+' , ' ', sentence).lower()

#calling the above function

book = get_book(url)
processed_book = preprocess(book)
print(processed_book)
```

```
# Output
 produced by martin adamson david widger with
 corrections by andrew sly the idiot by fyodor
 dostoyevsky translated by eva martin part i i. towards
 the end of november during a thaw at nine o clock one
 morning a train on the warsaw and petersburg railway
 was approaching the latter city at full speed. the
 morning was so damp and misty that it was only with
 great difficulty that the day succeeded in breaking
 and it was impossible to distinguish anything more
 than a few yards away from the carriage windows.
 some of the passengers by this particular train were
 returning from abroad but the third class carriages
 were the best filled chiefly with insignificant
 persons of various occupations and degrees picked up
 at the different stations nearer town. all of them
 seemed weary and most of them had sleepy eyes and a
 shivering expression while their complexions generally
 appeared to have taken on the colour of the fog
 outside. when da
```

2. Perform some exploratory data analysis on this data using regex

```
# Count number of times "the" is appeared in the book
len(re.findall(r'the', processed_book))
```

```
#Output
302
```

```
#Replace "i" with "I"
processed_book = re.sub(r'\si\s', " I ", processed_book)
print(processed_book)
```

```
#output
 produced by martin adamson david widger with
 corrections by andrew sly the idiot by fyodor
 dostoyevsky translated by eva martin part I i. towards
 the end of november during a thaw at nine o clock one
 morning a train on the warsaw and petersburg railway
 was approaching the latter city at full speed. the
 morning was so damp and misty that it was only with
 great difficulty that the day succeeded in breaking
 and it was impossible to distinguish anything more
 than a few yards away from the carriage windows.
 some of the passengers by this particular train were
 returning from abroad but the third class carriages
 were the best filled chiefly with insignificant
 persons of various occupations and degrees picked up
 at the different stations nearer town. all of them
 seemed weary and most of them had sleepy eyes and a
 shivering expression while their complexions generally
 appeared to have taken on the colour of the fog
 outside. when da

#find all occurance of text in the format "abc--xyz"
re.findall(r'[a-zA-Z0-9]*--[a-zA-Z0-9]*', book)

#output
 ['ironical--it',
 'malicious--smile',
 'fur--or',
 'astrachan--overcoat',
 'it--the',
 'Italy--was',
 'malady--a',
```

```
'money--and',
'little--to',
'No--Mr',
'is--where',
'I--I',
'I--',
'--though',
'crime--we',
'or--judge',
'gaiters--still',
'--if',
'through--well',
'say--through',
'however--and',
'Epanchin--oh',
'too--at',
'was--and',
'Andreevitch--that',
'everyone--that',
'reduce--or',
'raise--to',
'listen--and',
'history--but',
'individual--one',
'yes--I',
'but--',
't--not',
'me--then',
'perhaps--',
'Yes--those',
'me--is',
```

```
'servility--if',
'Rogojin--hereditary',
'citizen--who',
'least--goodness',
'memory--but',
'latter--since',
'Rogojin--hung',
'him--I',
'anything--she',
'old--and',
'you--scarecrow',
'certainly--certainly',
'father--I',
'Barashkoff--I',
'see--and',
'everything--Lebedeff',
'about--he',
'now--I',
'Lihachof--',
'Zaleshoff--looking',
'old--fifty',
'so--and',
'this--do',
'day--not',
'that--',
'do--by',
'know--my',
'illness--I',
'well--here',
'fellow--you']
```

Recipe 1-7. Handling Strings

In this recipe, we are going to discuss how to handle strings and dealing with text data.

We can do all sort of basic text explorations using string operations.

Problem

You want to explore handling strings.

Solution

The simplest way to do this is by using the below string functionality.

`s.find(t)` index of first instance of string t inside s (-1 if not found)
`s.rfind(t)` index of last instance of string t inside s (-1 if not found)
`s.index(t)` like s.find(t) except it raises ValueError if not found
`s.rindex(t)` like s.rfind(t) except it raises ValueError if not found
`s.join(text)` combine the words of the text into a string using s as the glue
`s.split(t)` split s into a list wherever a t is found (whitespace by default)
`s.splitlines()` split s into a list of strings, one per line
`s.lower()` a lowercased version of the string s
`s.upper()` an uppercased version of the string s
`s.title()` a titlecased version of the string s
`s.strip()` a copy of s without leading or trailing whitespace
`s.replace(t, u)` replace instances of t with u inside s

How It Works

Now let us look at a few of the examples.

Replacing content

Create a string and replace the content. Creating Strings is easy, and it is done by enclosing the characters in single or double quotes. And to replace, you can use the `replace` function.

1. Creating a string

```
String_v1 = "I am exploring NLP"

#To extract particular character or range of characters
from string

print(String_v1[0])

#output
"I"

#To extract exploring

print(String_v1[5:14])

#output
exploring
```

2. Replace “exploring” with “learning” in the above string

```
String_v2 = String_v1.replace("exploring", "learning")
print(String_v2)

#Output
I am learning NLP
```

Concatenating two strings

Here's the simple code:

```
s1 = "nlp"
s2 = "machine learning"
s3 = s1+s2
print(s3)

#output
'nlpmachine learning'
```

Searching for a substring in a string

Use the `find` function to fetch the starting index value of the substring in the whole string.

```
var="I am learning NLP"
f= "learn"
var.find(f)

#output
5
```

Recipe 1-8. Scraping Text from the Web

In this recipe, we are going to discuss how to scrape data from the web.

Caution Before scraping any websites, blogs, or e-commerce websites, please make sure you read the terms and conditions of the websites on whether it gives permissions for data scraping.

So, what is web scraping, also called web harvesting or web data extraction?

It is a technique to extract a large amount of data from websites and save it in a database or locally. You can use this data to extract information related to your customers/users/products for the business's benefit.

Prerequisite: Basic understanding of HTML structure.

Problem

You want to extract data from the web by scraping. Here we have taken the example of the IMDB website for scraping top movies.

Solution

The simplest way to do this is by using beautiful soup or scrapy library from Python. Let's use beautiful soup in this recipe.

How It Works

Let's follow the steps in this section to extract data from the web.

Step 8-1 Install all the necessary libraries

```
!pip install bs4
!pip install requests
```

Step 8-2 Import the libraries

```
from bs4 import BeautifulSoup
import requests
import pandas as pd
from pandas import Series, DataFrame
from ipywidgets import FloatProgress
from time import sleep
from IPython.display import display
import re
import pickle
```

Step 8-3 Identify the url to extract the data

```
url = 'http://www.imdb.com/chart/top?ref_=nv_mv_250_6'
```

Step 8-4 Request the url and download the content using beautiful soup

```
result = requests.get(url)
c = result.content
soup = BeautifulSoup(c,"lxml")
```

Step 8-5 Understand the website page structure to extract the required information

Go to the website and right-click on the page content to inspect the html structure of the website.

Identify the data and fields you want to extract. Say, for example, we want the Movie name and IMDB rating from this page.

So, we will have to check under which div or class the movie names are present in the HTML and parse the beautiful soup accordingly.

In the below example, to extract the movie name, we can parse our soup through `<table class ="chart full-width">` and `<td class="titleColumn">`.

Similarly, we can fetch the other details. For more details, please refer to the code in step 8-6.

Step 8-6 Use beautiful soup to extract and parse the data from HTML tags

```
summary = soup.find('div',{'class':'article'})

# Create empty lists to append the extracted data.

moviename = []
cast = []
description = []
rating = []
ratingoutof = []
year = []
genre = []
movielength = []
rot_audscore = []
rot_avgrating = []
rot_users = []

# Extracting the required data from the html soup.

rgx = re.compile('[%s]' % '()')
f = FloatProgress(min=0, max=250)
display(f)
for row,i in zip(summary.find('table').
findAll('tr'),range(len(summary.find('table').findAll('tr')))):
    for sitem in row.findAll('span',{'class':'secondaryInfo'}):
        s = sitem.find(text=True)
        year.append(rgx.sub('', s))
    for ritem in row.findAll('td',{'class':'ratingColumn
    imdbRating'}):
        for iget in ritem.findAll('strong'):
```

```
            rating.append(iget.find(text=True))
            ratingoutof.append(iget.get('title').split(' ', 4)[3])
    for item in row.findAll('td',{'class':'titleColumn'}):
        for href in item.findAll('a',href=True):
            moviename.append(href.find(text=True))
            rurl = 'https://www.rottentomatoes.com/m/'+ href.
            find(text=True)
            try:
                rresult = requests.get(rurl)
            except requests.exceptions.ConnectionError:
                status_code = "Connection refused"
            rc = rresult.content
            rsoup = BeautifulSoup(rc)
            try:
                rot_audscore.append(rsoup.find('div',
                {'class':'meter-value'}).find('span',
                {'class':'superPageFontColor'}).text)
                rot_avgrating.append(rsoup.find('div',
                {'class':'audience-info hidden-xs
                superPageFontColor'}).find('div').contents[2].
                strip())
                rot_users.append(rsoup.find('div',
                {'class':'audience-info hidden-xs
                superPageFontColor'}).contents[3].contents[2].
                strip())
            except AttributeError:
                rot_audscore.append("")
                rot_avgrating.append("")
                rot_users.append("")
            cast.append(href.get('title'))
```

```
            imdb = "http://www.imdb.com" + href.get('href')
            try:
                iresult = requests.get(imdb)
                ic = iresult.content
                isoup = BeautifulSoup(ic)
                description.append(isoup.find('div',
                {'class':'summary_text'}).find(text=True).strip())
                genre.append(isoup.find('span',{'class':'itempr
                op'}).find(text=True))
                movielength.append(isoup.find('time',
                {'itemprop':'duration'}).find(text=True).strip())
            except requests.exceptions.ConnectionError:
                description.append("")
                genre.append("")
                movielength.append("")
    sleep(.1)
    f.value = i
```

Note that there is a high chance that you might encounter an error while executing the above script because of the following reasons:

- Your request to the URL has failed, so maybe you need to try again after some time. This is common in web scraping.
- Web pages are dynamic. The HTML tags of websites keep changing. Understand the tags and make small changes in the code in accordance with HTML, and you are good to go.

Step 8-7 Convert lists to data frame and you can perform the analysis that meets the business requirements

```
# List to pandas series

moviename = Series(moviename)
cast = Series(cast)
description = Series(description)
rating = Series(rating)
ratingoutof = Series(ratingoutof)
year = Series(year)
genre = Series(genre)
movielength = Series(movielength)
rot_audscore = Series(rot_audscore)
rot_avgrating = Series(rot_avgrating)
rot_users = Series(rot_users)

# creating dataframe and doing analysis

imdb_df = pd.concat([moviename,year,description,genre,
          movielength,cast,rating,ratingoutof,
          rot_audscore,rot_avgrating,rot_users],axis=1)
imdb_df.columns = ['moviename','year','description','genre',
                   'movielength','cast','imdb_rating',
                   'imdb_ratingbasedon','tomatoes_audscore',
                   'tomatoes_rating','tomatoes_ratingbasedon']
imdb_df['rank'] = imdb_df.index + 1
imdb_df.head(1)
```

```
#output
```

	moviename	year	description	genre	movielength	cast	imdb_rating	imdb_ratingbasedon
0	The Shawshank Redemption	1994	Two imprisoned men bond over a number of years...	wrongful imprisonment	NaN	Frank Darabont (dir.), Tim Robbins, Morgan Fre...	9.2	1,994,354

Step 8-8 Download the data frame

```
# Saving the file as CSV.

imdb_df.to_csv("imdbdataexport.csv")
```

We have implemented most of the ways and techniques to extract text data from possible sources. In the coming chapters, we will look at how to explore, process, and clean this data, followed by feature engineering and building NLP applications.

CHAPTER 2

Exploring and Processing Text Data

In this chapter, we are going to cover various methods and techniques to preprocess the text data along with exploratory data analysis.

We are going to discuss the following recipes under text preprocessing and exploratory data analysis.

Recipe 1. Lowercasing

Recipe 2. Punctuation removal

Recipe 3. Stop words removal

Recipe 4. Text standardization

Recipe 5. Spelling correction

Recipe 6. Tokenization

Recipe 7. Stemming

Recipe 8. Lemmatization

Recipe 9. Exploratory data analysis

Recipe 10. End-to-end processing pipeline

A. Kulkarni and A. Shivananda, *Natural Language Processing Recipes*,
https://doi.org/10.1007/978-1-4842-4267-4_2

Before directly jumping into the recipes, let us first understand the need for preprocessing the text data. As we all know, around 90% of the world's data is unstructured and may be present in the form of an image, text, audio, and video. Text can come in a variety of forms from a list of individual words, to sentences to multiple paragraphs with special characters (like tweets and other punctuations). It also may be present in the form of web, HTML, documents, etc. And this data is never clean and consists of a lot of noise. It needs to be treated and then perform a few of the preprocessing functions to make sure we have the right input data for the feature engineering and model building. Suppose if we don't preprocess the data, any algorithms that are built on top of such data will not add any value for the business. This reminds me of a very popular phrase in the Data Science world "Garbage in – Garbage out."

Preprocessing involves transforming raw text data into an understandable format. Real-world data is very often incomplete, inconsistent, and filled with a lot of noise and is likely to contain many errors. Preprocessing is a proven method of resolving such issues. Data preprocessing prepares raw text data for further processing.

Recipe 2-1. Converting Text Data to Lowercase

In this recipe, we are going to discuss how to lowercase the text data in order to have all the data in a uniform format and to make sure "NLP" and "nlp" are treated as the same.

Problem

How to lowercase the text data?

Solution

The simplest way to do this is by using the default `lower()` function in Python.

The `lower()` method converts all uppercase characters in a string into lowercase characters and returns them.

How It Works

Let's follow the steps in this section to lowercase a given text or document. Here, we are going to use Python.

Step 1-1 Read/create the text data

Let's create a list of strings and assign it to a variable.

```
text=['This is introduction to NLP','It is likely to be useful,
to people ','Machine learning is the new electrcity','There
would be less hype around AI and more action going
forward','python is the best tool!','R is good langauage','I
like this book','I want more books like this']
```

```
#convert list to data frame
import pandas as pd
df = pd.DataFrame({'tweet':text})
print(df)
```

```
#output
                                                tweet
0                         This is introduction to NLP
1              It is likely to be useful, to people
2             Machine learning is the new electrcity
3  There would be less hype around AI and more ac...
4                            python is the best tool!
```

```
5                                  R is good langauage
6                                     I like this book
7                           I want more books like this
```

Step 1-2 Execute lower() function on the text data

When there is just the string, apply the `lower()` function directly as shown below:

```
x = 'Testing'
x2 = x.lower()
print(x2)
```

```
#output
'testing'
```

When you want to perform lowercasing on a data frame, use the apply a function as shown below:

```
df['tweet'] = df['tweet'].apply(lambda x: " ".join(x.lower()
for x in x.split()))
df['tweet']
```

```
#output
0                            this is introduction to nlp
1                  it is likely to be useful, to people
2                machine learning is the new electrcity
3    there would be less hype around ai and more ac...
4                               python is the best tool!
5                                    r is good langauage
6                                       i like this book
7                            i want more books like this
```

That's all. We have converted the whole tweet column into lowercase. Let's see what else we can do in the next recipes.

Recipe 2-2. Removing Punctuation

In this recipe, we are going to discuss how to remove punctuation from the text data. This step is very important as punctuation doesn't add any extra information or value. Hence removal of all such instances will help reduce the size of the data and increase computational efficiency.

Problem

You want to remove punctuation from the text data.

Solution

The simplest way to do this is by using the regex and `replace()` function in Python.

How It Works

Let's follow the steps in this section to remove punctuation from the text data.

Step 2-1 Read/create the text data

Let's create a list of strings and assign it to a variable.

```
text=['This is introduction to NLP','It is likely to be useful,
to people ','Machine learning is the new electrcity',
'There would be less hype around AI and more action going
forward','python is the best tool!','R is good langauage',
'I like this book','I want more books like this']
#convert list to dataframe
import pandas as pd
df = pd.DataFrame({'tweet':text})
print(df)
```

```
#output
 tweet
0 This is introduction to NLP
1 It is likely to be useful, to people
2 Machine learning is the new electrcity
3 There would be less hype around AI and more ac...
4 python is the best tool!
5 R is good langauage
6 I like this book
7 I want more books like this
```

Step 2-2 Execute below function on the text data

Using the regex and `replace()` function, we can remove the punctuation as shown below:

```
import re

s = "I. like. This book!"
s1 = re.sub(r'[^\w\s]',",s)
s1

#output
'I like This book'
```

Or:

```
df['tweet'] = df['tweet'].str.replace('[^\w\s]',")
df['tweet']

#output
0                          this is introduction to nlp
1                 it is likely to be useful to people
2               machine learning is the new electrcity
3    there would be less hype around ai and more ac...
```

```
4                                  python is the best tool
5                                      r is good langauage
6                                         i like this book
7                               i want more books like this
```

Or:

```
import string

s = "I. like. This book!"

for c in string.punctuation:
      s= s.replace(c,"")
s

#output
'I like This book'
```

Recipe 2-3. Removing Stop Words

In this recipe, we are going to discuss how to remove stop words. Stop words are very common words that carry no meaning or less meaning compared to other keywords. If we remove the words that are less commonly used, we can focus on the important keywords instead. Say, for example, in the context of a search engine, if your search query is "How to develop chatbot using python," if the search engine tries to find web pages that contained the terms "how," "to," "develop," "chatbot," "using," "python," the search engine is going to find a lot more pages that contain the terms "how" and "to" than pages that contain information about developing chatbot because the terms "how" and "to" are so commonly used in the English language. So, if we remove such terms, the search engine can actually focus on retrieving pages that contain the keywords: "develop," "chatbot," "python" – which would more closely bring up pages that are of real interest. Similarly we can remove more common words and rare words as well.

Problem

You want to remove stop words.

Solution

The simplest way to do this by using the NLTK library, or you can build your own stop words file.

How It Works

Let's follow the steps in this section to remove stop words from the text data.

Step 3-1 Read/create the text data

Let's create a list of strings and assign it to a variable.

```
text=['This is introduction to NLP','It is likely to be useful,
to people ','Machine learning is the new electrcity',
'There would be less hype around AI and more action going
forward','python is the best tool!','R is good langauage','I like
this book','I want more books like this']

#convert list to data frame
import pandas as pd
df = pd.DataFrame({'tweet':text})
print(df)

#output
 tweet
0 This is introduction to NLP
1 It is likely to be useful, to people
2 Machine learning is the new electrcity
3 There would be less hype around AI and more ac...
```

```
4 python is the best tool!
5 R is good langauage
6 I like this book
7 I want more books like this
```

Step 3-2 Execute below commands on the text data

Using the NLTK library, we can remove the punctuation as shown below.

```
#install and import libraries

!pip install nltk
import nltk
nltk.download()
from nltk.corpus import stopwords

#remove stop words

stop = stopwords.words('english')
df['tweet'] = df['tweet'].apply(lambda x: "
              ".join(x for x in x.split() if x not in stop))
df['tweet']

#output

0                                  introduction nlp
1                              likely useful people
2                   machine learning new electrcity
3    would less hype around ai action going forward
4                                  python best tool
5                                  r good langauage
6                                        like book
7                                   want books like
```

There are no stop words now. Everything has been removed in this step.

Recipe 2-4. Standardizing Text

In this recipe, we are going to discuss how to standardize the text. But before that, let's understand what is text standardization and why we need to do it. Most of the text data is in the form of either customer reviews, blogs, or tweets, where there is a high chance of people using short words and abbreviations to represent the same meaning. This may help the downstream process to easily understand and resolve the semantics of the text.

Problem

You want to standardize text.

Solution

We can write our own custom dictionary to look for short words and abbreviations.

How It Works

Let's follow the steps in this section to perform text standardization.

Step 4-1 Create a custom lookup dictionary

The dictionary will be for text standardization based on your data.

```
lookup_dict = {'nlp':'natural language processing',
'ur':'your', "wbu" : "what about you"}

import re
```

Step 4-2 Create a custom function for text standardization

Here is the code:

```
def text_std(input_text):
 words = input_text.split()
 new_words = []
 for word in words:
     word = re.sub(r'[^\w\s]',",word)
     if word.lower() in lookup_dict:
         word = lookup_dict[word.lower()]
         new_words.append(word)
         new_text = " ".join(new_words)
 return new_text
```

Step 4-3 Run the text_std function

We also need to check the output:

```
text_std("I like nlp it's ur choice")
```

```
#output
'natural language processing your'
```

```
Here, nlp has standardised to 'natural language processing' and
ur to 'your'.
```

Recipe 2-5. Correcting Spelling

In this recipe, we are going to discuss how to do spelling correction. But before that, let's understand why this spelling correction is important. Most of the text data is in the form of either customer reviews, blogs, or tweets, where there is a high chance of people using short words and

making typo errors. This will help us in reducing multiple copies of words, which represents the same meaning. For example, "proccessing" and "processing" will be treated as different words even if they are used in the same sense.

Note that abbreviations should be handled before this step, or else the corrector would fail at times. Say, for example, "ur" (actually means "your") would be corrected to "or."

Problem

You want to do spelling correction.

Solution

The simplest way to do this by using the TextBlob library.

How It Works

Let's follow the steps in this section to do spelling correction.

Step 5-1 Read/create the text data

Let's create a list of strings and assign it to a variable.

```
text=['Introduction to NLP','It is likely to be useful, to
people ','Machine learning is the new electrcity', 'R is good
langauage','I like this book','I want more books like this']

#convert list to dataframe
import pandas as pd
df = pd.DataFrame({'tweet':text})
print(df)
```

```
#output
                                    tweet
0                      Introduction to NLP
1   It is likely to be useful, to people
2  Machine learning is the new electrcity
3                      R is good langauage
4                         I like this book
5              I want more books like this
```

Step 5-2 Execute below code on the text data

Using TextBlob, we can do spelling correction as shown below:

```
#Install textblob library
!pip install textblob

#import libraries and use 'correct' function

from textblob import TextBlob

df['tweet'].apply(lambda x: str(TextBlob(x).correct()))

#output
0                         Introduction to NLP
1      It is likely to be useful, to people
2    Machine learning is the new electricity
3                          R is good language
4                            I like this book
5                I want more books like this
```

If you clearly observe this, it corrected the spelling of `electricity` and language.

```
#You can also use autocorrect library as shown below

#install autocorrect

!pip install autocorrect

from autocorrect import spell
print(spell(u'mussage'))
print(spell(u'sirvice'))

#output
'message'
'service'
```

Recipe 2-6. Tokenizing Text

In this recipe, we would look at the ways to tokenize. Tokenization refers to splitting text into minimal meaningful units. There is a sentence tokenizer and word tokenizer. We will see a word tokenizer in this recipe, which is a mandatory step in text preprocessing for any kind of analysis. There are many libraries to perform tokenization like NLTK, SpaCy, and TextBlob. Here are a few ways to achieve it.

Problem

You want to do tokenization.

Solution

The simplest way to do this is by using the TextBlob library.

How It Works

Let's follow the steps in this section to perform tokenization.

Step 6-1 Read/create the text data

Let's create a list of strings and assign it to a variable.

```
text=['This is introduction to NLP','It is likely to be useful,
to people ','Machine learning is the new electrcity',
'There would be less hype around AI and more action going
forward','python is the best tool!','R is good langauage',
'I like this book','I want more books like this']
```

```
#convert list to dataframe
import pandas as pd
df = pd.DataFrame({'tweet':text})
print(df)
```

```
#output
 tweet
0 This is introduction to NLP
1 It is likely to be useful, to people
2 Machine learning is the new electrcity
3 There would be less hype around AI and more ac...
4 python is the best tool!
5 R is good langauage
6 I like this book
7 I want more books like this
```

Step 6-2 Execute below code on the text data

The result of tokenization is a list of tokens:

```
#Using textblob
from textblob import TextBlob
TextBlob(df['tweet'][3]).words

#output
WordList(['would', 'less', 'hype', 'around', 'ai', 'action',
'going', 'forward'])

#using NLTK
import nltk

#create data
mystring = "My favorite animal is cat"

nltk.word_tokenize(mystring)

#output
['My', 'favorite', 'animal', 'is', 'cat']

#using split function from python
mystring.split()

#output
['My', 'favorite', 'animal', 'is', 'cat']
```

Recipe 2-7. Stemming

In this recipe, we will discuss stemming. Stemming is a process of extracting a root word. For example, “fish,” “fishes,” and “fishing” are stemmed into fish.

Problem

You want to do stemming.

Solution

The simplest way to do this by using NLTK or a TextBlob library.

How It Works

Let's follow the steps in this section to perform stemming.

Step 7-1 Read the text data

Let's create a list of strings and assign it to a variable.

```
text=['I like fishing','I eat fish','There are many fishes in pound']
```

```
#convert list to dataframe
import pandas as pd
df = pd.DataFrame({'tweet':text})
print(df)
```

```
#output
                            tweet
0                 I like fishing
1                     I eat fish
2  There are many fishes in pound
```

Step 7-2 Stemming the text

Execute the below code on the text data:

```
#Import library
from nltk.stem import PorterStemmer
```

```
st = PorterStemmer()

df['tweet'][:5].apply(lambda x: " ".join([st.stem(word) for
word in x.split()]))

#output
0                     I like fish
1                      I eat fish
2    there are mani fish in pound
```

If you observe this, you will notice that fish, fishing, and fishes have been stemmed to fish.

Recipe 2-8. Lemmatizing

In this recipe, we will discuss lemmatization. Lemmatization is a process of extracting a root word by considering the vocabulary. For example, "good," "better," or "best" is lemmatized into good.

The part of speech of a word is determined in lemmatization. It will return the dictionary form of a word, which must be a valid word while stemming just extracts the root word.

- Lemmatization handles matching "car" to "cars" along with matching "car" to "automobile."
- Stemming handles matching "car" to "cars."

Lemmatization can get better results.

- The stemmed form of leafs is leaf.
- The stemmed form of leaves is leav.
- The lemmatized form of leafs is leaf.
- The lemmatized form of leaves is leaf.

Problem

You want to perform lemmatization.

Solution

The simplest way to do this is by using NLTK or the TextBlob library.

How It Works

Let's follow the steps in this section to perform lemmatization.

Step 8-1 Read the text data

Let's create a list of strings and assign it to a variable.

```
text=['I like fishing','I eat fish','There are many fishes in
pound', 'leaves and leaf']
```

```
#convert list to dataframe
import pandas as pd
df = pd.DataFrame({'tweet':text})
```

```
print(df)
                                tweet
0                      I like fishing
1                          I eat fish
2  There are multiple fishes in pound
3                     leaves and leaf
```

Step 8-2 Lemmatizing the data

Execute the below code on the text data:

```
#Import library
from textblob import Word

#Code for lemmatize
df['tweet'] = df['tweet'].apply(lambda x: " ".join([Word(word).
lemmatize() for word in x.split()]))

df['tweet']

#output
0                      I like fishing
1                          I eat fish
2    There are multiple fish in pound
3                       leaf and leaf
```

You can observe that fish and fishes are lemmatized to fish and, as explained, leaves and leaf are lemmatized to leaf.

Recipe 2-9. Exploring Text Data

So far, we are comfortable with data collection and text preprocessing. Let us perform some exploratory data analysis.

Problem

You want to explore and understand the text data.

Solution

The simplest way to do this by using NLTK or the TextBlob library.

How It Works

Let's follow the steps in this process.

Step 9-1 Read the text data

Execute the below code to download the dataset, if you haven't already done so:

```
nltk.download().
```

```
#Importing data
import nltk
from nltk.corpus import webtext
nltk.download('webtext')
wt_sentences = webtext.sents('firefox.txt')
wt_words = webtext.words('firefox.txt')
```

Step 9-2 Import necessary libraries

Import Library for computing frequency:

```
from nltk.probability import FreqDist
from nltk.corpus import stopwords
import string
```

Step 9-3 Check number of words in the data

Count the number of words:

```
len(wt_sentences)
```

```
#output
1142
```

```
len(wt_words)
```

```
#output
102457
```

Step 9-4 Compute the frequency of all words in the reviews

Generating frequency for all the words:

```
frequency_dist = nltk.FreqDist(wt_words)
frequency_dist
```

```
#showing only top few results


FreqDist({'slowing': 1,
          'warnings': 6,
          'rule': 1,
          'Top': 2,
          'XBL': 12,
          'installation': 44,
          'Networking': 1,
          'inccorrect': 1,
          'killed': 3,
          ']"': 1,
          'LOCKS': 1,
          'limited': 2,
          'cookies': 57,
          'method': 12,
          'arbitrary': 2,
          'b': 3,
          'titlebar': 6,
```

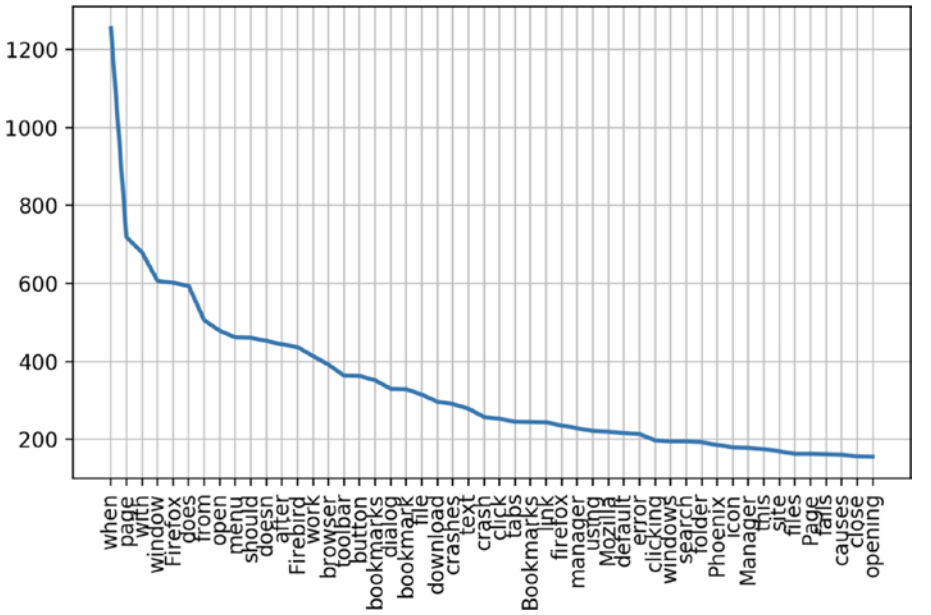
```
sorted_frequency_dist =sorted(frequency_dist,key=frequency_
dist.__getitem__, reverse=True)
sorted_frequency_dist
```

```
['.',
 'in',
 'to',
 '"',
 'the',
 "'",
 'not',
 '-',
 'when',
 'on',
 'a',
 'is',
 't',
 'and',
 'of',
```

Step 9-5 Consider words with length greater than 3 and plot

Let's take the words only if their frequency is greater than 3.

```
large_words = dict([(k,v) for k,v in frequency_dist.items() if
len(k)>3])

frequency_dist = nltk.FreqDist(large_words)
frequency_dist.plot(50,cumulative=False)

#output
```

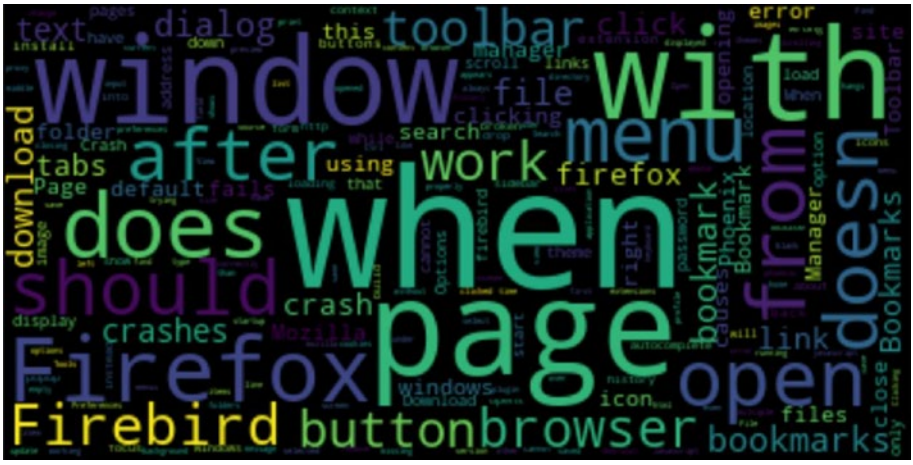


Step 9-6 Build Wordcloud

Wordcloud is the pictorial representation of the most frequently repeated words representing the size of the word.

```
#install library
!pip install wordcloud

#build wordcloud

from wordcloud import WordCloud
wcloud = WordCloud().generate_from_frequencies(frequency_dist)

#plotting the wordcloud

import matplotlib.pyplot as plt
plt.imshow(wcloud, interpolation='bilinear')
```

```
plt.axis("off")
(-0.5, 399.5, 199.5, -0.5)
plt.show()

#output
```



Readers, give this a try: Remove the stop words and then build the word cloud. The output would look something like that below.

Recipe 2-10. Building a Text Preprocessing Pipeline

So far, we have completed most of the text manipulation and processing techniques and methods. In this recipe, let's do something interesting.

Problem

You want to build an end-to-end text preprocessing pipeline. Whenever you want to do preprocessing for any NLP application, you can directly plug in data to this pipeline function and get the required clean text data as the output.

Solution

The simplest way to do this by creating the custom function with all the techniques learned so far.

How It Works

This works by putting all the possible processing techniques into a wrapper function and passing the data through it.

Step 10-1 Read/create the text data

Let's create a list of strings and assign it to a variable. Maybe a tweet sample:

```
tweet_sample= "How to take control of your #debt https://
personal.vanguard.com/us/insights/saving-investing/
debt-management.#Best advice for #family #financial #success
(@PrepareToWin)"
```

You can also use your Twitter data extracted in Chapter 1.

Step 10-2 Process the text

Execute the below function to process the tweet:

```
def processRow(row):

    import re
    import nltk
    from textblob import TextBlob
    from nltk.corpus import stopwords
    from nltk.stem import PorterStemmer
    from textblob import Word
    from nltk.util import ngrams
    import re
    from wordcloud import WordCloud, STOPWORDS
    from nltk.tokenize import word_tokenize

    tweet = row
    #Lower case
    tweet.lower()
    #Removes unicode strings like "\u002c" and "x96"
    tweet = re.sub(r'(\\u[0-9A-Fa-f]+)',r", tweet)
    tweet = re.sub(r'[^\x00-\x7f]',r",tweet)
    #convert any url to URL
    tweet = re.sub('((www\.[^\s]+)|(https?://[^\s]+))','URL',tweet)
    #Convert any @Username to "AT_USER"
    tweet = re.sub('@[^\s]+','AT_USER',tweet)
    #Remove additional white spaces
    tweet = re.sub('[\s]+', ' ', tweet)
    tweet = re.sub('[\n]+', ' ', tweet)
    #Remove not alphanumeric symbols white spaces
    tweet = re.sub(r'[^\w]', ' ', tweet)
    #Removes hastag in front of a word """
```

```
tweet = re.sub(r'#([^\s]+)', r'\1', tweet)
#Replace #word with word
tweet = re.sub(r'#([^\s]+)', r'\1', tweet)
#Remove :( or :)
tweet = tweet.replace(':)',")
tweet = tweet.replace(':(',")
#remove numbers
tweet = ".join([i for i in tweet if not i.isdigit()])
#remove multiple exclamation
tweet = re.sub(r"(\!)\1+", ' ', tweet)
#remove multiple question marks
tweet = re.sub(r"(\?)\1+", ' ', tweet)
#remove multistop
tweet = re.sub(r"(\.)\1+", ' ', tweet)
#lemma
from textblob import Word
tweet =" ".join([Word(word).lemmatize() for word in tweet.
         split()])
#stemmer
#st = PorterStemmer()
#tweet=" ".join([st.stem(word) for word in tweet.split()])
#Removes emoticons from text
tweet = re.sub(':\)|;\)|:-\)|\(-:|:-D|=D|:P|xD|X
        -p|\^\^|:-*|\^\.\^|\^\-\^|\^\_\^|\,-\)|\)-:|:\'\
        (|:\(|:-\(|:\S|T\.T|\.\_\.|:<|:-\S|:-<|\*\-
        \*|:O|=O|=\-O|O\.o|XO|O\_O|:-\@|=/|:/|X\-\
        (|>\.<|>=\(|D:', ", tweet)
#trim
tweet = tweet.strip('\'"')
```

```
    row = tweet

    return row

#call the function with your data
processRow(tweet_sample)

#output
'How to take control of your debt URL Best advice for family
financial success AT_USER'
```

CHAPTER 3

Converting Text to Features

In this chapter, we are going to cover basic to advanced feature engineering (text to features) methods. By the end of this chapter, you will be comfortable with the following recipes:

Recipe 1. One Hot encoding

Recipe 2. Count vectorizer

Recipe 3. N-grams

Recipe 4. Co-occurrence matrix

Recipe 5. Hash vectorizer

Recipe 6. Term Frequency-Inverse Document Frequency (TF-IDF)

Recipe 7. Word embedding

Recipe 8. Implementing fastText

Now that all the text preprocessing steps are discussed, let's explore feature engineering, the foundation for Natural Language Processing. As we already know, machines or algorithms cannot understand the characters/words or sentences, they can only take numbers as input that also includes binaries. But the inherent nature of text data is unstructured and noisy, which makes it impossible to interact with machines.

A. Kulkarni and A. Shivananda, *Natural Language Processing Recipes*,
https://doi.org/10.1007/978-1-4842-4267-4_3

The procedure of converting raw text data into machine understandable format (numbers) is called feature engineering of text data. Machine learning and deep learning algorithms' performance and accuracy is fundamentally dependent on the type of feature engineering technique used.

In this chapter, we will discuss different types of feature engineering methods along with some state-of-the-art techniques; their functionalities, advantages, disadvantages; and examples for each. All of these will make you realize the importance of feature engineering.

Recipe 3-1. Converting Text to Features Using One Hot Encoding

The traditional method used for feature engineering is One Hot encoding. If anyone knows the basics of machine learning, One Hot encoding is something they should have come across for sure at some point of time or maybe most of the time. It is a process of converting categorical variables into features or columns and coding one or zero for the presence of that particular category. We are going to use the same logic here, and the number of features is going to be the number of total tokens present in the whole corpus.

Problem

You want to convert text to feature using One Hot encoding.

Solution

One Hot Encoding will basically convert characters or words into binary numbers as shown below.

	I	love	NLP	is	future
I love NLP	1	1	1	0	0
NLP is future	0	0	1	1	1

How It Works

There are so many functions to generate One Hot encoding. We will take one function and discuss it in depth.

Step 1-1 Store the text in a variable

This is for a single line:

```
Text = "I am learning NLP"
```

Step 1-2 Execute below function on the text data

Below is the function from the pandas library to convert text to feature.

```
# Importing the library

import pandas as pd

# Generating the features

pd.get_dummies(Text.split())
```

Result:

```
   I  NLP  am  learning
0  1    0   0         0
1  0    0   1         0
2  0    0   0         1
3  0    1   0         0
```

Output has 4 features since the number of distinct words present in the input was 4.

Recipe 3-2. Converting Text to Features Using Count Vectorizing

The approach in Recipe 3-1 has a disadvantage. It does not take the frequency of the word occurring into consideration. If a particular word is appearing multiple times, there is a chance of missing the information if it is not included in the analysis. A count vectorizer will solve that problem.

In this recipe, we will see the other method of converting text to feature, which is a count vectorizer.

Problem

How do we convert text to feature using a count vectorizer?

Solution

Count vectorizer is almost similar to One Hot encoding. The only difference is instead of checking whether the particular word is present or not, it will count the words that are present in the document.

Observe the below example. The words "I" and "NLP" occur twice in the first document.

	I	love	NLP	is	future	will	learn	in	2month
I love NLP and I will learn NLP in 2 months	2	1	2	0	0	1	1	1	1
NLP is future	0	0	1	1	1	0	0	0	0

How It Works

Sklearn has a feature extraction function that extracts features out of the text. Let's discuss how to execute the same. Import the `CountVectorizer` function from Sklearn as explained below.

```
#importing the function

from sklearn.feature_extraction.text import CountVectorizer

# Text

text = ["I love NLP and I will learn NLP in 2month "]

# create the transform

vectorizer = CountVectorizer()

# tokenizing

vectorizer.fit(text)

# encode document

vector = vectorizer.transform(text)

# summarize & generating output

print(vectorizer.vocabulary_)
print(vector.toarray())

Result:


{'love': 4, 'nlp': 5, 'and': 1, 'will': 6, 'learn': 3, 'in': 2,
'2month': 0}
[[1 1 1 1 1 2 1]]
```

The fifth token `nlp` has appeared twice in the document.

Recipe 3-3. Generating N-grams

If you observe the above methods, each word is considered as a feature. There is a drawback to this method.

It does not consider the previous and the next words, to see if that would give a proper and complete meaning to the words.

For example: consider the word "not bad." If this is split into individual words, then it will lose out on conveying "good" – which is what this word actually means.

As we saw, we might lose potential information or insight because a lot of words make sense once they are put together. This problem can be solved by N-grams.

N-grams are the fusion of multiple letters or multiple words. They are formed in such a way that even the previous and next words are captured.

- Unigrams are the unique words present in the sentence.
- Bigram is the combination of 2 words.
- Trigram is 3 words and so on.

For example,
"I am learning NLP"
Unigrams: "I", "am", " learning", "NLP"
Bigrams: "I am", "am learning", "learning NLP"
Trigrams: "I am learning", "am learning NLP"

Problem

Generate the N-grams for the given sentence.

Solution

There are a lot of packages that will generate the N-grams. The one that is mostly used is TextBlob.

How It Works

Following are the steps.

Step 3-1 Generating N-grams using TextBlob

Let us see how to generate N-grams using TextBlob.

```
Text = "I am learning NLP"
```

Use the below `TextBlob` function to create N-grams. Use the text that is defined above and mention the "n" based on the requirement.

```
#Import textblob
from textblob import TextBlob

#For unigram : Use n = 1

TextBlob(Text).ngrams(1)
```

Output:

```
[WordList(['I']), WordList(['am']), WordList(['learning']),
WordList(['NLP'])]

#For Bigram : For bigrams, use n = 2

TextBlob(Text).ngrams(2)

[WordList(['I', 'am']),
 WordList(['am', 'learning']),
 WordList(['learning', 'NLP'])]
```

If we observe, we have 3 lists with 2 words at an instance.

Step 3-2 Bigram-based features for a document

Just like in the last recipe, we will use count vectorizer to generate features. Using the same function, let us generate bigram features and see what the output looks like.

```
#importing the function

from sklearn.feature_extraction.text import CountVectorizer

# Text

text = ["I love NLP and I will learn NLP in 2month "]

# create the transform

vectorizer = CountVectorizer(ngram_range=(2,2))

# tokenizing

vectorizer.fit(text)

# encode document

vector = vectorizer.transform(text)

# summarize & generating output
print(vectorizer.vocabulary_)
print(vector.toarray())
```

Result:

```
{'love nlp': 3, 'nlp and': 4, 'and will': 0, 'will learn': 6,
'learn nlp': 2, 'nlp in': 5, 'in 2month': 1}
[[1 1 1 1 1 1 1]]
```

The output has features with bigrams, and for our example, the count is one for all the tokens.

Recipe 3-4. Generating Co-occurrence Matrix

Let's discuss one more method of feature engineering called a co-occurrence matrix.

Problem

Understand and generate a co-occurrence matrix.

Solution

A co-occurrence matrix is like a count vectorizer where it counts the occurrence of the words together, instead of individual words.

How It Works

Let's see how to generate these kinds of matrixes using `nltk, bigrams,` and some basic Python coding skills.

Step 4-1 Import the necessary libraries

Here is the code:

```
import numpy as np
import nltk
from nltk import bigrams
import itertools
```

Step 4-2 Create function for co-occurrence matrix

The co_occurrence_matrix function is below.

```
def co_occurrence_matrix(corpus):
    vocab = set(corpus)
    vocab = list(vocab)
    vocab_to_index = { word:i for i, word in enumerate(vocab) }
    # Create bigrams from all words in corpus
    bi_grams = list(bigrams(corpus))
    # Frequency distribution of bigrams ((word1, word2),
      num_occurrences)
    bigram_freq = nltk.FreqDist(bi_grams).most_common(len(bi_
                  grams))
    # Initialise co-occurrence matrix
    # co_occurrence_matrix[current][previous]
    co_occurrence_matrix = np.zeros((len(vocab), len(vocab)))

    # Loop through the bigrams taking the current and previous word,
    # and the number of occurrences of the bigram.
    for bigram in bigram_freq:
        current = bigram[0][1]
        previous = bigram[0][0]
        count = bigram[1]
        pos_current = vocab_to_index[current]
        pos_previous = vocab_to_index[previous]
        co_occurrence_matrix[pos_current][pos_previous] = count
    co_occurrence_matrix = np.matrix(co_occurrence_matrix)
    # return the matrix and the index
    return co_occurrence_matrix,vocab_to_index
```

Step 4-3 Generate co-occurrence matrix

Here are the sentences for testing:

```
sentences = [['I', 'love', 'nlp'],
                  ['I', 'love','to' 'learn'],
                  ['nlp', 'is', 'future'],
                  ['nlp', 'is', 'cool']]

# create one list using many lists

merged = list(itertools.chain.from_iterable(sentences))
matrix = co_occurrence_matrix(merged)

# generate the matrix

CoMatrixFinal = pd.DataFrame(matrix[0], index=vocab_to_index,
columns=vocab_to_index)
print(CoMatrixFinal)
```

```
           I   is  love  future  tolearn  cool  nlp
I        0.0  0.0   0.0     0.0      0.0   0.0  1.0
is       0.0  0.0   0.0     0.0      0.0   0.0  2.0
love     2.0  0.0   0.0     0.0      0.0   0.0  0.0
future   0.0  1.0   0.0     0.0      0.0   0.0  0.0
tolearn  0.0  0.0   1.0     0.0      0.0   0.0  0.0
cool     0.0  1.0   0.0     0.0      0.0   0.0  0.0
nlp      0.0  0.0   1.0     1.0      1.0   0.0  0.0
```

If you observe, "I," "love," and "is," nlp" has appeared together twice, and a few other words appeared only once.

Recipe 3-5. Hash Vectorizing

A count vectorizer and co-occurrence matrix have one limitation though. In these methods, the vocabulary can become very large and cause memory/computation issues.

One of the ways to solve this problem is a Hash Vectorizer.

Problem

Understand and generate a Hash Vectorizer.

Solution

Hash Vectorizer is memory efficient and instead of storing the tokens as strings, the vectorizer applies the hashing trick to encode them as numerical indexes. The downside is that it's one way and once vectorized, the features cannot be retrieved.

How It Works

Let's take an example and see how to do it using `sklearn`.

Step 5-1 Import the necessary libraries and create document

Here's the code:

```
from sklearn.feature_extraction.text import HashingVectorizer

# list of text documents
text = ["The quick brown fox jumped over the lazy dog."]
```

Step 5-2 Generate hash vectorizer matrix

Let's create the HashingVectorizer of a vector size of 10.

```
# transform
vectorizer = HashingVectorizer(n_features=10)

# create the hashing vector
vector = vectorizer.transform(text)

# summarize the vector
print(vector.shape)
print(vector.toarray())
```

```
(1, 10)
[[ 0.          0.57735027  0.        0.        0.        0.        0.
  -0.57735027  -0.57735027  0.        ]]
```

It created vector of size 10 and now this can be used for any supervised/unsupervised tasks.

Recipe 3-6. Converting Text to Features Using TF-IDF

Again, in the above-mentioned text-to-feature methods, there are few drawbacks, hence the introduction of TF-IDF. Below are the disadvantages of the above methods.

- Let's say a particular word is appearing in all the documents of the corpus, then it will achieve higher importance in our previous methods. That's bad for our analysis.
- The whole idea of having TF-IDF is to reflect on how important a word is to a document in a collection, and hence normalizing words appeared frequently in all the documents.

Problem

Text to feature using TF-IDF.

Solution

Term frequency (TF): Term frequency is simply the ratio of the count of a word present in a sentence, to the length of the sentence.

TF is basically capturing the importance of the word irrespective of the length of the document. For example, a word with the frequency of 3 with the length of sentence being 10 is not the same as when the word length of sentence is 100 words. It should get more importance in the first scenario; that is what TF does.

Inverse Document Frequency (IDF): IDF of each word is the log of the ratio of the total number of rows to the number of rows in a particular document in which that word is present.

IDF = log(N/n), where N is the total number of rows and n is the number of rows in which the word was present.

IDF will measure the rareness of a term. Words like "a," and "the" show up in all the documents of the corpus, but rare words will not be there in all the documents. So, if a word is appearing in almost all documents, then that word is of no use to us since it is not helping to classify or in information retrieval. IDF will nullify this problem.

TF-IDF is the simple product of TF and IDF so that both of the drawbacks are addressed, which makes predictions and information retrieval relevant.

How It Works

Let's look at the following steps.

Step 6-1 Read the text data

A familiar phrase:

```
Text = ["The quick brown fox jumped over the lazy dog.",
"The dog.",
"The fox"]
```

Step 6-2 Creating the Features

Execute the below code on the text data:

```
#Import TfidfVectorizer

from sklearn.feature_extraction.text import TfidfVectorizer

#Create the transform

vectorizer = TfidfVectorizer()

#Tokenize and build vocab

vectorizer.fit(Text)

#Summarize

print(vectorizer.vocabulary_)
print(vectorizer.idf_)
```

Result:

```
Text = ["The quick brown fox jumped over the lazy dog.",
"The dog.",
"The fox"]

{'the': 7, 'quick': 6, 'brown': 0, 'fox': 2, 'jumped': 3,
'over': 5, 'lazy': 4, 'dog': 1}
```

```
[ 1.69314718  1.28768207  1.28768207  1.69314718   1.69314718
  1.69314718  1.69314718  1.    ]
```

If you observe, "the" is appearing in all the 3 documents and it does not add much value, and hence the vector value is 1, which is less than all the other vector representations of the tokens.

All these methods or techniques we have looked into so far are based on frequency and hence called frequency-based embeddings or features. And in the next recipe, let us look at prediction-based embeddings, typically called word embeddings.

Recipe 3-7. Implementing Word Embeddings

This recipe assumes that you have a working knowledge of how a neural network works and the mechanisms by which weights in the neural network are updated. If new to a Neural Network (NN), it is suggested that you go through Chapter 6 to gain a basic understanding of how NN works.

Even though all previous methods solve most of the problems, once we get into more complicated problems where we want to capture the semantic relation between the words, these methods fail to perform.

Below are the challenges:

- All these techniques fail to capture the context and meaning of the words. All the methods discussed so far basically depend on the appearance or frequency of the words. But we need to look at how to capture the context or semantic relations: that is, how frequently the words are appearing close by.

a. I am eating an *apple*.

b. I am using *apple*.

If you observe the above example, Apple gives different meanings when it is used with different (close by) adjacent words, eating and using.

- For a problem like a document classification (book classification in the library), a document is really huge and there are a humongous number of tokens generated. In these scenarios, your number of features can get out of control (wherein) thus hampering the accuracy and performance.

A machine/algorithm can match two documents/texts and say whether they are same or not. But how do we make machines tell you about cricket or Virat Kohli when you search for MS Dhoni? How do you make a machine understand that "Apple" in "Apple is a tasty fruit" is a fruit that can be eaten and not a company?

The answer to the above questions lies in creating a representation for words that capture their meanings, semantic relationships, and the different types of contexts they are used in.

The above challenges are addressed by **Word Embeddings**.

Word embedding is the feature learning technique where words from the vocabulary are mapped to vectors of real numbers capturing the contextual hierarchy.

If you observe the below table, every word is represented with 4 numbers called vectors. Using the word embeddings technique, we are going to derive those vectors for each and every word so that we can use it in future analysis. In the below example, the dimension is 4. But we usually use a dimension greater than 100.

Words	Vectors			
text	0.36	0.36	-0.43	0.36
idea	-0.56	-0.56	0.72	-0.56
word	0.35	-0.43	0.12	0.72
encode	0.19	0.19	0.19	0.43
document	-0.43	0.19	-0.43	0.43
grams	0.72	-0.43	0.72	0.12
process	0.43	0.72	0.43	0.43
feature	0.12	0.45	0.12	0.87

Problem

You want to implement word embeddings.

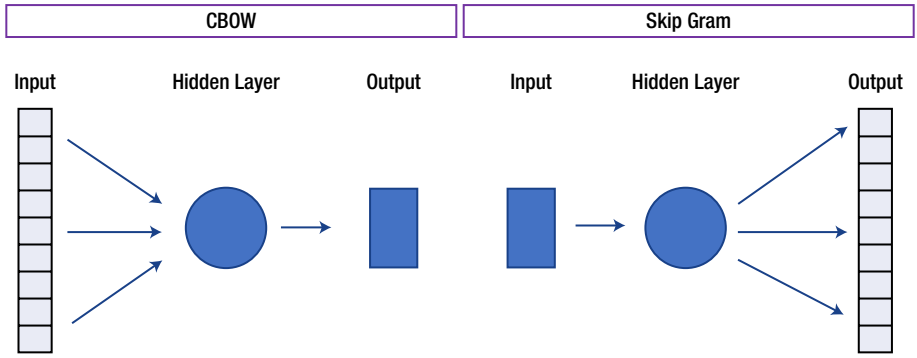
Solution

Word embeddings are prediction based, and they use shallow neural networks to train the model that will lead to learning the weight and using them as a vector representation.

word2vec: word2vec is the deep learning Google framework to train word embeddings. It will use all the words of the whole corpus and predict the nearby words. It will create a vector for all the words present in the corpus in a way so that the context is captured. It also outperforms any other methodologies in the space of word similarity and word analogies.

There are mainly 2 types in word2vec.

- Skip-Gram
- Continuous Bag of Words (CBOW)



How It Works

The above figure shows the architecture of the CBOW and skip-gram algorithms used to build word embeddings. Let us see how these models work in detail.

Skip-Gram

The skip-gram model (Mikolov et al., 2013)[1] is used to predict the probabilities of a word given the context of word or words.

Let us take a small sentence and understand how it actually works. Each sentence will generate a target word and context, which are the words nearby. The number of words to be considered around the target variable is called the window size. The table below shows all the possible target and context variables for window size 2. Window size needs to be selected based on data and the resources at your disposal. The larger the window size, the higher the computing power.

[1] https://arxiv.org/abs/1310.4546

Text = "I love NLP and I will learn NLP in 2 months"

	Target word	Context
I love NLP	I	love, NLP
I love NLP and	love	love, NLP, and
I love NLP and I will learn	NLP	I, love, and, I
…	…	…
in 2 months	month	in, 2

Since it takes a lot of text and computing power, let us go ahead and take sample data and build a skip-gram model.

As mentioned in Chapter 3, import the text corpus and break it into sentences. Perform some cleaning and preprocessing like the removal of punctuation and digits, and split the sentences into words or tokens, etc.

```
#Example sentences

sentences = [['I', 'love', 'nlp'],
                  ['I', 'will', 'learn', 'nlp', 'in', '2','months'],
                  ['nlp', 'is', 'future'],
                  ['nlp', 'saves', 'time', 'and', 'solves',
                   'lot', 'of', 'industry', 'problems'],
                  ['nlp', 'uses', 'machine', 'learning']]

#import library

!pip install gensim

import gensim
from gensim.models import Word2Vec
from sklearn.decomposition import PCA
from matplotlib import pyplot
```

```
# training the model

skipgram = Word2Vec(sentences, size =50, window = 3, min_count=1,
sg = 1)
print(skipgram)

# access vector for one word

print(skipgram['nlp'])

[ 0.00552227 -0.00723104  0.00857073  0.00368054 -0.00071274
  0.00837146
  0.00179965 -0.0049786  -0.00448666 -0.00182289  0.00857488
 -0.00499459
  0.00188365 -0.0093498   0.00174774 -0.00609793  -0.00533857
 -0.007905
 -0.00176814 -0.00024082 -0.00181886 -0.00093836 -0.00382601
 -0.00986026
  0.00312014 -0.00821249  0.00787507 -0.00864689 -0.00686584
 -0.00370761
  0.0056183   0.00859488 -0.00163146  0.00928791  0.00904601
  0.00443816
 -0.00192308  0.00941    -0.00202355 -0.00756564 -0.00105471
  0.00170084
  0.00606918 -0.00848301 -0.00543473  0.00747958  0.0003408
  0.00512787
 -0.00909613  0.00683905]
```

Since our vector size parameter was 50, the model gives a vector of size 50 for each word.

```
# access vector for another one word

print(skipgram['deep'])

KeyError: "word 'deep' not in vocabulary"
```

We get an error saying the word doesn't exist because this word was not there in our input training data. This is the reason we need to train the algorithm on as much data possible so that we do not miss out on words.

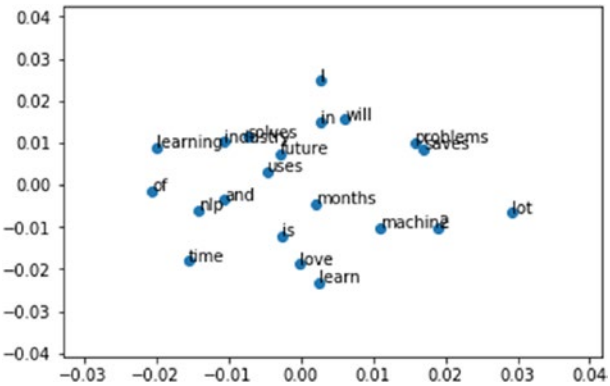
There is one more way to tackle this problem. Read Recipe 3-6 in this chapter for the answer.

```
# save model

skipgram.save('skipgram.bin')

# load model

skipgram = Word2Vec.load('skipgram.bin')
```

T – SNE plot is one of the ways to evaluate word embeddings. Let's generate it and see how it looks.

```
# T – SNE plot

X = skipgram[skipgram.wv.vocab]
pca = PCA(n_components=2)
result = pca.fit_transform(X)

# create a scatter plot of the projection

pyplot.scatter(result[:, 0], result[:, 1])
words = list(skipgram.wv.vocab)
for i, word in enumerate(words):
       pyplot.annotate(word, xy=(result[i, 0], result[i, 1]))
pyplot.show()
```

Result:



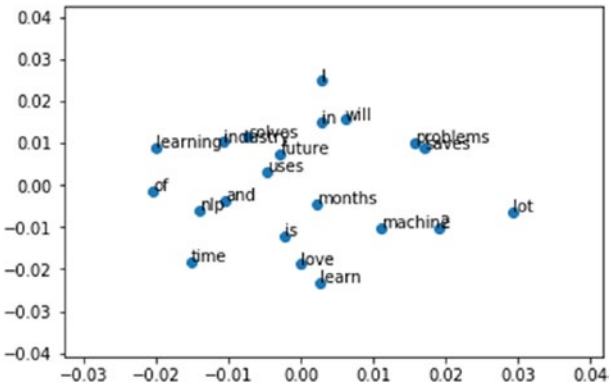
Continuous Bag of Words (CBOW)

Now let's see how to build CBOW model.

```
#import library

from gensim.models import Word2Vec
from sklearn.decomposition import PCA
from matplotlib import pyplot

#Example sentences

sentences = [['I', 'love', 'nlp'],
                  ['I', 'will', 'learn', 'nlp', 'in', '2','months'],
                  ['nlp', 'is', 'future'],
                  ['nlp', 'saves', 'time', 'and', 'solves',
                   'lot', 'of', 'industry', 'problems'],
                  ['nlp', 'uses', 'machine', 'learning']]
```

```
# training the model

cbow = Word2Vec(sentences, size =50, window = 3, min_count=1,sg = 1)
print(cbow)

# access vector for one word

print(cbow['nlp'])

# save model

cbow.save('cbow.bin')

# load model

cbow = Word2Vec.load('cbow.bin')

# T – SNE plot

X = cbow[cbow.wv.vocab]
pca = PCA(n_components=2)
result = pca.fit_transform(X)

# create a scatter plot of the projection

pyplot.scatter(result[:, 0], result[:, 1])
words = list(cbow.wv.vocab)
for i, word in enumerate(words):
       pyplot.annotate(word, xy=(result[i, 0], result[i, 1]))
pyplot.show()
```

Result:



But to train these models, it requires a huge amount of computing power. So, let us go ahead and use Google's pre-trained model, which has been trained with over 100 billion words.

Download the model from the below path and keep it in your local storage:

https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit

Import the gensim package and follow the steps to understand Google's word2vec.

```
# import gensim package

import gensim

# load the saved model

model = gensim.models.Word2Vec.load_word2vec_format('C:\\
Users\\GoogleNews-vectors-negative300.bin', binary=True)

#Checking how similarity works.

print (model.similarity('this', 'is'))
```

```
Output:
0.407970363878

#Lets check one more.
print (model.similarity('post', 'book'))

Output:
0.0572043891977
```

"This" and "is" have a good amount of similarity, but the similarity between the words "post" and "book" is poor. For any given set of words, it uses the vectors of both the words and calculates the similarity between them.

```
# Finding the odd one out.

model.doesnt_match('breakfast cereal dinner lunch';.split())
```

Output:

```
'cereal'
```

Of 'breakfast', 'cereal', 'dinner' and 'lunch', only cereal is the word that is not anywhere related to the remaining 3 words.

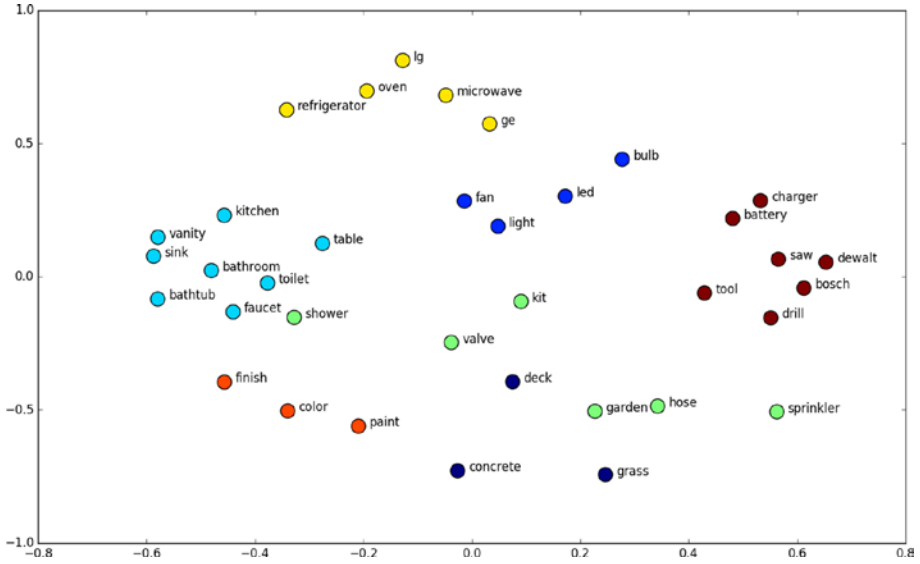
```
# It is also finding the relations between words.

word_vectors.most_similar(positive=['woman', 'king'],
negative=['man'])
```

Output:

```
queen: 0.7699
```

If you add 'woman' and 'king' and minus man, it is predicting queen as output with 77% confidence. Isn't this amazing?

Let's have a look at few of the interesting examples using T – SNE plot for word embeddings.



Above is the word embedding's output representation of home interiors and exteriors. If you clearly observe, all the words related to electric fittings are near to each other; similarly, words related to bathroom fittings are near to each other, and so on. This is the beauty of word embeddings.

Recipe 3-8 Implementing fastText

fastText is another deep learning framework developed by Facebook to capture context and meaning.

Problem

How to implement fastText in Python.

Solution

fastText is the improvised version of word2vec. word2vec basically considers words to build the representation. But fastText takes each character while computing the representation of the word.

How It Works

Let us see how to build a fastText word embedding.

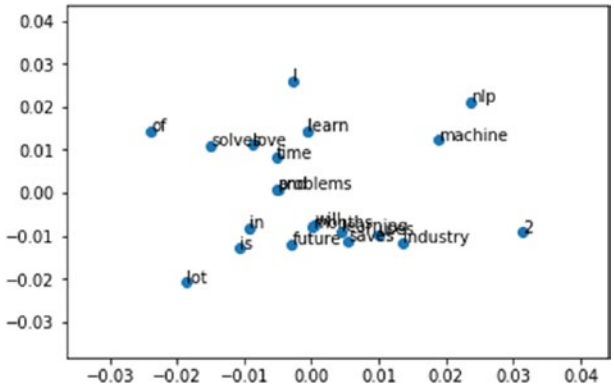
```
# Import FastText

from gensim.models import FastText
from sklearn.decomposition import PCA
from matplotlib import pyplot

#Example sentences

sentences = [['I', 'love', 'nlp'],
                  ['I', 'will', 'learn', 'nlp', 'in', '2','months'],
                  ['nlp', 'is', 'future'],
                  ['nlp', 'saves', 'time', 'and', 'solves',
                   'lot', 'of', 'industry', 'problems'],
                  ['nlp', 'uses', 'machine', 'learning']]

fast = FastText(sentences,size=20, window=1, min_count=1,
workers=5, min_n=1, max_n=2)

# vector for word nlp

print(fast['nlp'])
```

```
[-0.00459182  0.00607472 -0.01119007  0.00555629 -0.00781679
 -0.01376211
  0.00675235 -0.00840158 -0.00319737  0.00924599  0.00214165
 -0.01063819
  0.01226836  0.00852781  0.01361119 -0.00257012  0.00819397
 -0.00410289
 -0.0053979  -0.01360016]
# vector for word deep

print(fast['deep'])

[ 0.00271002 -0.00242539 -0.00771885 -0.00396854  0.0114902
 -0.00640606
  0.00637542 -0.01248098 -0.01207364  0.01400793 -0.00476079
 -0.00230879
  0.02009759 -0.01952532  0.01558956 -0.01581665  0.00510567
 -0.00957186
 -0.00963234 -0.02059373]
```

This is the advantage of using fastText. The "deep" was not present in training of word2vec and we did not get a vector for that word. But since fastText is building on character level, even for the word that was not there in training, it will provide results. You can see the vector for the word "deep," but it's not present in the input data.

```
# load model

fast = Word2Vec.load('fast.bin')

# visualize

X = fast[fast.wv.vocab]
pca = PCA(n_components=2)
result = pca.fit_transform(X)
```

```
# create a scatter plot of the projection

pyplot.scatter(result[:, 0], result[:, 1])
words = list(fast.wv.vocab)
for i, word in enumerate(words):
      pyplot.annotate(word, xy=(result[i, 0], result[i, 1]))
pyplot.show()
```



The figure above shows the embedding representation for fastText. If you observe closely, the words "love" and "solve" are close together in fastText but in your skip-gram and CBOW, "love" and "learn" are near to each other. This is an effect of character-level embeddings.

We hope that by now you are familiar and comfortable with processing the natural language. Now that data is cleaned and features are created, let's jump into building some applications around it that solves the business problem.

CHAPTER 4

Advanced Natural Language Processing

In this chapter, we are going to cover various advanced NLP techniques and leverage machine learning algorithms to extract information from text data as well as some of the advanced NLP applications with the solution approach and implementation.

Recipe 1. Noun Phrase extraction

Recipe 2. Text similarity

Recipe 3. Parts of speech tagging

Recipe 4. Information extraction – NER – Entity recognition

Recipe 5. Topic modeling

Recipe 6. Text classification

Recipe 7. Sentiment analysis

Recipe 8. Word sense disambiguation

Recipe 9. Speech recognition and speech to text

Recipe 10. Text to speech

Recipe 11. Language detection and translation

A. Kulkarni and A. Shivananda, *Natural Language Processing Recipes*,
https://doi.org/10.1007/978-1-4842-4267-4_4

Before getting into recipes, let's understand the NLP pipeline and life cycle first. There are so many concepts we are implementing in this book, and we might get overwhelmed by the content of it. To make it simpler and smoother, let's see what is the flow that we need to follow for an NLP solution.

For example, let's consider customer sentiment analysis and prediction for a product or brand or service.

- **Define the Problem**: Understand the customer sentiment across the products.
- **Understand the depth and breadth of the problem**: Understand the customer/user sentiments across the product; why we are doing this? What is the business impact? Etc.
- **Data requirement brainstorming**: Have a brainstorming activity to list out all possible data points.
    - All the reviews from customers on e-commerce platforms like Amazon, Flipkart, etc.
    - Emails sent by customers
    - Warranty claim forms
    - Survey data
    - Call center conversations using speech to text
    - Feedback forms
    - Social media data like Twitter, Facebook, and LinkedIn

- **Data collection**: We learned different techniques to collect the data in Chapter 1. Based on the data and the problem, we might have to incorporate different data collection methods. In this case, we can use web scraping and Twitter APIs.
- **Text Preprocessing**: We know that data won't always be clean. We need to spend a significant amount of time to process it and extract insight out of it using different methods that we discussed earlier in Chapter 2.
- **Text to feature**: As we discussed, texts are characters and machines will have a tough time understanding them. We have to convert them to features that machines and algorithms can understand using any of the methods we learned in the previous chapter.
- **Machine learning/Deep learning**: Machine learning/Deep learning is a part of an artificial intelligence umbrella that will make systems automatically learn patterns in the data without being programmed. Most of the NLP solutions are based on this, and since we converted text to features, we can leverage machine learning or deep learning algorithms to achieve the goals like text classification, natural language generation, etc.
- **Insights and deployment**: There is absolutely no use for building NLP solutions without proper insights being communicated to the business. Always take time to connect the dots between model/analysis output and the business, thereby creating the maximum impact.

Recipe 4-1. Extracting Noun Phrases

In this recipe, let us extract a noun phrase from the text data (a sentence or the documents).

Problem

You want to extract a noun phrase.

Solution

Noun Phrase extraction is important when you want to analyze the "who" in a sentence. Let's see an example below using `TextBlob`.

How It Works

Execute the below code to extract noun phrases.

```
#Import libraries
import nltk
from textblob import TextBlob

#Extract noun
blob = TextBlob("John is learning natural language processing")
for np in blob.noun_phrases:
    print(np)
```

Output:

```
john
natural language processing
```

Recipe 4-2. Finding Similarity Between Texts

In this recipe, we are going to discuss how to find the similarity between two documents or text. There are many similarity metrics like Euclidian, cosine, Jaccard, etc. Applications of text similarity can be found in areas like spelling correction and data deduplication.

Here are a few of the similarity measures:

Cosine similarity: Calculates the cosine of the angle between the two vectors.

Jaccard similarity: The score is calculated using the intersection or union of words.

Jaccard Index = (the number in both sets) / (the number in either set) * 100.

Levenshtein distance: Minimal number of insertions, deletions, and replacements required for transforming string "a" into string "b."

Hamming distance: Number of positions with the same symbol in both strings. But it can be defined only for strings with equal length.

Problem

You want to find the similarity between texts/documents.

Solution

The simplest way to do this is by using cosine similarity from the sklearn library.

How It Works

Let's follow the steps in this section to compute the similarity score between text documents.

Step 2-1 Create/read the text data

Here is the data:

```
documents = (
"I like NLP",
"I am exploring NLP",
"I am a beginner in NLP",
"I want to learn NLP",
"I like advanced NLP"
)
```

Step 2-2 Find the similarity

Execute the below code to find the similarity.

```
#Import libraries
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

#Compute tfidf : feature engineering(refer previous chapter –
Recipe 3-4)

tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(documents)

tfidf_matrix.shape

#output
(5, 10)
```

```
#compute similarity for first sentence with rest of the sentences
cosine_similarity(tfidf_matrix[0:1],tfidf_matrix)

#output
array([[ 1.        ,  0.17682765,  0.14284054,  0.13489366,
0.68374784]])
```

If we clearly observe, the first sentence and last sentence have higher similarity compared to the rest of the sentences.

Phonetic matching

The next version of similarity checking is phonetic matching, which roughly matches the two words or sentences and also creates an alphanumeric string as an encoded version of the text or word. It is very useful for searching large text corpora, correcting spelling errors, and matching relevant names. Soundex and Metaphone are two main phonetic algorithms used for this purpose. The simplest way to do this is by using the fuzzy library.

1. Install and import the library

   ```
   !pip install fuzzy
   import fuzzy
   ```

2. Run the Soundex function

   ```
   soundex = fuzzy.Soundex(4)
   ```

3. Generate the phonetic form

   ```
   soundex('natural')

   #output
   'N364'

   soundex('natuaral')
   ```

```
#output
'N364'

soundex('language')

#output
'L52'

soundex('processing')

#output
'P625'
```

Soundex is treating "natural" and "natuaral" as the same, and the phonetic code for both of the strings is "N364." And for "language" and "processing," it is "L52" and "P625" respectively.

Recipe 4-3. Tagging Part of Speech

Part of speech (POS) tagging is another crucial part of natural language processing that involves labeling the words with a part of speech such as noun, verb, adjective, etc. POS is the base for Named Entity Resolution, Sentiment Analysis, Question Answering, and Word Sense Disambiguation.

Problem

Tagging the parts of speech for a sentence.

Solution

There are 2 ways a tagger can be built.

- Rule based - Rules created manually, which tag a word belonging to a particular POS.

- Stochastic based - These algorithms capture the sequence of the words and tag the probability of the sequence using hidden Markov models.

How It Works

Again, NLTK has the best POS tagging module. nltk.pos_tag(word) is the function that will generate the POS tagging for any given word. Use for loop and generate POS for all the words present in the document.

Step 3-1 Store the text in a variable

Here is the variable:

```
Text  =  "I love NLP and I will learn NLP in 2 month"
```

Step 3-2 NLTK for POS

Now the code:

```
# Importing necessary packages and stopwords
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize, sent_tokenize
stop_words = set(stopwords.words('english'))

# Tokenize the text
tokens = sent_tokenize(text)

#Generate tagging for all the tokens using loop
for i in tokens:
    words = nltk.word_tokenize(i)
    words = [w for w in words if not w in stop_words]
    #  POS-tagger.
    tags = nltk.pos_tag(words)

tags
```

Results:

```
[('I', 'PRP'),
 ('love', 'VBP'),
 ('NLP', 'NNP'),
 ('I', 'PRP'),
 ('learn', 'VBP'),
 ('NLP', 'RB'),
 ('2month', 'CD')]
```

Below are the short forms and explanation of POS tagging. The word "love" is VBP, which means verb, sing. present, non-3d take.

- CC coordinating conjunction
- CD cardinal digit
- DT determiner
- EX existential there (like: "there is" ... think of it like "there exists")
- FW foreign word
- IN preposition/subordinating conjunction
- JJ adjective 'big'
- JJR adjective, comparative 'bigger'
- JJS adjective, superlative 'biggest'
- LS list marker 1)
- MD modal could, will
- NN noun, singular 'desk'
- NNS noun plural 'desks'

- NNP proper noun, singular 'Harrison'
- NNPS proper noun, plural 'Americans'
- PDT predeterminer 'all the kids'
- POS possessive ending parent's
- PRP personal pronoun I, he, she
- PRP$ possessive pronoun my, his, hers
- RB adverb very, silently
- RBR adverb, comparative better
- RBS adverb, superlative best
- RP particle give up
- TO to go 'to' the store
- UH interjection
- VB verb, base form take
- VBD verb, past tense took
- VBG verb, gerund/present participle taking
- VBN verb, past participle taken
- VBP verb, sing. present, non-3d take
- VBZ verb, 3rd person sing. present takes
- WDT wh-determiner which
- WP wh-pronoun who, what
- WP$ possessive wh-pronoun whose
- WRB wh-adverb where, when

Recipe 4-4. Extract Entities from Text

In this recipe, we are going to discuss how to identify and extract entities from the text, called Named Entity Recognition. There are multiple libraries to perform this task like NLTK chunker, StanfordNER, SpaCy, opennlp, and NeuroNER; and there are a lot of APIs also like WatsonNLU, AlchemyAPI, NERD, Google Cloud NLP API, and many more.

Problem

You want to identify and extract entities from the text.

Solution

The simplest way to do this is by using the `ne_chunk` from NLTK or SpaCy.

How It Works

Let's follow the steps in this section to perform NER.

Step 4-1 Read/create the text data

Here is the text:

```
sent = "John is studying at Stanford University in California"
```

Step 4-2 Extract the entities

Execute the below code.

Using NLTK

```
#import libraries
import nltk
```

```
from nltk import ne_chunk
from nltk import word_tokenize

#NER
ne_chunk(nltk.pos_tag(word_tokenize(sent)), binary=False)

#output

Tree('S', [Tree('PERSON', [('John', 'NNP')]), ('is', 'VBZ'),
('studying', 'VBG'), ('at', 'IN'), Tree('ORGANIZATION',
[('Stanford', 'NNP'), ('University', 'NNP')]), ('in', 'IN'),
Tree('GPE', [('California', 'NNP')])])

Here "John" is tagged as "PERSON"
"Stanford" as "ORGANIZATION"
"California" as "GPE". Geopolitical entity, i.e. countries,
cities, states.
```

Using SpaCy

```
import spacy
nlp = spacy.load('en')

# Read/create a sentence
doc = nlp(u'Apple is ready to launch new phone worth $10000 in
New york time square ')

for ent in doc.ents:
   print(ent.text, ent.start_char, ent.end_char, ent.label_)

#output

Apple 0 5 ORG
10000 42 47 MONEY
New york 51 59 GPE
```

According to the output, Apple is an organization, 10000 is money, and New York is place. The results are accurate and can be used for any NLP applications.

Recipe 4-5. Extracting Topics from Text

In this recipe, we are going to discuss how to identify topics from the document. Say, for example, there is an online library with multiple departments based on the kind of book. As the new book comes in, you want to look at the unique keywords/topics and decide on which department this book might belong to and place it accordingly. In these kinds of situations, topic modeling would be handy.

Basically, this is document tagging and clustering.

Problem

You want to extract or identify topics from the document.

Solution

The simplest way to do this by using the gensim library.

How It Works

Let's follow the steps in this section to identify topics within documents using genism.

Step 5-1 Create the text data

Here is the text:

```
doc1 = "I am learning NLP, it is very interesting and exciting.
it includes machine learning and deep learning"
```

```
doc2 = "My father is a data scientist and he is nlp expert"
doc3 = "My sister has good exposure into android development"

doc_complete = [doc1, doc2, doc3]
doc_complete

#output
['I am learning NLP, it is very interesting and exciting. it
includes machine learning and deep learning',
 'My father is a data scientist and he is nlp expert',
 'My sister has good exposure into android development']
```

Step 5-2 Cleaning and preprocessing

Next, we clean it up:

```
# Install and import libraries

!pip install gensim
from nltk.corpus import stopwords
from nltk.stem.wordnet import WordNetLemmatizer
import string

# Text preprocessing as discussed in chapter 2

stop = set(stopwords.words('english'))
exclude = set(string.punctuation)
lemma = WordNetLemmatizer()
def clean(doc):
    stop_free = " ".join([i for i in doc.lower().split()
                if i not in stop])
    punc_free = ''.join(ch for ch in stop_free if ch not in
                exclude)
    normalized = " ".join(lemma.lemmatize(word) for word in
                 punc_free.split())
```

```
    return normalized

doc_clean = [clean(doc).split() for doc in doc_complete]

doc_clean

#output
[['learning',
  'nlp',
  'interesting',
  'exciting',
  'includes',
  'machine',
  'learning',
  'deep',
  'learning'],
 ['father', 'data', 'scientist', 'nlp', 'expert'],
 ['sister', 'good', 'exposure', 'android', 'development']]
```

Step 5-3 Preparing document term matrix

The code is below:

```
# Importing gensim

import gensim
from gensim import corpora

# Creating the term dictionary of our corpus, where every
unique term is assigned an index.

dictionary = corpora.Dictionary(doc_clean)

# Converting a list of documents (corpus) into Document-Term
Matrix using dictionary prepared above.
```

```
doc_term_matrix = [dictionary.doc2bow(doc) for doc in doc_clean]

doc_term_matrix

#output
[[(0, 1), (1, 1), (2, 1), (3, 1), (4, 3), (5, 1), (6, 1)],
 [(6, 1), (7, 1), (8, 1), (9, 1), (10, 1)],
 [(11, 1), (12, 1), (13, 1), (14, 1), (15, 1)]]
```

Step 5-4 LDA model

The final part is to create the LDA model:

```
# Creating the object for LDA model using gensim library
Lda = gensim.models.ldamodel.LdaModel

# Running and Training LDA model on the document term matrix
for 3 topics.
ldamodel = Lda(doc_term_matrix, num_topics=3, id2word =
dictionary, passes=50)

# Results
print(ldamodel.print_topics())

#output
[(0, '0.063*"nlp" + 0.063*"father" + 0.063*"data" +
0.063*"scientist" + 0.063*"expert" + 0.063*"good" +
0.063*"exposure" + 0.063*"development" + 0.063*"android" +
0.063*"sister"'), (1, '0.232*"learning" + 0.093*"nlp" +
0.093*"deep" + 0.093*"includes" + 0.093*"interesting" +
0.093*"machine" + 0.093*"exciting" + 0.023*"scientist" +
0.023*"data" + 0.023*"father"'), (2, '0.087*"sister" +
0.087*"good" + 0.087*"exposure" + 0.087*"development" +
0.087*"android" + 0.087*"father" + 0.087*"scientist" +
0.087*"data" + 0.087*"expert" + 0.087*"nlp"')]
```

All the weights associated with the topics from the sentence seem almost similar. You can perform this on huge data to extract significant topics. The whole idea to implement this on sample data is to make you familiar with it, and you can use the same code snippet to perform on the huge data for significant results and insights.

Recipe 4-6. Classifying Text

Text classification – The aim of text classification is to automatically classify the text documents based on pretrained categories.

Applications:

- Sentiment Analysis
- Document classification
- Spam – ham mail classification
- Resume shortlisting
- Document summarization

Problem

Spam - ham classification using machine learning.

Solution

If you observe, your Gmail has a folder called “Spam.” It will basically classify your emails into spam and ham so that you don’t have to read unnecessary emails.

How It Works

Let's follow the step-by-step method to build the classifier.

Step 6-1 Data collection and understanding

Please download data from the below link and save it in your working directory:

https://www.kaggle.com/uciml/sms-spam-collection-dataset#spam.csv

```
#Read the data
Email_Data = pd.read_csv("spam.csv",encoding ='latin1')

#Data undestanding
Email_Data.columns

#output
Index(['v1', 'v2', 'Unnamed: 2', 'Unnamed: 3', 'Unnamed: 4'],
dtype='object')

Email_Data = Email_Data[['v1', 'v2']]
Email_Data = Email_Data.rename(columns={"v1":"Target",
"v2":"Email"})

Email_Data.head()

#output
    Target   Email
0      ham   Go until jurong point, crazy.. Available only ...
1      ham   Ok lar... Joking wif u oni...
2      spam  Free entry in 2 a wkly comp to win FA Cup fina...
3      ham   U dun say so early hor... U c already then say...
4      ham   Nah I don't think he goes to usf, he lives aro...
```

Step 6-2 Text processing and feature engineering

The code is below:

```
#import
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import string
from nltk.stem import SnowballStemmer
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
import os
from textblob import TextBlob
from nltk.stem import PorterStemmer
from textblob import Word
from sklearn.feature_extraction.text import CountVectorizer,
TfidfVectorizer
import sklearn.feature_extraction.text as text
from sklearn import model_selection, preprocessing, linear_
model, naive_bayes, metrics, svm

#pre processing steps like lower case, stemming and
lemmatization

Email_Data['Email'] = Email_Data['Email'].apply(lambda x:
                      " ".join(x.lower() for x in x.split()))
stop = stopwords.words('english')
Email_Data['Email'] = Email_Data['Email'].apply(lambda x: " ".join
                      (x for x in x.split() if x not in stop))
st = PorterStemmer()
```

```
Email_Data['Email'] = Email_Data['Email'].apply(lambda x: " ".join
                      ([st.stem(word) for word in x.split()]))
Email_Data['Email'] = Email_Data['Email'].apply(lambda x: " ".join
                      ([Word(word).lemmatize() for word in
                      x.split()]))

Email_Data.head()

#output
  Target                                                Email
0    ham  go jurong point, crazy.. avail bugi n great wo...
1    ham                        ok lar... joke wif u oni...
2    spam free entri 2 wkli comp win fa cup final tkt 21...
3    ham          u dun say earli hor... u c alreadi say...
4    ham              nah think goe usf, live around though

#Splitting data into train and validation

train_x, valid_x, train_y, valid_y = model_selection.train_
test_split(Email_Data['Email'], Email_Data['Target'])

# TFIDF feature generation for a maximum of 5000 features

encoder = preprocessing.LabelEncoder()
train_y = encoder.fit_transform(train_y)
valid_y = encoder.fit_transform(valid_y)

tfidf_vect = TfidfVectorizer(analyzer='word',
             token_pattern=r'\w{1,}', max_features=5000)
tfidf_vect.fit(Email_Data['Email'])
xtrain_tfidf =  tfidf_vect.transform(train_x)
xvalid_tfidf =  tfidf_vect.transform(valid_x)

xtrain_tfidf.data
```

```
#output
array([0.39933971, 0.36719906, 0.60411187, ..., 0.36682939,
0.30602539, 0.38290119])
```

Step 6-3 Model training

This is the generalized function for training any given model:

```
def train_model(classifier, feature_vector_train, label,
feature_vector_valid, is_neural_net=False):
    # fit the training dataset on the classifier
    classifier.fit(feature_vector_train, label)
    # predict the labels on validation dataset
    predictions = classifier.predict(feature_vector_valid)
    return metrics.accuracy_score(predictions, valid_y)

# Naive Bayes trainig
accuracy = train_model(naive_bayes.MultinomialNB(alpha=0.2),
xtrain_tfidf, train_y, xvalid_tfidf)
print ("Accuracy: ", accuracy)

#output
Accuracy:  0.985642498205

# Linear Classifier on Word Level TF IDF Vectors
accuracy = train_model(linear_model.LogisticRegression(),
           xtrain_tfidf, train_y, xvalid_tfidf)
print ("Accuracy: ", accuracy)

#output
Accuracy:  0.970567121321
```

Naive Bayes is giving better results than the linear classifier. We can try many more classifiers and then choose the best one.

Recipe 4-7. Carrying Out Sentiment Analysis

In this recipe, we are going to discuss how to understand the sentiment of a particular sentence or statement. Sentiment analysis is one of the widely used techniques across the industries to understand the sentiments of the customers/users around the products/services. Sentiment analysis gives the sentiment score of a sentence/statement tending toward positive or negative.

Problem

You want to do a sentiment analysis.

Solution

The simplest way to do this by using a TextBlob or vedar library.

How It Works

Let's follow the steps in this section to do sentiment analysis using TextBlob. It will basically give 2 metrics.

- Polarity = Polarity lies in the range of [-1,1] where 1 means a positive statement and -1 means a negative statement.
- Subjectivity = Subjectivity refers that mostly it is a public opinion and not factual information [0,1].

Step 7-1 Create the sample data

Here is the sample data:

```
review = "I like this phone. screen quality and camera clarity
         is really good."
review2 = "This tv is not good. Bad quality, no clarity, worst
          experience"
```

Step 7-2 Cleaning and preprocessing

Refer to Chapter 2, Recipe 2-10, for this step.

Step 7-3 Get the sentiment scores

Using a pretrained model from TextBlob to get the sentiment scores:

```
#import libraries
from textblob import TextBlob

#TextBlob has a pre trained sentiment prediction model
blob = TextBlob(review)
blob.sentiment

#output
Sentiment(polarity=0.7, subjectivity=0.6000000000000001)
```

It seems like a very positive review.

```
#now lets look at the sentiment of review2
blob = TextBlob(review2)
blob.sentiment
```

```
#output
Sentiment(polarity=-0.6833333333333332,
subjectivity=0.7555555555555555)
```

This is a negative review, as the polarity is "-0.68."

Note: We will cover a one real-time use case on sentiment analysis with an end-to-end implementation in the next chapter, Recipe 5-2.

Recipe 4-8. Disambiguating Text

There is ambiguity that arises due to a different meaning of words in a different context.

For example,

```
Text1 = 'I went to the bank to deposit my money'
Text2 = 'The river bank was full of dead fishes'
```

In the above texts, the word "bank" has different meanings based on the context of the sentence.

Problem

Understanding disambiguating word sense.

Solution

The Lesk algorithm is one of the best algorithms for word sense disambiguation. Let's see how to solve using the package pywsd and nltk.

How It Works

Below are the steps to achieve the results.

Step 8-1 Import libraries

First, import the libraries:

```
#Install pywsd

!pip install pywsd

#Import functions

from nltk.corpus import wordnet as wn
from nltk.stem import PorterStemmer
from itertools import chain
from pywsd.lesk import simple_lesk
```

Step 8-2 Disambiguating word sense

Now the code:

```
# Sentences

bank_sents = ['I went to the bank to deposit my money',
'The river bank was full of dead fishes']

# calling the lesk function and printing results for both the
sentences

print ("Context-1:", bank_sents[0])
answer = simple_lesk(bank_sents[0],'bank')
print ("Sense:", answer)
print ("Definition : ", answer.definition())

print ("Context-2:", bank_sents[1])
answer = simple_lesk(bank_sents[1],'bank','n')
print ("Sense:", answer)
print ("Definition : ", answer.definition())

#Result:
```

```
Context-1: I went to the bank to deposit my money
Sense: Synset('depository_financial_institution.n.01')
Definition :  a financial institution that accepts deposits and
channels the money into lending activities

Context-2: The river bank was full of dead fishes
Sense: Synset('bank.n.01')
Definition :  sloping land (especially the slope beside a body
of water)
```

Observe that in context-1, "bank" is a financial institution, but in context-2, "bank" is sloping land.

Recipe 4-9. Converting Speech to Text

Converting speech to text is a very useful NLP technique.

Problem

You want to convert speech to text.

Solution

The simplest way to do this by using Speech Recognition and PyAudio.

How It Works

Let's follow the steps in this section to implement speech to text.

Step 9-1 Understanding/defining business problem

Interaction with machines is trending toward the voice, which is the usual way of human communication. Popular examples are Siri, Alexa's Google Voice, etc.

Step 9-2 Install and import necessary libraries

Here are the libraries:

```
!pip install SpeechRecognition
!pip install PyAudio
```

```
import speech_recognition as sr
```

Step 9-3 Run below code

Now after you run the below code snippet, whatever you say on the microphone (using `recognize_google` function) gets converted into text.

```
r=sr.Recognizer()
```

```
with sr.Microphone() as source:
    print("Please say something")
    audio = r.listen(source)
    print("Time over, thanks")
```

```
try:
    print("I think you said: "+r.recognize_google(audio));
except:
    pass;
```

```
#output
Please say something
Time over, thanks
I think you said: I am learning natural language processing
```

This code works with the default language "English." If you speak in any other language, for example Hindi, the text is interpreted in the form of English, like as below:

```
#code snippet
r=sr.Recognizer()

with sr.Microphone() as source:
    print("Please say something")
    audio = r.listen(source)
    print("Time over, thanks")

try:
    print("I think you said: "+r.recognize_google(audio));
except:
    pass;

#output
Please say something
Time over, thanks
I think you said: aapka naam kya hai
```

If you want the text in the spoken language, please run the below code snippet. Where we have made the minor change is in the `recognize_google –language('hi-IN'`, which means Hindi).

```
#code snippet
r=sr.Recognizer()

with sr.Microphone() as source:
    print("Please say something")
    audio = r.listen(source)
    print("Time over, thanks")

try:
```

```
    print("I think you said: "+r.recognize_google(audio,
    language ='hi-IN'));
except sr.UnknownValueError:
    print("Google Speech Recognition could not understand audio")
except sr.RequestError as e:
    print("Could not request results from Google Speech
    Recognition service; {0}".format(e))
except:
    pass;
```

Recipe 4-10. Converting Text to Speech

Converting text to speech is another useful NLP technique.

Problem

You want to convert text to speech.

Solution

The simplest way to do this by using the gTTs library.

How It Works

Let's follow the steps in this section to implement text to speech.

Step 10-1 Install and import necessary libraries

Here are the libraries:

```
!pip install gTTS
```

```
from gtts import gTTS
```

Step 10-2 Run below code, gTTS function

Now after you run the below code snippet, whatever you input in the text parameter gets converted into audio.

```
#chooses the language, English('en')

convert = gTTS(text='I like this NLP book', lang='en', slow=False)

# Saving the converted audio in a mp3 file named
myobj.save("audio.mp3")

#output
Please play the audio.mp3 file saved in your local machine to
hear the audio.
```

Recipe 4-11. Translating Speech

Language detection and translation.

Problem

Whenever you try to analyze data from blogs that are hosted across the globe, especially websites from countries like China, where Chinese is used predominantly, analyzing such data or performing NLP tasks on such data would be difficult. That's where language translation comes to the rescue. You want to translate one language to another.

Solution

The easiest way to do this by using the goslate library.

How It Works

Let's follow the steps in this section to implement language translation in Python.

Step 11-1 Install and import necessary libraries

Here are the libraries:

```
!pip install goslate
import goslate
```

Step 11-2 Input text

A simple phrase:

```
text = "Bonjour le monde"
```

Step 11-3 Run goslate function

The translation function:

```
gs = goslate.Goslate()
translatedText = gs.translate(text,'en')

print(translatedText)

#output
Hi world
```

Well, it feels accomplished, isn't it? We have implemented so many advanced NLP applications and techniques. That is not all folks; we have a couple more interesting chapters ahead, where we will look at the industrial applications around NLP, their solution approach, and end-to-end implementation.

CHAPTER 5

Implementing Industry Applications

In this chapter, we are going to implement end-to-end solutions for a few of the Industry applications around NLP.

Recipe 1. Consumer complaint classification

Recipe 2. Customer reviews sentiment prediction

Recipe 3. Data stitching using record linkage

Recipe 4. Text summarization for subject notes

Recipe 5. Document clustering

Recipe 6. Search engine and learning to rank

We believe that after 4 chapters, you are comfortable with the concepts of natural language processing and ready to solve business problems. Here we need to keep all 4 chapters in mind and think of approaches to solve these problems at hand. It can be one concept or a series of concepts that will be leveraged to build applications.

So, let's go one by one and understand end-to-end implementation.

A. Kulkarni and A. Shivananda, *Natural Language Processing Recipes*,
https://doi.org/10.1007/978-1-4842-4267-4_5

Recipe 5-1. Implementing Multiclass Classification

Let's understand how to do multiclass classification for text data in Python through solving Consumer complaint classifications for the finance industry.

Problem

Each week the Consumer Financial Protection Bureau sends thousands of consumers' complaints about financial products and services to companies for a response. Classify those consumer complaints into the product category it belongs to using the description of the complaint.

Solution

The goal of the project is to classify the complaint into a specific product category. Since it has multiple categories, it becomes a multiclass classification that can be solved through many of the machine learning algorithms.

Once the algorithm is in place, whenever there is a new complaint, we can easily categorize it and can then be redirected to the concerned person. This will save a lot of time because we are minimizing the human intervention to decide whom this complaint should go to.

How It Works

Let's explore the data and build classification problem using many machine learning algorithms and see which one gives better results.

Step 1-1 Getting the data from Kaggle

Go to the below link and download the data.

https://www.kaggle.com/subhassing/exploring-consumer-complaint-data/data

Step 1-2 Import the libraries

Here are the libraries:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import string
from nltk.stem import SnowballStemmer
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
import os
from textblob import TextBlob
from nltk.stem import PorterStemmer
from textblob import Word
from sklearn.feature_extraction.text import CountVectorizer,
TfidfVectorizer
import sklearn.feature_extraction.text as text
from sklearn import model_selection, preprocessing,
linear_model, naive_bayes, metrics, svm
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import LinearSVC
from sklearn.model_selection import cross_val_score
from io import StringIO
import seaborn as sns
```

Step 1-3 Importing the data

Import the data that was downloaded in the last step:

```
Data = pd.read_csv("/Consumer_Complaints.csv",encoding='latin-1')
```

Step 1-4 Data understanding

Let's analyze the columns:

```
Data.dtypes
date_received                 object
product                       object
sub_product                   object
issue                         object
sub_issue                     object
consumer_complaint_narrative  object
company_public_response       object
company                       object
state                         object
zipcode                       object
tags                          object
consumer_consent_provided     object
submitted_via                 object
date_sent_to_company          object
company_response_to_consumer  object
timely_response               object
consumer_disputed?            object
complaint_id                   int64
```

```
# Selecting required columns and rows
Data = Data[['product', 'consumer_complaint_narrative']]
Data = Data[pd.notnull(Data['consumer_complaint_narrative'])]
```

```
# See top 5 rows
Data.head()
```

```
                product                consumer_complaint_narrative
190126  Debt collection    XXXX has claimed I owe them {$27.00}
                           for XXXX ...
190135    Consumer Loan    Due to inconsistencies in the amount
                           owed that...
190155         Mortgage    In XX/XX/XXXX my wages that I earned at
                           my job...
190207         Mortgage    I have an open and current mortgage
                           with Chase...
190208         Mortgage    XXXX was submitted XX/XX/XXXX. At the
                           time I s...
```

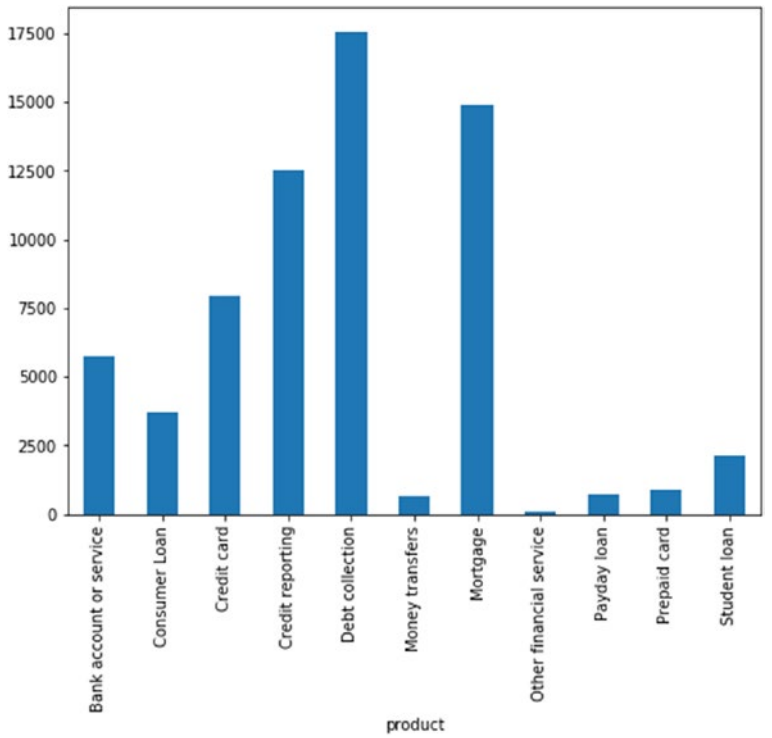
```
# Factorizing the category column
Data['category_id'] = Data['product'].factorize()[0]
Data.head()
```

```
                product    consumer_complaint_narrative   \
190126  Debt collection    XXXX has claimed I owe them {$27.00}
                           for XXXX ...
190135    Consumer Loan    Due to inconsistencies in the amount
                           owed that...

        category_id
190126            0
190135            1
```

```
# Check the distriution of complaints by category
Data.groupby('product').consumer_complaint_narrative.count()
product
Bank account or service    5711
Consumer Loan              3678
Credit card                7929
```

```
Credit reporting           12526
Debt collection            17552
Money transfers              666
Mortgage                   14919
Other financial service      110
Payday loan                  726
Prepaid card                 861
Student loan                2128
```

```
# Lets plot it and see
fig = plt.figure(figsize=(8,6))
Data.groupby('product').consumer_complaint_narrative.count().
plot.bar(ylim=0)
plt.show()
```



Debt collection and Mortgage have the highest number of complaints registered.

Step 1-5 Splitting the data

Split the data into train and validation:

```
train_x, valid_x, train_y, valid_y = model_selection.train_
test_split(Data['consumer_complaint_narrative'], Data['product'])
```

Step 1-6 Feature engineering using TF-IDF

Create TF-IDF vectors as we discussed in Chapter 3. Here we consider maximum features to be 5000.

```
encoder = preprocessing.LabelEncoder()
train_y = encoder.fit_transform(train_y)
valid_y = encoder.fit_transform(valid_y)

tfidf_vect = TfidfVectorizer(analyzer='word',
token_pattern=r'\w{1,}', max_features=5000)
tfidf_vect.fit(Data['consumer_complaint_narrative'])
xtrain_tfidf =  tfidf_vect.transform(train_x)
xvalid_tfidf =  tfidf_vect.transform(valid_x)
```

Step 1-7 Model building and evaluation

Suppose we are building a linear classifier on word-level TF-IDF vectors. We are using default hyper parameters for the classifier. Parameters can be changed like C, max_iter, or solver to obtain better results.

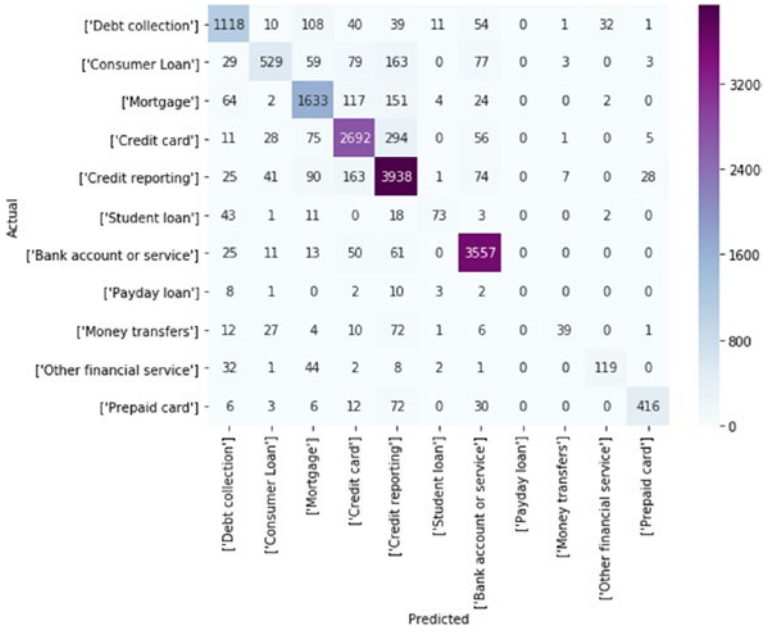
```
model = linear_model.LogisticRegression().fit(xtrain_tfidf, train_y)

# Model summary
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_
intercept=True,
```

```
          intercept_scaling=1, max_iter=100, multi_class='ovr',
          n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear',
          tol=0.0001,
          verbose=0, warm_start=False)
# Checking accuracy
accuracy = metrics.accuracy_score(model.predict(xvalid_tfidf),
valid_y)
print ("Accuracy: ", accuracy)
Accuracy:  0.845048497186
# Classification report
print(metrics.classification_report(valid_y, model.
predict(xvalid_tfidf),target_names=Data['product'].unique()))
```

	precision	recall	f1-score	support
Debt collection	0.81	0.79	0.80	1414
Consumer Loan	0.81	0.56	0.66	942
Mortgage	0.80	0.82	0.81	1997
Credit card	0.85	0.85	0.85	3162
Credit reporting	0.82	0.90	0.86	4367
Student loan	0.77	0.48	0.59	151
Bank account or service	0.92	0.96	0.94	3717
Payday loan	0.00	0.00	0.00	26
Money transfers	0.76	0.23	0.35	172
Other financial service	0.77	0.57	0.65	209
Prepaid card	0.92	0.76	0.83	545
avg / total	0.84	0.85	0.84	16702

```
#confusion matrix
conf_mat = confusion_matrix(valid_y, model.predict(xvalid_tfidf))
# Vizualizing confusion matrix
```

```
category_id_df = Data[['product', 'category_id']].drop_duplicates()
.sort_values('category_id')
category_to_id = dict(category_id_df.values)
id_to_category = dict(category_id_df[['category_id',
'product']].values)

fig, ax = plt.subplots(figsize=(8,6))
sns.heatmap(conf_mat, annot=True, fmt='d', cmap="BuPu",
            xticklabels=category_id_df[['product']].values,
            yticklabels=category_id_df[['product']].values)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```



The accuracy of 85% is good for a baseline model. Precision and recall look pretty good across the categories except for "Payday loan." If you look for Payload loan, most of the wrong predictions are Debt collection and Credit card, which might be because of the smaller number of samples in that category. It also sounds like it's a subcategory of a credit card. We can add these samples to any other group to make the model more stable. Let's see what prediction looks like for one example.

```
# Prediction example
texts = ["This company refuses to provide me verification and
        validation of debt"+ "per my right under the FDCPA.
        I do not believe this debt is mine."]
text_features = tfidf_vect.transform(texts)
predictions = model.predict(text_features)
print(texts)
print("  - Predicted as: '{}'".format(id_to_
category[predictions[0]]))
```

```
Result :
['This company refuses to provide me verification and
validation of debtper my right under the FDCPA. I do not
believe this debt is mine.']
  - Predicted as: 'Credit reporting'
```

To increase the accuracy, we can do the following things:

- Reiterate the process with different algorithms like Random Forest, SVM, GBM, Neural Networks, Naive Bayes.
- Deep learning techniques like RNN and LSTM (will be discussed in next chapter) can also be used.

- In each of these algorithms, there are so many parameters to be tuned to get better results. It can be easily done through Grid search, which will basically try out all possible combinations and give the best out.

Recipe 5-2. Implementing Sentiment Analysis

In this recipe, we are going to implement, end to end, one of the popular NLP industrial applications – Sentiment Analysis. It is very important from a business standpoint to understand how customer feedback is on the products/services they offer to improvise on the products/service for customer satisfaction.

Problem

We want to implement sentiment analysis.

Solution

The simplest way to do this by using the `TextBlob` or `vaderSentiment` library. Since we have used TextBlob previously, now let us use vader.

How It Works

Let's follow the steps in this section to implement sentiment analysis on the business problem.

Step 2-1 Understanding/defining business problem

Understand how products are doing in the market. How are customers reacting to a particular product? What is the consumer's sentiment across products? Many more questions like these can be answered using sentiment analysis.

Step 2-2 Identifying potential data sources, collection, and understanding

We have a dataset for Amazon food reviews. Let's use that data and extract insight out of it. You can download the data from the link below:

```
https://www.kaggle.com/snap/amazon-fine-food-reviews
```

```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

#Read the data
df = pd.read_csv('Reviews.csv')

# Look at the top 5 rows of the data
df.head(5)

#output
```

	Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenominator	Score	Time	Summary	Text
0	1	B001E4KFG0	A3SGXH7AUHU8GW	delmartian	1	1	5	1303862400	Good Quality Dog Food	I have bought several of the Vitality canned d...
1	2	B00813GRG4	A1D87F6ZCVE5NK	dll pa	0	0	1	1346976000	Not as Advertised	Product arrived labeled as Jumbo Salted Peanut...
2	3	B000LQOCH0	ABXLMWJIXXAIN	Natalia Corres "Natalia Corres"	1	1	4	1219017600	"Delight" says it all	This is a confection that has been around a fe...
3	4	B000UA0QIQ	A395BORC6FGVXV	Karl	3	3	2	1307923200	Cough Medicine	If you are looking for the secret ingredient i...
4	5	B006K2ZZ7K	A1UQRSCLF8GW1T	Michael D. Bigham "M. Wassir"	0	0	5	1350777600	Great taffy	Great taffy at a great price. There was a wid...

```
# Understand the data types of the columns
df.info()

# Output
Data columns (total 10 columns):
Id                      5 non-null int64
ProductId               5 non-null object
UserId                  5 non-null object
ProfileName             5 non-null object
HelpfulnessNumerator    5 non-null int64
HelpfulnessDenominator  5 non-null int64
Score                   5 non-null int64
Time                    5 non-null int64
Summary                 5 non-null object
Text                    5 non-null object
dtypes: int64(5), object(5)
# Looking at the summary of the reviews.
df.Summary.head(5)

# Output
0    Good Quality Dog Food
1        Not as Advertised
```

```
2    "Delight" says it all
3           Cough Medicine
4              Great taffy
```

```
# Looking at the description of the reviews
df.Text.head(5)
```

```
#output
0    I have bought several of the Vitality canned d...
1    Product arrived labeled as Jumbo Salted Peanut...
2    This is a confection that has been around a fe...
3    If you are looking for the secret ingredient i...
4    Great taffy at a great price.  There was a wid...
```

Step 2-3 Text preprocessing

We all know the importance of this step. Let us perform a preprocessing task as discussed in Chapter 2.

```
# Import libraries
from nltk.corpus import stopwords
from textblob import TextBlob
from textblob import Word
```

```
# Lower casing and removing punctuations
df['Text'] = df['Text'].apply(lambda x: " ".join(x.lower() for
x in x.split()))
df['Text'] = df['Text'].str.replace('[^\w\s]',")
df.Text.head(5)
# Output
0    i have bought several of the vitality canned d...
1    product arrived labeled as jumbo salted peanut...
2    this is a confection that has been around a fe...
3    if you are looking for the secret ingredient i...
4    great taffy at a great price there was a wide ...
```

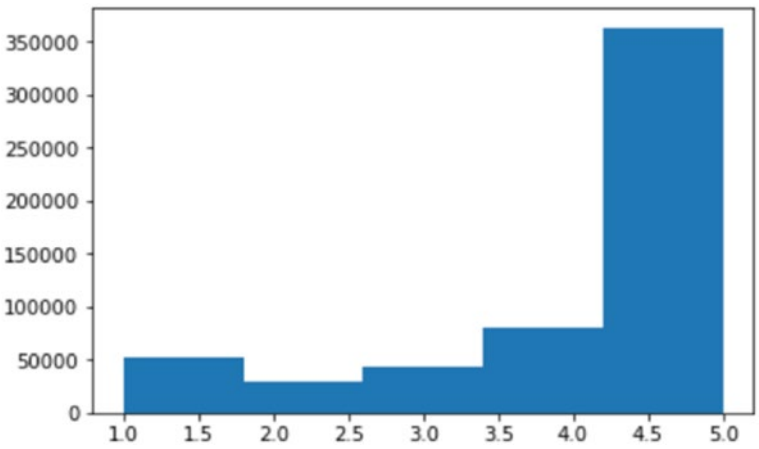
```
# Removal of stop words
stop = stopwords.words('english')
df['Text'] = df['Text'].apply(lambda x: " ".join(x for x in
x.split() if x not in stop))
df.Text.head(5)
# Output
0    bought several vitality canned dog food produc...
1    product arrived labeled jumbo salted peanutsth...
2    confection around centuries light pillowy citr...
3    looking secret ingredient robitussin believe f...
4    great taffy great price wide assortment yummy ...

# Spelling correction
df['Text'] = df['Text'].apply(lambda x: str(TextBlob(x).
correct()))
df.Text.head(5)
# Output
0    bought several vitality canned dog food produc...
1    product arrived labelled lumbo halted peanutst...
2    connection around centuries light pillow citie...
3    looking secret ingredient robitussin believe f...
4    great staff great price wide assortment mummy ...

# Lemmatization
df['Text'] = df['Text'].apply(lambda x: " ".join([Word(word).
lemmatize() for word in x.split()]))
df.Text.head(5)
# Output
0    bought several vitality canned dog food produc...
1    product arrived labelled lumbo halted peanutst...
2    connection around century light pillow city ge...
3    looking secret ingredient robitussin believe f...
4    great staff great price wide assortment mummy ...
```

Step 2-4 Exploratory data analysis

This step is not connected anywhere in predicting sentiments; what we are trying to do here is to dig deeper into the data and understand it.

```
# Create a new data frame "reviews" to perform exploratory data
analysis upon that
reviews = df
# Dropping null values
reviews.dropna(inplace=True)
```

```
# The histogram reveals this dataset is highly unbalanced
towards high rating.
reviews.Score.hist(bins=5,grid=False)
plt.show()
print(reviews.groupby('Score').count().Id)
```




```
# To make it balanced data, we sampled each score by the lowest
n-count from above. (i.e. 29743 reviews scored as '2')
score_1 = reviews[reviews['Score'] == 1].sample(n=29743)
score_2 = reviews[reviews['Score'] == 2].sample(n=29743)
```

```
score_3 = reviews[reviews['Score'] == 3].sample(n=29743)
score_4 = reviews[reviews['Score'] == 4].sample(n=29743)
score_5 = reviews[reviews['Score'] == 5].sample(n=29743)
# Here we recreate a 'balanced' dataset.
reviews_sample = pd.concat([score_1,score_2,score_3,score_4,
score_5],axis=0)
reviews_sample.reset_index(drop=True,inplace=True)
```

You can use this dataset if you are training your own sentiment classifier from scratch. And to do this. you can follow the same steps as in text classification (Recipe 5-1). Here our target variable would be positive, negative, and neutral created using score.

- Score <= 2: Negative
- Score = 3: Neutral
- Score > =4: Positive

Having said that, let's get back to our exploratory data analysis.

```
# Printing count by 'Score' to check dataset is now balanced.
print(reviews_sample.groupby('Score').count().Id)
# Output
Score
1    29743
2    29743
3    29743
4    29743
5    29743

# Let's build a word cloud looking at the 'Summary'  text
from wordcloud import WordCloud
from wordcloud import STOPWORDS
```

```
# Wordcloud function's input needs to be a single string of text.
# Here I'm concatenating all Summaries into a single string.
# similarly you can build for Text column
reviews_str = reviews_sample.Summary.str.cat()
wordcloud = WordCloud(background_color='white').
generate(reviews_str)
plt.figure(figsize=(10,10))
plt.imshow(wordcloud,interpolation='bilinear')
plt.axis("off")
plt.show()
```


```
# Now let's split the data into Negative (Score is 1 or 2) and
Positive (4 or #5) Reviews.
negative_reviews = reviews_sample[reviews_sample['Score'].
                   isin([1,2]) ]
positive_reviews = reviews_sample[reviews_sample['Score'].
                   isin([4,5]) ]
# Transform to single string
negative_reviews_str = negative_reviews.Summary.str.cat()
positive_reviews_str = positive_reviews.Summary.str.cat()
# Create wordclouds
```

```
wordcloud_negative = WordCloud(background_color='white').generate
                     (negative_reviews_str)
wordcloud_positive = WordCloud(background_color='white').generate
                     (positive_reviews_str)
# Plot
fig = plt.figure(figsize=(10,10))
ax1 = fig.add_subplot(211)
ax1.imshow(wordcloud_negative,interpolation='bilinear')
ax1.axis("off")
ax1.set_title('Reviews with Negative Scores',fontsize=20)
```


```
ax2 = fig.add_subplot(212)
ax2.imshow(wordcloud_positive,interpolation='bilinear')
ax2.axis("off")
ax2.set_title('Reviews with Positive Scores',fontsize=20)
plt.show()
#output
```

Step 2-5 Feature engineering

This step is not required as we are not building the model from scratch; rather we are using the pretrained model from the library `vaderSentiment`.

If you want to build the model from scratch, you can leverage the above positive and negative classes created while exploring as a target variable and then training the model. You can follow the same steps as text classification explained in Recipe 5-1 to build a sentiment classifier from scratch.

Step 2-6 Sentiment scores

Sentiment Analysis: Pretrained model takes the input from the text description and outputs the sentiment score ranging from -1 to +1 for each sentence.

```
#Importing required libraries
import pandas as pd
```

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import re
import os
import sys
import ast
plt.style.use('fivethirtyeight')
# Function for getting the sentiment
cp = sns.color_palette()
from vaderSentiment.vaderSentiment import
SentimentIntensityAnalyzer
analyzer = SentimentIntensityAnalyzer()

# Generating sentiment for all the sentence present in the
dataset
emptyline=[]
for row in df['Text']:
    vs=analyzer.polarity_scores(row)
    emptyline.append(vs)

# Creating new dataframe with sentiments
df_sentiments=pd.DataFrame(emptyline)
df_sentiments.head(5)

# Output
      compound    neg    neu    pos
0     0.9413      0.000  0.503  0.497
1    -0.5719      0.258  0.644  0.099
2     0.8031      0.133  0.599  0.268
3     0.4404      0.000  0.854  0.146
4     0.9186      0.000  0.455  0.545
```

```
# Merging the sentiments back to reviews dataframe
df_c = pd.concat([df.reset_index(drop=True), d], axis=1)
df_c.head(3)
#output sample
```

	ProfileName	HelpfulnessNumerator	HelpfulnessDenominator	Score	Time	Summary	Text	compound	neg	neu	pos
JHU8GW	delmartian	1	1	5	1303862400	Good Quality Dog Food	bought several vitality canned dog food produc...	0.9413	0.000	0.503	0.497
VE5NK	dll pa	0	0	1	1346976000	Not as Advertised	product arrived labelled lumbo halted peanutst...	-0.5719	0.258	0.644	0.099
0XAIN	Natalia Corres "Natalia Corres"	1	1	4	1219017600	"Delight" says it all	connection around century light pillow city ge...	0.8031	0.133	0.599	0.268

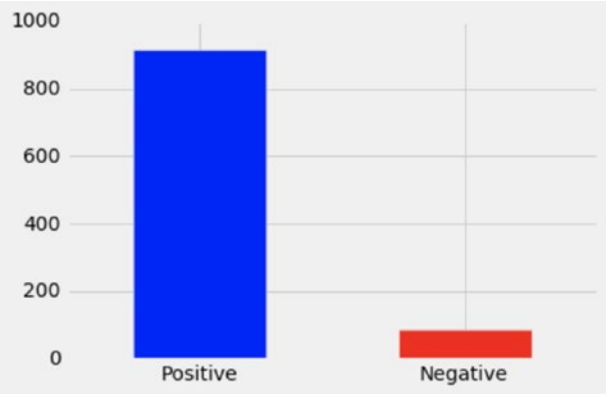
```
# Convert scores into positive and negetive sentiments using
some threshold
df_c['Sentiment'] = np.where(df_c['compound'] >= 0 , 'Positive',
                    'Negative')
df_c.head(5)
#output sample
```

rofileName	HelpfulnessNumerator	HelpfulnessDenominator	Score	Time	Summary	Text	compound	neg	neu	pos	Sentiment
elmartian	1	1	5	1303862400	Good Quality Dog Food	bought several vitality canned dog food produc...	0.9413	0.000	0.503	0.497	Positive
ll pa	0	0	1	1346976000	Not as Advertised	product arrived labelled lumbo halted peanutst...	-0.5719	0.258	0.644	0.099	Negative
atalia orres Natalia orres"	1	1	4	1219017600	"Delight" says it all	connection around century light pillow city ge...	0.8031	0.133	0.599	0.268	Positive

Step 2-7 Business insights

Let's see how the overall sentiment is using the sentiment we generated.

```
result=df_c['Sentiment'].value_counts()
result.plot(kind='bar', rot=0,color='br');
```



We just took a sample of 1000 reviews and completed sentiment analysis. If you look, more than 900 (>90%) reviews are positive, which is really good for any business.

We can also group by-products, that is, sentiments by-products to understand the high-level customer feedback against products.

```
#Sample code snippet
result=df_c.groupby('ProductId')['Sentiment'].value_counts().
unstack()
result[['Negative','Positive']].plot(kind='bar',
rot=0,color='rb')
```

Similarly, we can analyze sentiments by month using the time column and many other such attributes.

Recipe 5-3. Applying Text Similarity Functions

This recipe covers data stitching using text similarity.

Problem

We will have multiple tables in the database, and sometimes there won't be a common "ID" or "KEY" to join them – scenarios like the following:

- Customer information scattered across multiple tables and systems.
- No global key to link them all together.
- A lot of variations in names and addresses.

Solution

This can be solved by applying text similarity functions on the demographic's columns like the first name, last name, address, etc. And based on the similarity score on a few common columns, we can decide either the record pair is a match or not a match.

How It Works

Let's follow the steps in this section to link the records.

Technical challenge:

- Huge records that need to be linked/stitched/deduplicated.
- Records come from various systems with differing schemas.

There is no global key or customer id to merge. There are two possible scenarios of data stitching or linking records:

- Multiple records of the same customer at the same table, and you want to dedupe.
- Records of same customers from multiple tables need to be merged.

For Recipe 3-A, let's solve scenario 1 that is deduplication and as a part of Recipe 3-B, let's solve scenario 2 that is record linkage from multiple tables.

Deduplication in the same table

Step 3A-1 Read and understand the data

We need the data first:

```
# Import package
!pip install recordlinkage
import recordlinkage

#For this demo let us use the inbuilt dataset from
recordlinkage library
#import data set
from recordlinkage.datasets import load_febrl1

#create a dataframe - dfa
dfA = load_febrl1()
dfA.head()

#output
```

	given_name	surname	street_number	address_1	address_2	suburb	postcode	state	date_of_birth	soc_sec_id
rec_id										
rec-223-org	NaN	waller	6	tullaroop street	willaroo	st james	4011	wa	19081209	6988048
rec-122-org	lachlan	berry	69	giblin street	killarney	bittern	4814	qld	19990219	7364009
rec-373-org	deakin	sondergeld	48	goldfinch circuit	kooltuo	canterbury	2776	vic	19600210	2635962
rec-10-dup-0	kayla	harrington	NaN	maltby circuit	coaling	coolaroo	3465	nsw	19150612	9004242
rec-227-org	luke	purdon	23	ramsay place	mirani	garbutt	2260	vic	19831024	8099933

Step 3A-2 Blocking

Here we reduce the comparison window and create record pairs. Why?

- Suppose there are huge records say, 100M records means (100M choose 2) ≈ 10^16 possible pairs
- Need heuristic to quickly cut that 10^16 down without losing many matches

This can be accomplished by extracting a "blocking key" How? Example:

- Record: first name: John, last name: Roberts, address: 20 Main St Plainville MA 01111
- Blocking key: first name - John
- Will be paired with: John Ray … 011
- Won't be paired with: Frank Sinatra … 07030
- Generate pairs only for records in the same block

Below is the blocking example at a glance: here blocking is done on the "Sndx-SN," column which is nothing but the Soundex value of the surname column as discussed in the previous chapter.

Database A – Blocking information

RecID	Surname	Sndx-SN	Postcode	F3D-PC
a1	smith	s530	2602	260
a2	neighan	n250	2604	260
a3	meier	m600	2050	205
a4	smithers	s536	2012	201
a5	nguyen	n250	2022	202
a6	faulkner	f425	2037	203
a7	sandy	s530	2713	271

Database B – Blocking information

RecID	Surname	Sndx-SN	Postcode	F3D-PC
b1	meier	m600	2000	200
b2	meier	m600	2604	260
b3	smith	s530	2619	261
b4	nguyen	n250	2002	200
b5	fawkner	f256	2037	203
b6	santi	s530	2113	211
b7	cain	c500	2020	202

Candidate record pairs generated from Surname blocking

BKVs	Candidate record pairs
m600	(a3, b1), (a3, b2)
n250	(a2, b4), (a5, b4)
s530	(a1, b3), (a1, b6), (a7, b3), (a7, b6)

Candidate record pairs generated from Postcode blocking

BKVs	Candidate record pairs
202	(a5, b7)
203	(a6, b5)
260	(a1, b2), (a2, b2)

(a1, b2)
(a1, b3)
(a1, b6)
(a2, b2)
(a2, b4)
(a3, b1)
(a3, b2)
(a5, b4)
(a5, b7)
(a6, b5)
(a7, b3)
(a7, b6)

There are many advanced blocking techniques, also, like the following:

- Standard blocking
  - Single column
  - Multiple columns
- Sorted neighborhood
- Q-gram: fuzzy blocking
- LSH
- Canopy clustering

This can be a new topic altogether, but for now, let's build the pairs using the first name as the blocking index.

```
indexer = recordlinkage.BlockIndex(on='given_name')
pairs = indexer.index(dfA)
print (len(pairs))
#output
2082
```

Step 3A-3 Similarity matching and scoring

Here we compute the similarity scores on the columns like given name, surname, and address between the record pairs generated in the previous step. For columns like date of birth, suburb, and state, we are using the exact match as it is important for this column to possess exact records.

We are using `jarowinkler`, but you can use any of the other similarity measures discussed in Chapter 4.

```
# This cell can take some time to compute.
compare_cl = recordlinkage.Compare()

compare_cl.string('given_name', 'given_name',method='jarowinkler',
label='given_name')
compare_cl.string('surname', 'surname', method='jarowinkler',
label='surname')
compare_cl.exact('date_of_birth', 'date_of_birth', label='date_
of_birth')
compare_cl.exact('suburb', 'suburb', label='suburb')
compare_cl.exact('state', 'state', label='state')
compare_cl.string('address_1', 'address_1',method='jarowinkler',
label='address_1')
features = compare_cl.compute(pairs, dfA)
features.sample(5)

#output
```

		given_name	surname	date_of_birth	suburb	state	address_1
rec_id	rec_id						
rec-115-dup-0	rec-120-dup-0	1.0	0.458333	0	0	0	0.548693
rec-245-dup-0	rec-331-org	1.0	0.000000	0	0	0	0.567617
rec-455-dup-0	rec-95-dup-0	1.0	0.561905	0	0	0	0.438095
rec-462-dup-0	rec-462-org	1.0	0.961905	1	0	1	1.000000
rec-132-org	rec-30-dup-0	1.0	0.455556	0	0	0	0.571429

So here record "rec-115-dup-0" is compared with "rec-120-dup-0." Since their first name (blocking column) is matching, similarity scores are calculated on the common columns for these pairs.

Step 3A-4 Predicting records match or do not match using ECM – classifier

Here is an unsupervised learning method to calculate the probability that the records match.

```
# select all the features except for given_name since its our
blocking key
features1 = features[['suburb','state','surname','date_of_
birth','address_1']]
# Unsupervised learning - probabilistic
ecm = recordlinkage.ECMClassifier()
result_ecm = ecm.learn((features1).astype(int),return_type =
'series')
result_ecm
#output
rec_id rec_id
rec-122-org rec-183-dup-0 0
 rec-248-org 0
 rec-469-org 0
 rec-74-org 0
 rec-183-org 0
 rec-360-dup-0 0
 rec-248-dup-0 0
 rec-469-dup-0 0
rec-183-dup-0 rec-248-org 0
 rec-469-org 0
```

```
 rec-74-org 0
 rec-183-org 1
 rec-360-dup-0 0
 rec-248-dup-0 0
 rec-469-dup-0 0
rec-248-org rec-469-org 0
 rec-74-org 0
 rec-360-dup-0 0
 rec-469-dup-0 0
rec-122-dup-0 rec-122-org 1
 rec-183-dup-0 0
 rec-248-org 0
 rec-469-org 0
 rec-74-org 0
 rec-183-org 0
 rec-360-dup-0 0
 rec-248-dup-0 0
 rec-469-dup-0 0
rec-469-org rec-74-org 0
rec-183-org rec-248-org 0
 ..
rec-208-dup-0 rec-208-org 1
rec-363-dup-0 rec-363-org 1
rec-265-dup-0 rec-265-org 1
rec-315-dup-0 rec-315-org 1
rec-410-dup-0 rec-410-org 1
rec-290-org rec-93-org 0
rec-460-dup-0 rec-460-org 1
rec-499-dup-0 rec-499-org 1
rec-11-dup-0 rec-11-org 1
rec-97-dup-0 rec-97-org 1
```

```
rec-213-dup-0 rec-421-dup-0 0
rec-349-dup-0 rec-376-dup-0 0
rec-371-dup-0 rec-371-org 1
rec-129-dup-0 rec-129-org 1
rec-462-dup-0 rec-462-org 1
rec-328-dup-0 rec-328-org 1
rec-308-dup-0 rec-308-org 1
rec-272-org rec-308-dup-0 0
 rec-308-org 0
rec-5-dup-0 rec-5-org 1
rec-407-dup-0 rec-407-org 1
rec-367-dup-0 rec-367-org 1
rec-103-dup-0 rec-103-org 1
rec-195-dup-0 rec-195-org 1
rec-184-dup-0 rec-184-org 1
rec-252-dup-0 rec-252-org 1
rec-48-dup-0 rec-48-org 1
rec-298-dup-0 rec-298-org 1
rec-282-dup-0 rec-282-org 1
rec-327-org rec-411-org 0
```

The output clearly shows that "rec-183-dup-0" matches "rec-183-org" and can be linked to one `global_id`. What we have done so far is deduplication: identifying multiple records of the same users from the individual table.

Records of same customers from multiple tables

Next, let us look at how we can solve this problem if records are in multiple tables without unique ids to merge with.

Step 3B-1 Read and understand the data

Let us use the built-in dataset from the recordlinkage library:

```
from recordlinkage.datasets import load_febrl4
dfA, dfB = load_febrl4()
dfA.head()
```

```
#output
```

	given_name	surname	street_number	address_1	address_2	suburb	postcode	state	date_of_birth	soc_sec_id
rec_id										
rec-1070-org	michaela	neumann	8	stanley street	miami	winston hills	4223	nsw	19151111	5304218
rec-1016-org	courtney	painter	12	pinkerton circuit	bega flats	richlands	4560	vic	19161214	4066625
rec-4405-org	charles	green	38	salkauskas crescent	kela	dapto	4566	nsw	19480930	4365168
rec-1288-org	vanessa	parr	905	macquoid place	broadbridge manor	south grafton	2135	sa	19951119	9239102
rec-3585-org	mikayla	malloney	37	randwick road	avalind	hoppers crossing	4552	vic	19860208	7207688

```
dfB.head()
#output
```

	given_name	surname	street_number	address_1	address_2	suburb	postcode	state	date_of_birth	soc_sec_id
rec_id										
rec-561-dup-0	elton	NaN	3	light setreet	pinehill	windermere	3212	vic	19651013	1551941
rec-2642-dup-0	mitchell	maxon	47	edkins street	lochaoair	north ryde	3355	nsw	19390212	8859999
rec-608-dup-0	NaN	white	72	lambrigg street	kelgoola	broadbeach waters	3159	vic	19620216	9731855
rec-3239-dup-0	elk i	menzies	1	lyster place	NaN	northwood	2585	vic	19980624	4970481
rec-2886-dup-0	NaN	garanggar	NaN	may maxwell crescent	springettst arcade	forest hill	2342	vic	19921016	1366884

Step 3B-2 Blocking – to reduce the comparison window and creating record pairs

This is the same as explained previously, considering the given_name as a blocking index:

```
indexer = recordlinkage.BlockIndex(on='given_name')
pairs = indexer.index(dfA, dfB)
```

Step 3B-3 Similarity matching

The explanation remains the same.

```
compare_cl = recordlinkage.Compare()
compare_cl.string('given_name', 'given_name',method='jarowinkler',
label='given_name')
compare_cl.string('surname', 'surname', method='jarowinkler',
label='surname')
compare_cl.exact('date_of_birth', 'date_of_birth', label='date_
of_birth')
compare_cl.exact('suburb', 'suburb', label='suburb')
compare_cl.exact('state', 'state', label='state')
compare_cl.string('address_1', 'address_1',method='jarowinkler',
label='address_1')
features = compare_cl.compute(pairs, dfA, dfB)
features.head(10)
#output
```

		given_name	surname	date_of_birth	suburb	state	address_1
rec_id	rec_id						
rec-1070-org	rec-3024-dup-0	1.0	0.436508	0	0	1	0.000000
	rec-2371-dup-0	1.0	0.490079	0	0	0	0.715873
	rec-4652-dup-0	1.0	0.490079	0	0	0	0.645604
	rec-4795-dup-0	1.0	0.000000	0	0	1	0.552381
	rec-1314-dup-0	1.0	0.000000	0	0	1	0.618254
rec-2371-org	rec-3024-dup-0	1.0	0.527778	0	0	0	0.000000
	rec-2371-dup-0	1.0	1.000000	1	1	1	1.000000
	rec-4652-dup-0	1.0	0.500000	0	0	1	0.635684
	rec-4795-dup-0	1.0	0.527778	0	0	0	0.411111
	rec-1314-dup-0	1.0	0.527778	0	0	0	0.672222

So here record "rec-1070-org" is compared with "rec-3024-dup-0," "rec-2371-dup-0," "rec-4652-dup-0," "rec-4795-dup-0," and "rec-1314-dup-0, since their first name (blocking column) is matching and similarity scores are calculated on the common columns for these pairs.

Step 3B-4 Predicting records match or do not match using ECM – classifier

Here is an unsupervised learning method to calculate the probability that the record is a match.

```
# select all the features except for given_name since its our
blocking key
features1 = features[['suburb','state','surname','date_of_birth',
'address_1']]
# unsupervised learning - probablistic
ecm = recordlinkage.ECMClassifier()
result_ecm = ecm.learn((features1).astype(int),return_type =
'series')
```

```
result_ecm
#output sample
rec_id        rec_id
rec-1070-org  rec-3024-dup-0    0
              rec-2371-dup-0    0
              rec-4652-dup-0    0
              rec-4795-dup-0    0
              rec-1314-dup-0    0
rec-2371-org  rec-3024-dup-0    0
              rec-2371-dup-0    1
              rec-4652-dup-0    0
              rec-4795-dup-0    0
              rec-1314-dup-0    0
rec-3582-org  rec-3024-dup-0    0
              rec-2371-dup-0    0
              rec-4652-dup-0    0
              rec-4795-dup-0    0
              rec-1314-dup-0    0
rec-3024-org  rec-3024-dup-0    1
              rec-2371-dup-0    0
              rec-4652-dup-0    0
              rec-4795-dup-0    0
              rec-1314-dup-0    0
rec-4652-org  rec-3024-dup-0    0
              rec-2371-dup-0    0
              rec-4652-dup-0    1
              rec-4795-dup-0    0
              rec-1314-dup-0    0
rec-4795-org  rec-3024-dup-0    0
              rec-2371-dup-0    0
              rec-4652-dup-0    0
              rec-4795-dup-0    1
```

```
              rec-1314-dup-0    0
                                ..
rec-2820-org  rec-2820-dup-0    1
              rec-991-dup-0     0
rec-1984-org  rec-1984-dup-0    1
rec-1662-org  rec-1984-dup-0    0
rec-4415-org  rec-1984-dup-0    0
rec-1920-org  rec-1920-dup-0    1
rec-303-org   rec-303-dup-0     1
rec-1915-org  rec-1915-dup-0    1
rec-4739-org  rec-4739-dup-0    1
              rec-4865-dup-0    0
rec-681-org   rec-4276-dup-0    0
rec-4603-org  rec-4848-dup-0    0
              rec-4603-dup-0    1
rec-3122-org  rec-4848-dup-0    0
              rec-4603-dup-0    0
rec-3711-org  rec-3711-dup-0    1
rec-4912-org  rec-4912-dup-0    1
rec-664-org   rec-664-dup-0     1
              rec-1311-dup-0    0
rec-4031-org  rec-4031-dup-0    1
rec-1413-org  rec-1413-dup-0    1
rec-735-org   rec-735-dup-0     1
rec-1361-org  rec-1361-dup-0    1
rec-3090-org  rec-3090-dup-0    1
rec-2571-org  rec-2571-dup-0    1
rec-4528-org  rec-4528-dup-0    1
rec-4887-org  rec-4887-dup-0    1
rec-4350-org  rec-4350-dup-0    1
rec-4569-org  rec-4569-dup-0    1
rec-3125-org  rec-3125-dup-0    1
```

The output clearly shows that "rec-122-dup-0" matches "rec-122-org" and can be linked to one global_id.

In this way, you can create a data lake consisting of a unique global id and consistent data across tables and also perform any kind of statistical analysis.

Recipe 5-4. Summarizing Text Data

If you just look around, there are lots of articles and books available. Let's assume you want to learn a concept in NLP and if you Google it, you will find an article. You like the content of the article, but it's too long to read it one more time. You want to basically summarize the article and save it somewhere so that you can read it later.

Well, NLP has a solution for that. Text summarization will help us do that. You don't have to read the full article or book every time.

Problem

Text summarization of article/document using different algorithms in Python.

Solution

Text summarization is the process of making large documents into smaller ones without losing the context, which eventually saves readers time. This can be done using different techniques like the following:

- TextRank: A graph-based ranking algorithm
- Feature-based text summarization
- LexRank: TF-IDF with a graph-based algorithm
- Topic based
- Using sentence embeddings
- Encoder-Decoder Model: Deep learning techniques

How It Works

We will explore the first 2 approaches in this recipe and see how it works.

Method 4-1 TextRank

TextRank is the graph-based ranking algorithm for NLP. It is basically inspired by PageRank, which is used in the Google search engine but particularly designed for text. It will extract the topics, create nodes out of them, and capture the relation between nodes to summarize the text.

Let's see how to do it using the Python package Gensim. "Summarize" is the function used.

Before that, let's import the notes. Let's say your article is Wikipedia for the topic of Natural language processing.

```
# Import BeautifulSoup and urllib libraries to fetch data from
Wikipedia.
from bs4 import BeautifulSoup
from urllib.request import urlopen

# Function to get data from Wikipedia
def get_only_text(url):
 page = urlopen(url)
 soup = BeautifulSoup(page)
 text = ' '.join(map(lambda p: p.text, soup.find_all('p')))
 print (text)
 return soup.title.text, text

# Mention the Wikipedia url
url="https://en.wikipedia.org/wiki/Natural_language_processing"
# Call the function created above
text = get_only_text(url)

# Count the number of letters
```

```
len(".join(text))
```

```
Result:
Out[74]: 8519
# Lets see first 1000 letters from the text
text[:1000]
```

```
Result :
Out[72]: '(\'Natural language processing - Wikipedia\',
\'Natural language processing (NLP) is an area of computer
science and artificial intelligence concerned with the
interactions between computers and human (natural) languages,
in particular how to program computers to process and analyze
large amounts of natural language\\xa0data.\\n Challenges
in natural language processing frequently involve speech
recognition, natural language understanding, and natural
language generation.\\n The history of natural language
processing generally started in the 1950s, although work can be
found from earlier periods.\\nIn 1950, Alan Turing published
an article titled "Intelligence" which proposed what is now
called the Turing test as a criterion of intelligence.\\n
The Georgetown experiment in 1954 involved fully automatic
translation of more than sixty Russian sentences into English.
The authors claimed that within three or five years, machine
translation would be a solved problem.[2]  However, real
progress was '
```

```
# Import summarize from gensim
from gensim.summarization.summarizer import summarize
from gensim.summarization import keywords
# Convert text to string format
text = str(text)
```

```
#Summarize the text with ratio 0.1 (10% of the total words.)
summarize(text, ratio=0.1)
Result:
Out[77]: 'However, part-of-speech tagging introduced the use
of hidden Markov models to natural language processing, and
increasingly, research has focused on statistical models,
which make soft, probabilistic decisions based on attaching
real-valued weights to the features making up the input data.
\nSuch models are generally more robust when given unfamiliar
input, especially input that contains errors (as is very
common for real-world data), and produce more reliable results
when integrated into a larger system comprising multiple
subtasks.\\n Many of the notable early successes occurred in
the field of machine translation, due especially to work at
IBM Research, where successively more complicated statistical
models were developed.'
```

That's it. The generated summary is as simple as that. If you read this summary and whole article, it's close enough. But still, there is a lot of room for improvement.

```
#keywords
print(keywords(text, ratio=0.1))

Result:
learning
learn
languages
process
systems
worlds
world
real
```

```
natural language processing
research
researched
results
result
data
statistical
hand
generation
generally
generic
general
generated
tasks
task
large
human
intelligence
input
called
calling
calls
produced
produce
produces
producing
possibly
possible
corpora
base
based
```

Method 4-2 Feature-based text summarization

Your feature-based text summarization methods will extract a feature from the sentence and check the importance to rank it. Position, length, term frequency, named entity, and many other features are used to calculate the score.

Luhn's Algorithm is one of the feature-based algorithms, and we will see how to implement it using the sumy library.

```
# Install sumy
!pip install sumy
# Import the packages
from sumy.parsers.html import HtmlParser
from sumy.parsers.plaintext import PlaintextParser
from sumy.nlp.tokenizers import Tokenizer
from sumy.summarizers.lsa import LsaSummarizer
from sumy.nlp.stemmers import Stemmer
from sumy.utils import get_stop_words
from sumy.summarizers.luhn import LuhnSummarizer

# Extracting and summarizing
LANGUAGE = "english"
SENTENCES_COUNT = 10

url="https://en.wikipedia.org/wiki/Natural_language_processing"
parser = HtmlParser.from_url(url, Tokenizer(LANGUAGE))
summarizer = LsaSummarizer()
summarizer = LsaSummarizer(Stemmer(LANGUAGE))
summarizer.stop_words = get_stop_words(LANGUAGE)
for sentence in summarizer(parser.document, SENTENCES_COUNT):
    print(sentence)
Result :
```

[2] However, real progress was much slower, and after the ALPAC report in 1966, which found that ten-year-long research had failed to fulfill the expectations, funding for machine translation was dramatically reduced.
However, there is an enormous amount of non-annotated data available (including, among other things, the entire content of the World Wide Web), which can often make up for the inferior results if the algorithm used has a low enough time complexity to be practical, which some such as Chinese Whispers do.
Since the so-called "statistical revolution"
in the late 1980s and mid 1990s, much natural language processing research has relied heavily on machine learning .
Increasingly, however, research has focused on statistical models , which make soft, probabilistic decisions based on attaching real-valued weights to each input feature.
Natural language understanding Convert chunks of text into more formal representations such as first-order logic structures that are easier for computer programs to manipulate.
[18] ^ Implementing an online help desk system based on conversational agent Authors: Alisa Kongthon, Chatchawal Sangkeettrakarn, Sarawoot Kongyoung and Choochart Haruechaiyasak.
[self-published source] ^ Chomskyan linguistics encourages the investigation of " corner cases " that stress the limits of its theoretical models (comparable to pathological phenomena in mathematics), typically created using thought experiments , rather than the systematic investigation of typical phenomena that occur in real-world data, as is the case in corpus linguistics .

^ Antonio Di Marco - Roberto Navigili, "Clustering and Diversifying Web Search Results with Graph Based Word Sense Induction" , 2013 Goldberg, Yoav (2016).
Scripts, plans, goals, and understanding: An inquiry into human knowledge structures ^ Kishorjit, N., Vidya Raj RK., Nirmal Y., and Sivaji B.
^ PASCAL Recognizing Textual Entailment Challenge (RTE-7) https://tac.nist.gov//2011/RTE/ ^ Yi, Chucai; Tian, Yingli (2012), "Assistive Text Reading from Complex Background for Blind Persons" , Camera-Based Document Analysis and Recognition , Springer Berlin Heidelberg, pp.

Problem solved. Now you don't have to read the whole notes; just read the summary whenever we are running low on time.

We can use many of the deep learning techniques to get better accuracy and better results like the Encoder-Decoder Model. We will see how to do that in the next chapter.

Recipe 5-5. Clustering Documents

Document clustering, also called text clustering, is a cluster analysis on textual documents. One of the typical usages would be document management.

Problem

Clustering or grouping the documents based on the patterns and similarities.

Solution

Document clustering yet again includes similar steps, so let's have a look at them:

1. Tokenization
2. Stemming and lemmatization
3. Removing stop words and punctuation
4. Computing term frequencies or TF-IDF
5. Clustering: K-means/Hierarchical; we can then use any of the clustering algorithms to cluster different documents based on the features we have generated
6. Evaluation and visualization: Finally, the clustering results can be visualized by plotting the clusters into a two-dimensional space

How It Works

Step 5-1 Import data and libraries

Here are the libraries, then the data:

```
!pip install mpld3
import numpy as np
import pandas as pd
import nltk
from nltk.stem.snowball import SnowballStemmer
from bs4 import BeautifulSoup
import re
import os
import codecs
```

```
from sklearn import feature_extraction
import mpld3
from sklearn.metrics.pairwise import cosine_similarity
import os
import matplotlib.pyplot as plt
import matplotlib as mpl
from sklearn.manifold import MDS

#Lets use the same complaint dataset we use for classification
Data = pd.read_csv("/Consumer_Complaints.
csv",encoding='latin-1')
#selecting required columns and rows
Data = Data[['consumer_complaint_narrative']]
Data = Data[pd.notnull(Data['consumer_complaint_narrative'])]

# lets do the clustering for just 200 documents. Its easier to
interpret.
Data_sample=Data.sample(200)
```

Step 5-2 Preprocessing and TF-IDF feature engineering

Now we preprocess it:

```
# Remove unwanted symbol
Data_sample['consumer_complaint_narrative'] = Data_
sample['consumer_complaint_narrative'].str.replace('XXXX',")
# Convert dataframe to list
complaints = Data_sample['consumer_complaint_narrative'].tolist()
# create the rank of documents – we will use it later
ranks = []
for i in range(1, len(complaints)+1):
    ranks.append(i)
```

```
# Stop Words
stopwords = nltk.corpus.stopwords.words('english')
# Load 'stemmer'
stemmer = SnowballStemmer("english")

# Functions for sentence tokenizer, to remove numeric tokens
and raw #punctuation
def tokenize_and_stem(text):
    tokens = [word for sent in nltk.sent_tokenize(text) for
    word in nltk.word_tokenize(sent)]
    filtered_tokens = []
    for token in tokens:
        if re.search('[a-zA-Z]', token):
            filtered_tokens.append(token)
    stems = [stemmer.stem(t) for t in filtered_tokens]
    return stems
def tokenize_only(text):
    tokens = [word.lower() for sent in nltk.sent_tokenize(text)
    for word in nltk.word_tokenize(sent)]
    filtered_tokens = []
    for token in tokens:
        if re.search('[a-zA-Z]', token):
            filtered_tokens.append(token)
    return filtered_tokens
from sklearn.feature_extraction.text import TfidfVectorizer
# tfidf vectorizer
tfidf_vectorizer = TfidfVectorizer(max_df=0.8, max_features=200000,
                                   min_df=0.2, stop_words='english',
                                   use_idf=True, tokenizer=tokenize_
                                   and_stem, ngram_range=(1,3))
```

```
#fit the vectorizer to data
tfidf_matrix = tfidf_vectorizer.fit_transform(complaints)
terms = tfidf_vectorizer.get_feature_names()
print(tfidf_matrix.shape)
(200, 30)
```

Step 5-3 Clustering using K-means

Let's start the clustering:

```
#Import Kmeans
from sklearn.cluster import KMeans

# Define number of clusters
num_clusters = 6

#Running clustering algorithm
km = KMeans(n_clusters=num_clusters)
km.fit(tfidf_matrix)

#final clusters
clusters = km.labels_.tolist()
complaints_data = { 'rank': ranks, 'complaints': complaints,
                   'cluster': clusters }
frame = pd.DataFrame(complaints_data, index = [clusters] ,
columns = ['rank', 'cluster'])
#number of docs per cluster
frame['cluster'].value_counts()

0 42
1 37
5 36
3 36
2 27
4 22
```

Step 5-4 Identify cluster behavior

Identify which are the top 5 words that are nearest to the cluster centroid.

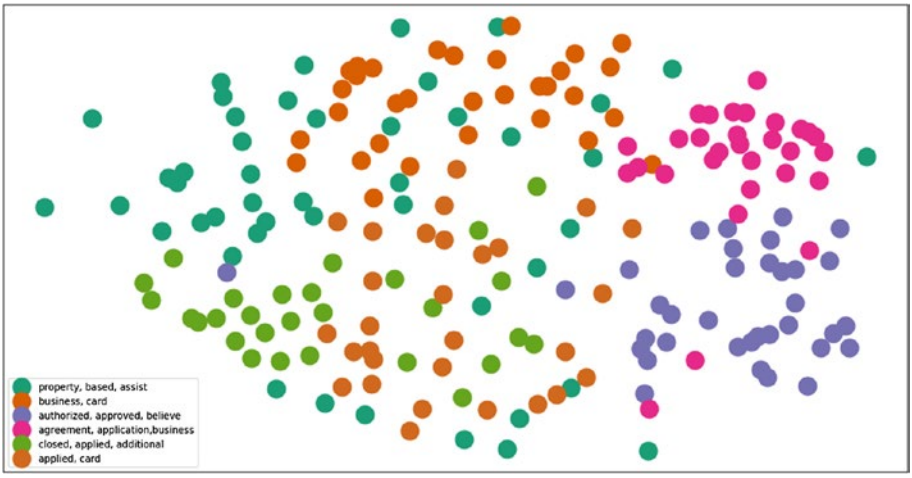
```
totalvocab_stemmed = []
totalvocab_tokenized = []
for i in complaints:
    allwords_stemmed = tokenize_and_stem(i)
    totalvocab_stemmed.extend(allwords_stemmed)

    allwords_tokenized = tokenize_only(i)
    totalvocab_tokenized.extend(allwords_tokenized)
vocab_frame = pd.DataFrame({'words': totalvocab_tokenized},
index = totalvocab_stemmed)
#sort cluster centers by proximity to centroid
order_centroids = km.cluster_centers_.argsort()[:, ::-1]
for i in range(num_clusters):
    print("Cluster %d words:" % i, end=")
    for ind in order_centroids[i, :6]:
        print(' %s' % vocab_frame.ix[terms[ind].split(' ')].
        values.tolist()[0][0].encode('utf-8', 'ignore'), end=',')
    print()
Cluster 0 words: b'needs', b'time', b'bank', b'information', b'told'
Cluster 1 words: b'account', b'bank', b'credit', b'time', b'months'
Cluster 2 words: b'debt', b'collection', b'number', b'credit', b"n't"
Cluster 3 words: b'report', b'credit', b'credit', b'account',
                 b'information'
Cluster 4 words: b'loan', b'payments', b'pay', b'months', b'state'
Cluster 5 words: b'payments', b'pay', b'told', b'did', b'credit'
```

Step 5-5 Plot the clusters on a 2D graph

Finally, we plot the clusters:

```
#Similarity
similarity_distance = 1 - cosine_similarity(tfidf_matrix)

# Convert two components as we're plotting points in a
two-dimensional plane
mds = MDS(n_components=2, dissimilarity="precomputed",
      random_state=1)
pos = mds.fit_transform(similarity_distance)  # shape
      (n_components, n_samples)
xs, ys = pos[:, 0], pos[:, 1]
#Set up colors per clusters using a dict
cluster_colors = {0: '#1b9e77', 1: '#d95f02', 2: '#7570b3',
                  3: '#e7298a', 4: '#66a61e', 5: '#D2691E'}
#set up cluster names using a dict
cluster_names = {0: 'property, based, assist',
                  1: 'business, card',
                  2: 'authorized, approved, believe',
                  3: 'agreement, application,business',
                  4: 'closed, applied, additional',
                  5: 'applied, card'}
# Finally plot it
%matplotlib inline

#Create data frame that has the result of the MDS and the cluster
df = pd.DataFrame(dict(x=xs, y=ys, label=clusters))
groups = df.groupby('label')
# Set up plot
fig, ax = plt.subplots(figsize=(17, 9)) # set size
for name, group in groups:
```

```
    ax.plot(group.x, group.y, marker='o', linestyle='', ms=20,
            label=cluster_names[name], color=cluster_colors[name],
            mec='none')
    ax.set_aspect('auto')
    ax.tick_params(\
        axis= 'x',
        which='both',
        bottom='off',
        top='off',
        labelbottom='off')
    ax.tick_params(\
        axis= 'y',
        which='both',
        left='off',
        top='off',
        labelleft='off')

ax.legend(numpoints=1)
plt.show()
```



That's it. We have clustered 200 complaints into 6 groups using K-means clustering. It basically clusters similar kinds of complaints to 6 buckets using TF-IDF. We can also use the word embeddings and solve this to achieve better clusters. 2D graphs provide a good look into the cluster's behavior and if we look, we will see that the same color dots (docs) are located closer to each other.

Recipe 5-6. NLP in a Search Engine

In this recipe, we are going to discuss what it takes to build a search engine from an NLP standpoint. Implementation of the same is beyond the scope of this book.

Problem

You want to know the architecture and NLP pipeline to build a search engine.

Solution

Figure 5-1 shows the whole process. Each step is explained in the "How It Works" section.

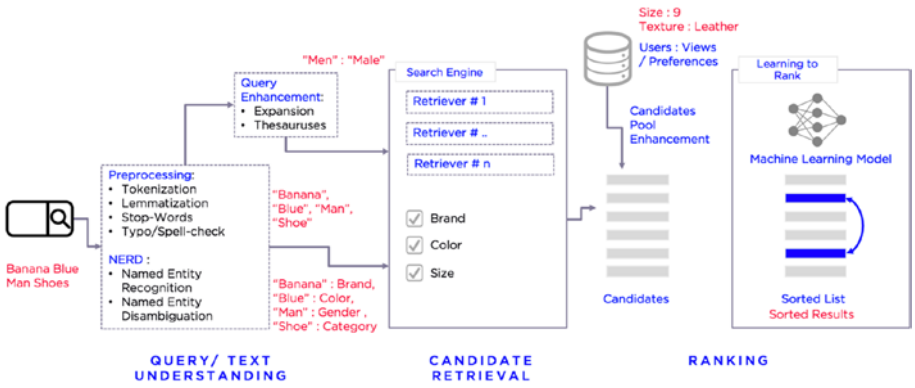


Figure 5-1. *The NLP process in a search engine*

How It Works

Let's follow and understand the above architecture step by step in this section to build the search engine from an NLP standpoint.

Step 6-1 Preprocessing

Whenever the user enters the search query, it is passed on to the NLP preprocessing pipeline:

1. Removal of noise and stop words
2. Tokenization
3. Stemming
4. Lemmatization

Step 6-2 The entity extraction model

Output from the above pipeline is fed into the entity extraction model. We can build the customized entity recognition model by using any of the libraries like StanfordNER or NLTK.

Or you can build an entity recognition model from scratch using conditional random fields or Markov models.

For example, suppose we are building a search engine for an e-commerce giant. Below are entities that we can train the model on:

- Gender
- Color
- Brand
- Product Category
- Product Type
- Price
- Size

Also, we can build named entity disambiguation using deep learning frameworks like RNN and LSTM. This is very important for the entities extractor to understand the content in which the entities are used. For example, pink can be a color or a brand. NED helps in such disambiguation.

NERD Model building steps:

- Data cleaning and preprocessing
- Training NER Model
- Testing and Validation
- Deployment

Ways to train/build NERD model:

- Named Entity Recognition and Disambiguation
- Stanford NER with customization
- Recurrent Neural Network (RNN) – LSTM (Long Short-Term Memory) to use context for disambiguation
- Joint Named Entity Recognition and Disambiguation

Step 6-3 Query enhancement/expansion

It is very important to understand the possible synonyms of the entities to make sure search results do not miss out on potential relevance. Say, for example, men's shoes can also be called as male shoes, men's sports shoes, men's formal shoes, men's loafers, men's sneakers.

Use locally-trained word embedding (using Word2Vec / GloVe Model) to achieve this.

Step 6-4 Use a search platform

Search platforms such as Solr or Elastic Search have major features that include full-text search hit highlighting, faceted search, real-time indexing, dynamic clustering, and database integration. This is not related to NLP; as an end-to-end application point of view, we have just given an introduction of what this is.

Step 6-5 Learning to rank

Once the search results are fetched from Solr or Elastic Search, they should be ranked based on the user preferences using the past behaviors.

CHAPTER 6

Deep Learning for NLP

In this chapter, we will implement deep learning for NLP:

Recipe 1. Information retrieval using deep learning

Recipe 2. Text classification using CNN, RNN, LSTM

Recipe 3. Predicting the next word/sequence of words using LSTM for Emails

Introduction to Deep Learning

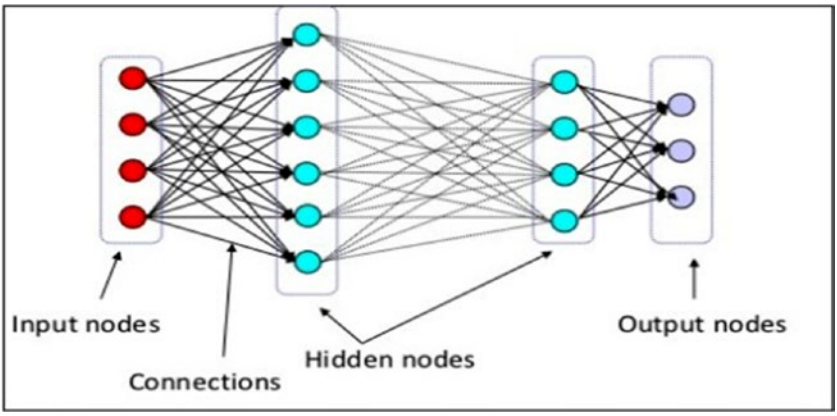
Deep learning is a subfield of machine learning that is inspired by the function of the brain. Just like how neurons are interconnected in the brain, neural networks also work the same. Each neuron takes input, does some kind of manipulation within the neuron, and produces an output that is closer to the expected output (in the case of labeled data).

What happens within the neuron is what we are interested in: to get to the most accurate results. In very simple words, it's giving weight to every input and generating a function to accumulate all these weights and pass it onto the next layer, which can be the output layer eventually.

A. Kulkarni and A. Shivananda, *Natural Language Processing Recipes*,
https://doi.org/10.1007/978-1-4842-4267-4_6

The network has 3 components:

- Input layer
- Hidden layer/layers
- Output layer



The functions can be of different types based on the problem or the data. These are also called activation functions. Below are the types.

- Linear Activation functions: A linear neuron takes a linear combination of the weighted inputs; and the output can take any value between -infinity to infinity.
- Nonlinear Activation function: These are the most used ones, and they make the output restricted between some range:
    - Sigmoid or Logit Activation Function: Basically, it scales down the output between 0 and 1 by applying a log function, which makes the classification problems easier.

- Softmax function: Softmax is almost similar to sigmoid, but it calculates the probabilities of the event over 'n' different classes, which will be useful to determine the target in multiclass classification problems.
- Tanh Function: The range of the tanh function is from (-1 to 1), and the rest remains the same as sigmoid.
- Rectified Linear Unit Activation function: ReLU converts anything that is less than zero to zero. So, the range becomes 0 to infinity.

We still haven't discussed how training is carried out in neural networks. Let's do that by taking one of the networks as an example, which is the convolutional neural network.

Convolutional Neural Networks

Convolutional Neural Networks (CNN) are similar to ordinary neural networks but have multiple hidden layers and a filter called the convolution layer. CNN is successful in identifying faces, objects, and traffic signs and also used in self-driving cars.

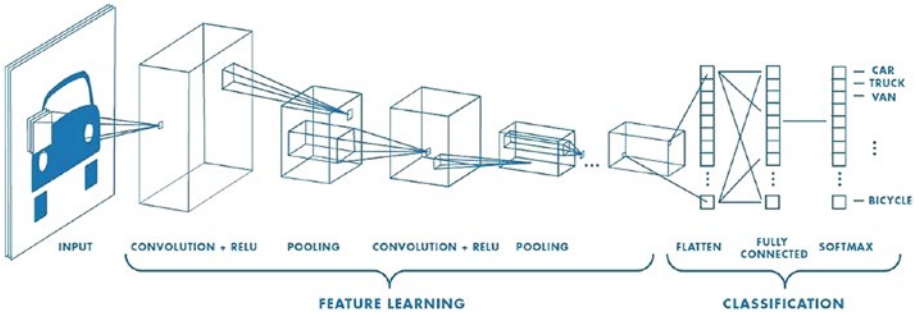
Data

As we all know, algorithms work basically on numerical data. Images and text data are unstructured data as we discussed earlier, and they need to be converted into numerical values even before we start anything.

- *Image*: Computer takes an image as an array of pixel values. Depending on the resolution and size of the image, it will see an X Y x Z array of numbers.

For example, there is a color image and its size is 480 x 480 pixels. The representation of the array will be 480 x 480 x 3 where 3 is the RGB value of the color. Each of these numbers varies from 0 to 255, which describes the pixel intensity/density at that point. The concept is that if given the computer and this array of numbers, it will output the probability of the image being a certain class in case of a classification problem.

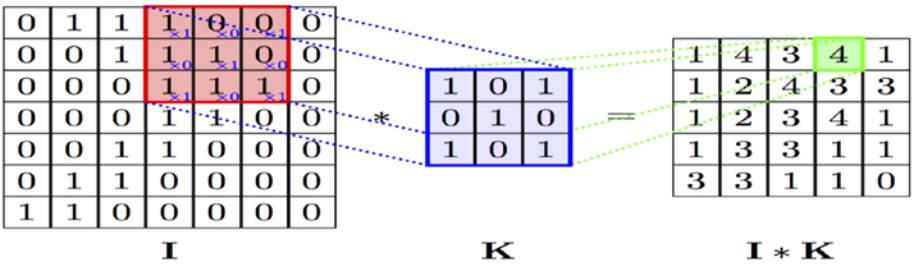
- *Text*: We already discussed throughout the book how to create features out of the text. We can use any of those techniques to convert text to features. RNN and LSTM are suited better for text-related solutions that we will discuss in the next sections.



Architecture

CNN is a special case of a neural network with an input layer, output layer, and multiple hidden layers. The hidden layers have 4 different procedures to complete the network. Each one is explained in detail.

Convolution

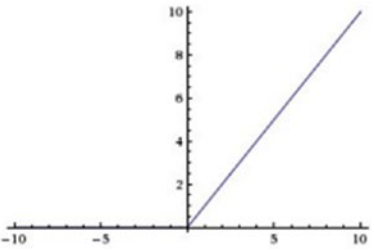


The Convolution layer is the heart of a Convolutional Neural Network, which does most of the computational operations. The name comes from the "convolution" operator that extracts features from the input image. These are also called filters (Orange color 3*3 matrix). The matrix formed by sliding the filter over the full image and calculating the dot product between these 2 matrices is called the 'Convolved Feature' or 'Activation Map' or the 'Feature Map.' Suppose that in table data, different types of features are calculated like "age" from "date of birth." The same way here also, straight edges, simple colors, and curves are some of the features that the filter will extract from the image.

During the training of the CNN, it learns the numbers or values present inside the filter and uses them on testing data. The greater the number of features, the more the image features get extracted and recognize all patterns in unseen images.

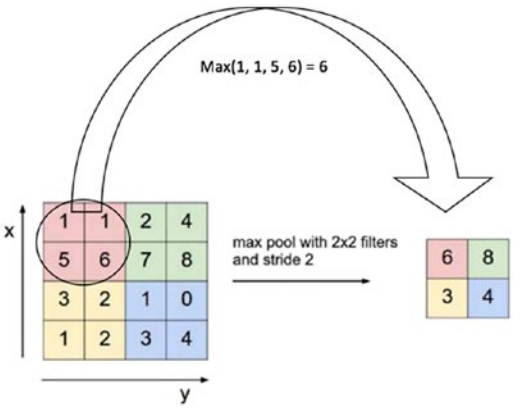
Nonlinearity (ReLU)

Output = Max(zero, Input)



ReLU (Rectified Linear Unit) is a nonlinear function that is used after a convolution layer in CNN architecture. It replaces all negative values in the matrix to zero. The purpose of ReLU is to introduce nonlinearity in the CNN to perform better.

Pooling



Pooling or subsampling is used to decrease the dimensionality of the feature without losing important information. It's done to reduce the huge number of inputs to a full connected layer and computation required to process the model. It also helps to reduce the overfitting of the model. It uses a 2 x 2 window and slides over the image and takes the maximum value in each region as shown in the figure. This is how it reduces dimensionality.

Flatten, Fully Connected, and Softmax Layers

The last layer is a dense layer that needs feature vectors as input. But the output from the pooling layer is not a 1D feature vector. This process of converting the output of convolution to a feature vector is called flattening. The Fully Connected layer takes an input from the flatten layer and gives out an N-dimensional vector where N is the number of classes.

The function of the fully connected layer is to use these features for classifying the input image into various classes based on the loss function on the training dataset. The Softmax function is used at the very end to convert these N-dimensional vectors into a probability for each class, which will eventually classify the image into a particular class.

Backpropagation: Training the Neural Network

In normal neural networks, you basically do Forward Propagation to get the output and check if this output is correct and calculate the error. In Backward Propagation, we are going backward through your network that finds the partial derivatives of the error with respect to each weight.

Let's see how exactly it works.

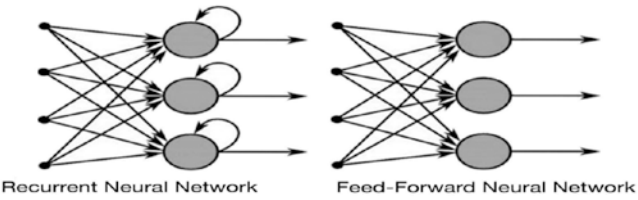
The input image is fed into the network and completes forward propagation, which is convolution, ReLU, and pooling operations with forward propagation in the fully Connected layer and generates output probabilities for each class. As per the feed forward rule, weights are randomly assigned and complete the first iteration of training and also output random probabilities. After the end of the first step, the network calculates the error at the output layer using

$$\textbf{Total Error} = \sum \tfrac{1}{2} (\textbf{target probability} - \textbf{output probability})^2$$

Now, your backpropagation starts to calculate the gradients of the error with respect to all weights in the network and use gradient descent to update all filter values and weights, which will eventually minimize the output error. Parameters like the number of filters, filter sizes, and the architecture of the network will be finalized while building your network. The filter matrix and connection weights will get updated for each run. The whole process is repeated for the complete training set until the error is minimized.

Recurrent Neural Networks

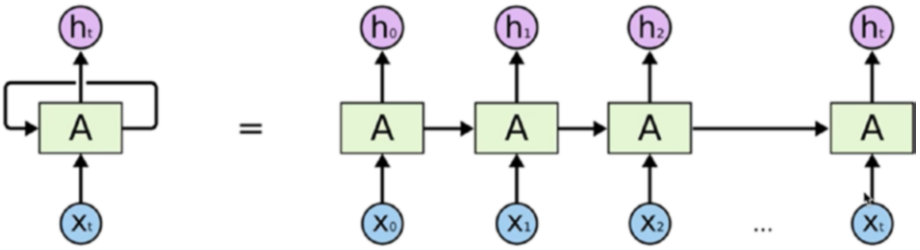
CNNs are basically used for computer vision problems but fail to solve sequence models. Sequence models are those where even a sequence of the entity also matters. For example, in the text, the order of the words matters to create meaningful sentences. This is where RNNs come into the picture and are useful with sequential data because each neuron can use its memory to remember information about the previous step.



It is quite complex to understand how exactly RNN is working. If you see the above figure, the recurrent neural network is taking the output from the hidden layer and sending it back to the same layer before giving the prediction.

Training RNN – Backpropagation Through Time (BPTT)

We know how feed forward and backpropagation work from CNN, so let's see how training is done in case of RNN.

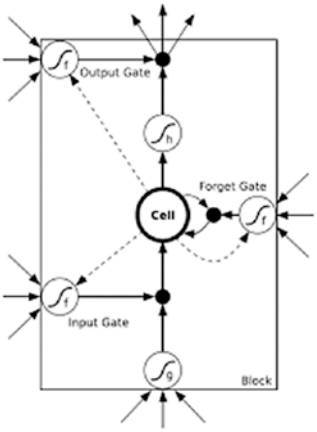


If we just discuss the hidden layer, it's not only taking input from the hidden layer, but we can also add another input to the same hidden layer. Now the backpropagation happens like any other previous training we have seen; it's just that now it is dependent on time. Here error is backpropagated from the last timestamp to the first through unrolling the hidden layers. This allows calculating the error for each timestamp and updating the weights. Recurrent networks with recurrent connections between hidden units read an entire sequence and then produce a required output.

When the values of a gradient are too small and the model takes way too long to learn, this is called Vanishing Gradients. This problem is solved by LSTMs.

Long Short-Term Memory (LSTM)

LSTMs are a kind of RNNs with betterment in equation and backpropagation, which makes it perform better. LSTMs work almost similarly to RNN, but these units can learn things with very long time gaps, and they can store information just like computers.



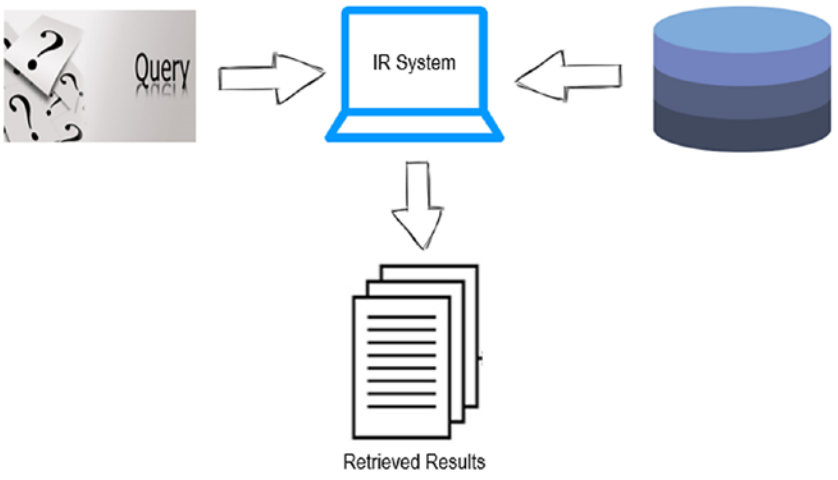
The algorithm learns the importance of the word or character through weighing methodology and decides whether to store it or not. For this, it uses regulated structures called gates that have the ability to remove or add information to the cell. These cells have a sigmoid layer that decides how much information should be passed. It has three layers, namely "input," "forget," and "output" to carry out this process.

Going in depth on how CNN and RNNs work is beyond the scope of this book. We have mentioned references at the end of the book if anyone is interested in learning about this in more depth.

Recipe 6-1. Retrieving Information

Information retrieval is one of the highly used applications of NLP and it is quite tricky. The meaning of the words or sentences not only depends on the exact words used but also on the context and meaning. Two sentences may be of completely different words but can convey the same meaning. We should be able to capture that as well.

An information retrieval (IR) system allows users to efficiently search documents and retrieve meaningful information based on a search text/query.



Problem

Information retrieval using word embeddings.

Solution

There are multiple ways to do Information retrieval. But we will see how to do it using word embeddings, which is very effective since it takes context also into consideration. We discussed how word embeddings are built in Chapter 3. We will just use the pretrained word2vec in this case.

Let's take a simple example and see how to build a document retrieval using query input. Let's say we have 4 documents in our database as below. (Just showcasing how it works. We will have too many documents in a real-world application.)

```
Doc1 = ["With the Union cabinet approving the amendments to the
Motor Vehicles Act, 2016, those caught for drunken driving will
have to have really deep pockets, as the fine payable in court
has been enhanced to Rs 10,000 for first-time offenders." ]

Doc2 = ["Natural language processing (NLP) is an area of
computer science and artificial intelligence concerned with the
interactions between computers and human (natural) languages,
in particular how to program computers to process and analyze
large amounts of natural language data."]

Doc3 = ["He points out that public transport is very good in
Mumbai and New Delhi, where there is a good network of suburban
and metro rail systems."]
```

```
Doc4 = ["But the man behind the wickets at the other end was
watching just as keenly. With an affirmative nod from Dhoni,
India captain Rohit Sharma promptly asked for a review. Sure
enough, the ball would have clipped the top of middle and leg."]
```

Assume we have numerous documents like this. And you want to retrieve the most relevant once for the query "cricket." Let's see how to build it.

```
query = "cricket"
```

How It Works

Step 1-1 Import the libraries

Here are the libraries:

```
import gensim
from gensim.models import Word2Vec
import numpy as np
import nltk
import itertools
from nltk.corpus import stopwords
from nltk.tokenize import sent_tokenize, word_tokenize
import scipy
from scipy import spatial
from nltk.tokenize.toktok import ToktokTokenizer
import re
tokenizer = ToktokTokenizer()
stopword_list = nltk.corpus.stopwords.words('english')
```

Step 1-2 Create/import documents

Randomly taking sentences from the internet:

```
Doc1 = ["With the Union cabinet approving the amendments to the
Motor Vehicles Act, 2016, those caught for drunken driving will
have to have really deep pockets, as the fine payable in court
has been enhanced to Rs 10,000 for first-time offenders." ]

Doc2 = ["Natural language processing (NLP) is an area of
computer science and artificial intelligence concerned with the
interactions between computers and human (natural) languages,
in particular how to program computers to process and analyze
large amounts of natural language data."]

Doc3 = ["He points out that public transport is very good in
Mumbai and New Delhi, where there is a good network of suburban
and metro rail systems."]

Doc4 = ["But the man behind the wickets at the other end was
watching just as keenly. With an affirmative nod from Dhoni,
India captain Rohit Sharma promptly asked for a review. Sure
enough, the ball would have clipped the top of middle and leg."]

# Put all the documents in one list

fin= Doc1+Doc2+Doc3+Doc4
```

Step 1-3 Download word2vec

As mentioned earlier, we are going to use the word embeddings to solve this problem. Download word2vec from the below link:

```
https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit
```

```
#load the model

model = gensim.models.KeyedVectors.load_word2vec_format
('/GoogleNews-vectors-negative300.bin', binary=True)
```

Step 1-4 Create IR system

Now we build the information retrieval system:

```
#Preprocessing

def remove_stopwords(text, is_lower_case=False):
    pattern = r'[^a-zA-z0-9\s]'
    text = re.sub(pattern, '', ''.join(text))
    tokens = tokenizer.tokenize(text)
    tokens = [token.strip() for token in tokens]
    if is_lower_case:
        filtered_tokens = [token for token in tokens if token
                           not in stopword_list]
    else:
        filtered_tokens = [token for token in tokens if token.
                           lower() not in stopword_list]
    filtered_text = ' '.join(filtered_tokens)
    return filtered_text

# Function to get the embedding vector for n dimension, we have
used "300"

def get_embedding(word):
    if word in model.wv.vocab:
        return model[x]
    else:
        return np.zeros(300)
```

For every document, we will get a lot of vectors based on the number of words present. We need to calculate the average vector for the document through taking a mean of all the word vectors.

```
# Getting average vector for each document
out_dict =  {}
for sen in fin:
    average_vector = (np.mean(np.array([get_embedding(x) for x
    in nltk.word_tokenize(remove_stopwords(sen))]), axis=0))
    dict = { sen : (average_vector) }
    out_dict.update(dict)

# Function to calculate the similarity between the query vector
and document vector

def get_sim(query_embedding, average_vector_doc):
    sim = [(1 - scipy.spatial.distance.cosine(query_embedding,
    average_vector_doc))]
    return sim

# Rank all the documents based on the similarity to get
relevant docs

def Ranked_documents(query):
    query_words =  (np.mean(np.array([get_embedding(x) for x in
    nltk.word_tokenize(query.lower())],dtype=float), axis=0))
    rank = []
    for k,v in out_dict.items():
        rank.append((k, get_sim(query_words, v)))
    rank = sorted(rank,key=lambda t: t[1], reverse=True)
    print('Ranked Documents :')
    return rank
```

Step 1-5 Results and applications

Let's see how the information retrieval system we built is working with a couple of examples.

```
# Call the IR function with a query

Ranked_documents("cricket")

Result :

[('But the man behind the wickets at the other end was watching
just as keenly. With an affirmative nod from Dhoni, India
captain Rohit Sharma promptly asked for a review. Sure enough,
the ball would have clipped the top of middle and leg.',
  [0.44954327116871795]),
 ('He points out that public transport is very good in Mumbai
and New Delhi, where there is a good network of suburban and
metro rail systems.',
  [0.23973446569030055]),
 ('With the Union cabinet approving the amendments to the Motor
Vehicles Act, 2016, those caught for drunken driving will have
to have really deep pockets, as the fine payable in court has
been enhanced to Rs 10,000 for first-time offenders.',
  [0.18323712012013349]),
 ('Natural language processing (NLP) is an area of computer
science and artificial intelligence concerned with the
interactions between computers and human (natural) languages,
in particular how to program computers to process and analyze
large amounts of natural language data.',
  [0.17995060855459855])]
```

If you see, doc4 (on top in result), this will be most relevant for the query "cricket" even though the word "cricket" is not even mentioned once with the similarity of 0.449.

Let's take one more example as may be driving.

```
Ranked_documents("driving")
```

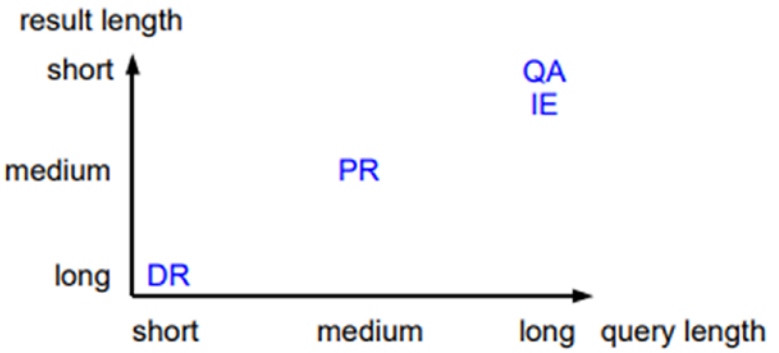
```
[('With the Union cabinet approving the amendments to the Motor
Vehicles Act, 2016, those caught for drunken driving will have
to have really deep pockets, as the fine payable in court has
been enhanced to Rs 10,000 for first-time offenders.',
  [0.35947287723800669]),
 ('But the man behind the wickets at the other end was watching
just as keenly. With an affirmative nod from Dhoni, India
captain Rohit Sharma promptly asked for a review. Sure enough,
the ball would have clipped the top of middle and leg.',
  [0.19042556935316801]),
 ('He points out that public transport is very good in Mumbai
and New Delhi, where there is a good network of suburban and
metro rail systems.',
  [0.17066536985237601]),
 ('Natural language processing (NLP) is an area of computer
science and artificial intelligence concerned with the
interactions between computers and human (natural) languages,
in particular how to program computers to process and analyze
large amounts of natural language data.',
  [0.088723080005327359])]
```

Again, since driving is connected to transport and the Motor Vehicles Act, it pulls out the most relevant documents on top. The first 2 documents are relevant to the query.

We can use the same approach and scale it up for as many documents as possible. For more accuracy, we can build our own embeddings, as we learned in Chapter 3, for specific industries since the one we are using is generalized.

This is the fundamental approach that can be used for many applications like the following:

- Search engines
- Document retrieval
- Passage retrieval
- Question and answer



It's been proven that results will be good when queries are longer and the result length is shorter. That's the reason we don't get great results in search engines when the search query has lesser number of words.

Recipe 6-2. Classifying Text with Deep Learning

In this recipe, let us build a text classifier using deep learning approaches.

Problem

We want to build a text classification model using CNN, RNN, and LSTM.

Solution

The approach and NLP pipeline would remain the same as discussed earlier. The only change would be that instead of using machine learning algorithms, we would be building models using deep learning algorithms.

How It Works

Let's follow the steps in this section to build the email classifier using the deep learning approaches.

Step 2-1 Understanding/defining business problem

Email classification (spam or ham). We need to classify spam or ham email based on email content.

Step 2-2 Identifying potential data sources, collection, and understanding

Using the same data used in Recipe 4-6 from Chapter 4:

```
#read file
file_content = pd.read_csv('spam.csv', encoding = "ISO-8859-1")

#check sample content in the email
file_content['v2'][1]

#output
'Ok lar... Joking wif u oni...'
```

Step 2-3 Text preprocessing

Let's preprocess the data:

```
#Import library
from nltk.corpus import stopwords
from nltk import *
from sklearn.feature_extraction.text import TfidfVectorizer
from nltk.stem import WordNetLemmatizer
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# Remove stop words
stop = stopwords.words('english')
file_content['v2'] = file_content['v2'].apply(lambda x: "
".join(x for x in x.split() if x not in stop))

# Delete unwanted columns
Email_Data = file_content[['v1', 'v2']]

# Rename column names
Email_Data = Email_Data.rename(columns={"v1":"Target", "v2":"Email"})
Email_Data.head()

 #output
    Target Email
0   ham    Go jurong point, crazy.. Available bugis n gre...
1   ham    Ok lar... Joking wif u oni...
2   spam   Free entry 2 wkly comp win FA Cup final tkts 2...
3   ham    U dun say early hor... U c already say...
4   ham    Nah I think goes usf, lives around though

#Delete punctuations, convert text in lower case and delete the
double space
```

```
Email_Data['Email'] = Email_Data['Email'].apply(lambda x:
re.sub('[!@#$:).;,?&]', '', x.lower()))
Email_Data['Email'] = Email_Data['Email'].apply(lambda x:
re.sub(' ', ' ', x))
Email_Data['Email'].head(5)

#output
0 go jurong point crazy available bugis n great ...
1 ok lar joking wif u oni
2 free entry 2 wkly comp win fa cup final tkts 2...
3 u dun say early hor u c already say
4 nah i think goes usf lives around though
Name: Email, dtype: object

#Separating text(input) and target classes

list_sentences_rawdata = Email_Data["Email"].fillna("_na_").values
list_classes = ["Target"]
target = Email_Data[list_classes].values

To_Process=Email_Data[['Email', 'Target']]
```

Step 2-4 Data preparation for model building

Now we prepare the data:

```
#Train and test split with 80:20 ratio
train, test = train_test_split(To_Process, test_size=0.2)

# Define the sequence lengths, max number of words and
embedding dimensions
# Sequence length of each sentence. If more, truncate. If less,
pad with zeros

MAX_SEQUENCE_LENGTH = 300
```

```
# Top 20000 frequently occurring words
MAX_NB_WORDS = 20000

# Get the frequently occurring words
 tokenizer = Tokenizer(num_words=MAX_NB_WORDS)
tokenizer.fit_on_texts(train.Email)
train_sequences = tokenizer.texts_to_sequences(train.Email)
test_sequences = tokenizer.texts_to_sequences(test.Email)

# dictionary containing words and their index
word_index = tokenizer.word_index
# print(tokenizer.word_index)
# total words in the corpus
print('Found %s unique tokens.' % len(word_index))

# get only the top frequent words on train
train_data = pad_sequences(train_sequences, maxlen=MAX_
SEQUENCE_LENGTH)

# get only the top frequent words on test
test_data = pad_sequences(test_sequences, maxlen=MAX_SEQUENCE_
LENGTH)

print(train_data.shape)
print(test_data.shape)

#output
Found 8443 unique tokens.
(4457, 300)
(1115, 300)

train_labels = train['Target']
test_labels = test['Target']

#import library
```

```
from sklearn.preprocessing import LabelEncoder
# converts the character array to numeric array. Assigns levels
to unique labels.

le = LabelEncoder()
le.fit(train_labels)
train_labels = le.transform(train_labels)
test_labels = le.transform(test_labels)

print(le.classes_)
print(np.unique(train_labels, return_counts=True))
print(np.unique(test_labels, return_counts=True))

#output
['ham' 'spam']
(array([0, 1]), array([3889, 568]))
(array([0, 1]), array([936, 179]))

# changing data types
labels_train = to_categorical(np.asarray(train_labels))
labels_test = to_categorical(np.asarray(test_labels))
print('Shape of data tensor:', train_data.shape)
print('Shape of label tensor:', labels_train.shape)
print('Shape of label tensor:', labels_test.shape)

#output
Shape of data tensor: (4457, 300)
Shape of label tensor: (4457, 2)
Shape of label tensor: (1115, 2)

EMBEDDING_DIM = 100
print(MAX_SEQUENCE_LENGTH)

#output
300
```

Step 2-5 Model building and predicting

We are building the models using different deep learning approaches like CNN, RNN, LSTM, and Bidirectional LSTM and comparing the performance of each model using different accuracy metrics.

We can now define our CNN model.

Here we define a single hidden layer with 128 memory units. The network uses a dropout with a probability of 0.5. The output layer is a dense layer using the softmax activation function to output a probability prediction.

```
# Import Libraries
import sys, os, re, csv, codecs, numpy as np, pandas as pd

from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
from keras.layers import Dense, Input, LSTM, Embedding,
Dropout, Activation
from keras.layers import Bidirectional, GlobalMaxPool1D,
Conv1D, SimpleRNN
from keras.models import Model
from keras.models import Sequential
from keras import initializers, regularizers, constraints,
optimizers, layers
from keras.layers import Dense, Input, Flatten, Dropout,
BatchNormalization
from keras.layers import Conv1D, MaxPooling1D, Embedding
from keras.models import Sequential

 print('Training CNN 1D model.')
```

```
model = Sequential()
model.add(Embedding(MAX_NB_WORDS,
 EMBEDDING_DIM,
 input_length=MAX_SEQUENCE_LENGTH
 ))
model.add(Dropout(0.5))
model.add(Conv1D(128, 5, activation='relu'))
model.add(MaxPooling1D(5))
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Conv1D(128, 5, activation='relu'))
model.add(MaxPooling1D(5))
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(2, activation='softmax'))

model.compile(loss='categorical_crossentropy',
 optimizer='rmsprop',
 metrics=['acc'])
```

We are now fitting our model to the data. Here we have 5 epochs and a batch size of 64 patterns.

```
model.fit(train_data, labels_train,
 batch_size=64,
 epochs=5,
 validation_data=(test_data, labels_test))
```

```
#output
```

```
Training CNN 1D model.
Train on 4457 samples, validate on 1115 samples
Epoch 1/5
4457/4457 [==============================] - 19s 4ms/step - loss: 0.3465 - acc: 0.8634 - val_loss: 0.3479 - val_acc:
0.9247
Epoch 2/5
4457/4457 [==============================] - 18s 4ms/step - loss: 0.1281 - acc: 0.9540 - val_loss: 0.1882 - val_acc:
0.9731
Epoch 3/5
4457/4457 [==============================] - 17s 4ms/step - loss: 0.0659 - acc: 0.9807 - val_loss: 0.5212 - val_acc:
0.9704
Epoch 4/5
4457/4457 [==============================] - 17s 4ms/step - loss: 0.0453 - acc: 0.9868 - val_loss: 0.5466 - val_acc:
0.9659
Epoch 5/5
4457/4457 [==============================] - 17s 4ms/step - loss: 0.0379 - acc: 0.9912 - val_loss: 0.5507 - val_acc:
0.9785

<keras.callbacks.History at 0x1a2df88f28>
```

```
#predictions on test data

predicted=model.predict(test_data)
predicted

#output
array([[0.5426713 , 0.45732868],
 [0.5431667 , 0.45683333],
 [0.53082496, 0.46917507],
 ...,
 [0.53582424, 0.46417573],
 [0.5305845 , 0.46941552],
 [0.53102577, 0.46897423]], dtype=float32)

#model evaluation

import sklearn
from sklearn.metrics import precision_recall_fscore_support as
score

precision, recall, fscore, support = score(labels_test,
predicted.round())

print('precision: {}'.format(precision))
print('recall: {}'.format(recall))
```

```
print('fscore: {}'.format(fscore))
print('support: {}'.format(support))

print("############################")

print(sklearn.metrics.classification_report(labels_test,
predicted.round()))

#output
```

```
precision: [0.98407643 0.94797688]
recall: [0.99038462 0.91620112]
fscore: [0.98722045 0.93181818]
support: [936 179]
############################
             precision    recall  f1-score   support

          0       0.98      0.99      0.99       936
          1       0.95      0.92      0.93       179

avg / total       0.98      0.98      0.98      1115
```

We can now define our RNN model.

```
#import library
from keras.layers.recurrent import SimpleRNN

#model training

print('Training SIMPLERNN model.')

model = Sequential()
model.add(Embedding(MAX_NB_WORDS,
 EMBEDDING_DIM,
 input_length=MAX_SEQUENCE_LENGTH
 ))
model.add(SimpleRNN(2, input_shape=(None,1)))

model.add(Dense(2,activation='softmax'))

model.compile(loss = 'binary_crossentropy',
optimizer='adam',metrics = ['accuracy'])
```

```
model.fit(train_data, labels_train,
 batch_size=16,
 epochs=5,
 validation_data=(test_data, labels_test))
```

```
#output
```

```
Training SIMPLERNN  model.
Train on 4457 samples, validate on 1115 samples
Epoch 1/5
4457/4457 [==============================] - 26s 6ms/step - loss: 0.2514 - acc: 0.9607 - val_loss: 0.1508 - val_acc:
0.9776
Epoch 2/5
4457/4457 [==============================] - 25s 6ms/step - loss: 0.0768 - acc: 0.9917 - val_loss: 0.1013 - val_acc:
0.9785
Epoch 3/5
4457/4457 [==============================] - 25s 6ms/step - loss: 0.0327 - acc: 0.9982 - val_loss: 0.0904 - val_acc:
0.9794
Epoch 4/5
4457/4457 [==============================] - 25s 6ms/step - loss: 0.0171 - acc: 0.9996 - val_loss: 0.0920 - val_acc:
0.9767
Epoch 5/5
4457/4457 [==============================] - 25s 6ms/step - loss: 0.0108 - acc: 1.0000 - val_loss: 0.0926 - val_acc:
0.9749
```

```
# prediction on test data
predicted_Srnn=model.predict(test_data)
predicted_Srnn
```

```
#output
array([[0.9959137 , 0.00408628],
 [0.99576926, 0.00423072],
 [0.99044365, 0.00955638],
 ...,
 [0.9920791 , 0.00792089],
 [0.9958105 , 0.00418955],
 [0.99660563, 0.00339443]], dtype=float32)
```

```
#model evaluation
```

```
from sklearn.metrics import precision_recall_fscore_support as score
```

```
precision, recall, fscore, support = score(labels_test,
predicted_Srnn.round())
```

```
print('precision: {}'.format(precision))
print('recall: {}'.format(recall))
print('fscore: {}'.format(fscore))
print('support: {}'.format(support))

print("############################")

print(sklearn.metrics.classification_report(labels_test,
predicted_Srnn.round()))

#output
```

```
precision: [0.97589099 0.9689441 ]
recall: [0.99465812 0.87150838]
fscore: [0.98518519 0.91764706]
support: [936 179]
############################
             precision    recall  f1-score   support

          0       0.98      0.99      0.99       936
          1       0.97      0.87      0.92       179

avg / total       0.97      0.97      0.97      1115
```

And here is our Long Short-Term Memory (LSTM):

```
#model training

print('Training LSTM model.')

model = Sequential()
model.add(Embedding(MAX_NB_WORDS,
 EMBEDDING_DIM,
 input_length=MAX_SEQUENCE_LENGTH
 ))
model.add(LSTM(output_dim=16, activation='relu', inner_
activation='hard_sigmoid',return_sequences=True))
model.add(Dropout(0.2))
model.add(BatchNormalization())
```

```
model.add(Flatten())

model.add(Dense(2,activation='softmax'))

model.compile(loss = 'binary_crossentropy',
optimizer='adam',metrics = ['accuracy'])

model.fit(train_data, labels_train,
 batch_size=16,
 epochs=5,
 validation_data=(test_data, labels_test))

#output
```

```
Training LSTM  model.
/Users/akulk7/anaconda/lib/python3.5/site-packages/ipykernel/__main__.py:12: UserWarning: Update your `LSTM` call to
the Keras 2 API: `LSTM(recurrent_activation="hard_sigmoid", return_sequences=True, units=16, activation="relu")`
Train on 4457 samples, validate on 1115 samples
Epoch 1/5
4457/4457 [==============================] - 75s 17ms/step - loss: 0.1260 - acc: 0.9587 - val_loss: 0.1605 - val_acc:
0.9596
Epoch 2/5
4457/4457 [==============================] - 72s 16ms/step - loss: 0.0147 - acc: 0.9964 - val_loss: 0.0810 - val_acc:
0.9794
Epoch 3/5
4457/4457 [==============================] - 72s 16ms/step - loss: 0.0028 - acc: 0.9991 - val_loss: 0.0968 - val_acc:
0.9812
Epoch 4/5
4457/4457 [==============================] - 73s 16ms/step - loss: 0.0018 - acc: 0.9998 - val_loss: 0.0892 - val_acc:
0.9830
Epoch 5/5
4457/4457 [==============================] - 78s 17ms/step - loss: 7.3629e-04 - acc: 0.9998 - val_loss: 0.1045 - val_
acc: 0.9830
```

```
#prediction on text data
predicted_lstm=model.predict(test_data)
predicted_lstm
array([[1.0000000e+00, 4.0581045e-09],
 [1.0000000e+00, 8.3188789e-13],
 [9.9999976e-01, 1.8647323e-07],
 ...,
 [9.9999976e-01, 1.8333606e-07],
 [1.0000000e+00, 1.7347950e-09],
 [9.9999988e-01, 1.3574694e-07]], dtype=float32)
```

```
#model evaluation

from sklearn.metrics import precision_recall_fscore_support as
score

precision, recall, fscore, support = score(labels_test,
predicted_lstm.round())

print('precision: {}'.format(precision))
print('recall: {}'.format(recall))
print('fscore: {}'.format(fscore))
print('support: {}'.format(support))

print("############################")

print(sklearn.metrics.classification_report(labels_test,
predicted_lstm.round()))

#output
```

```
precision: [0.98010471 1.        ]
recall: [1.         0.89385475]
fscore: [0.98995241 0.9439528 ]
support: [936 179]
############################
              precision    recall  f1-score   support

           0       0.98      1.00      0.99       936
           1       1.00      0.89      0.94       179

avg / total       0.98      0.98      0.98      1115
```

Finally, let's see what is Bidirectional LSTM and implement the same.

As we know, LSTM preserves information from inputs using the hidden state. In bidirectional LSTMs, inputs are fed in two ways: one from previous to future and the other going backward from future to past, helping in learning future representation as well. Bidirectional LSTMs are known for producing very good results as they are capable of understanding the context better.

```
#model training

print('Training Bidirectional LSTM model.')

model = Sequential()
model.add(Embedding(MAX_NB_WORDS,
 EMBEDDING_DIM,
 input_length=MAX_SEQUENCE_LENGTH
 ))
model.add(Bidirectional(LSTM(16, return_sequences=True,
dropout=0.1, recurrent_dropout=0.1)))
model.add(Conv1D(16, kernel_size = 3, padding = "valid",
kernel_initializer = "glorot_uniform"))
model.add(GlobalMaxPool1D())
model.add(Dense(50, activation="relu"))
model.add(Dropout(0.1))

model.add(Dense(2,activation='softmax'))

model.compile(loss = 'binary_crossentropy',
optimizer='adam',metrics = ['accuracy'])

model.fit(train_data, labels_train,
 batch_size=16,
 epochs=3,
 validation_data=(test_data, labels_test))

#output
```

```
Training Bidirectional LSTM  model.
Train on 4457 samples, validate on 1115 samples
Epoch 1/3
4457/4457 [==============================] - 104s 23ms/step - loss: 0.1401 - acc: 0.9502 - val_loss: 0.0669 - val_ac
c: 0.9821
Epoch 2/3
4457/4457 [==============================] - 99s 22ms/step - loss: 0.0119 - acc: 0.9960 - val_loss: 0.0776 - val_acc:
0.9812
Epoch 3/3
4457/4457 [==============================] - 100s 22ms/step - loss: 0.0020 - acc: 0.9998 - val_loss: 0.0890 - val_ac
c: 0.9857
```

```
# prediction on test data

predicted_blstm=model.predict(test_data)
predicted_blstm

#output
array([[9.9999976e-01, 2.6086647e-07],
 [9.9999809e-01, 1.9633851e-06],
 [9.9999833e-01, 1.6918856e-06],
 ...,
 [9.9999273e-01, 7.2622524e-06],
 [9.9999964e-01, 3.3541210e-07],
 [9.9999964e-01, 3.5427794e-07]], dtype=float32)

#model evaluation

from sklearn.metrics import precision_recall_fscore_support as
score

precision, recall, fscore, support = score(labels_test,
predicted_blstm.round())

print('precision: {}'.format(precision))
print('recall: {}'.format(recall))
print('fscore: {}'.format(fscore))
print('support: {}'.format(support))

print("############################")

print(sklearn.metrics.classification_report(labels_test,
predicted_blstm.round()))
```

```
#output
```

```
precision: [0.98421053 0.99393939]
recall: [0.99893162 0.91620112]
fscore: [0.99151644 0.95348837]
support: [936 179]
###########################
              precision    recall  f1-score   support

           0       0.98      1.00      0.99       936
           1       0.99      0.92      0.95       179

avg / total       0.99      0.99      0.99      1115
```

We can see that Bidirectional LSTM outperforms the rest of the algorithms.

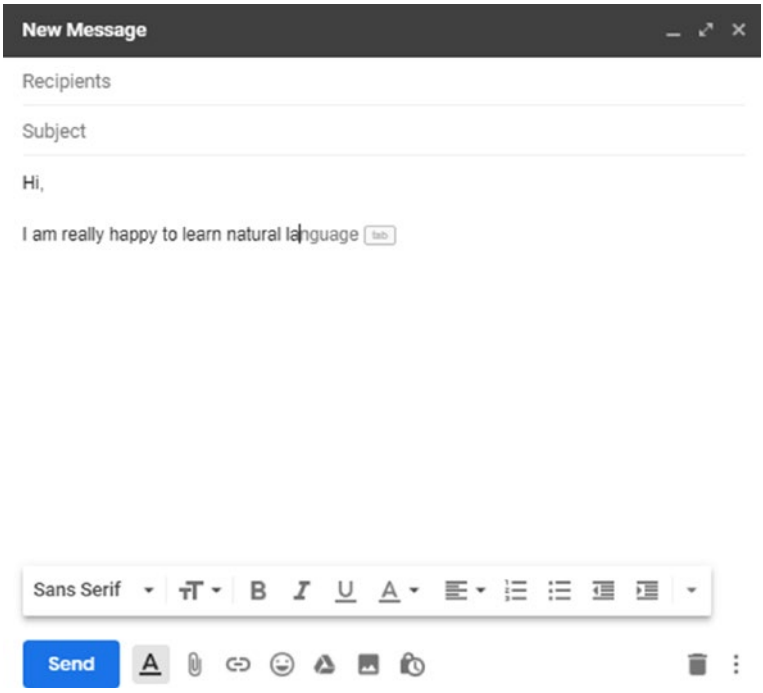
Recipe 6-3. Next Word Prediction

Autofill/showing what could be the potential sequence of words saves a lot of time while writing emails and makes users happy to use it in any product.

Problem

You want to build a model to predict/suggest the next word based on a previous sequence of words using Email Data.

Like you see in the below image, language is being suggested as the next word.



Solution

In this section, we will build an LSTM model to learn sequences of words from email data. We will use this model to predict the next word.

How It Works

Let's follow the steps in this section to build the next word prediction model using the deep learning approach.

Step 3-1 Understanding/defining business problem

Predict the next word based on the sequence of words or sentences.

Step 3-2 Identifying potential data sources, collection, and understanding

For this problem, let us use the same email data used in Recipe 4-6 from Chapter 4. This has a lot less data, but still to showcase the working flow, we are fine with this data. The more data, the better the accuracy.

```
file_content = pd.read_csv('spam.csv', encoding = "ISO-8859-1")

# Just selecting emails and connverting it into list
Email_Data = file_content[[ 'v2']]

list_data = Email_Data.values.tolist()
list_data

#output
[['Go until jurong point, crazy.. Available only in bugis n
  great world la e buffet... Cine there got amore wat...'],
 ['Ok lar... Joking wif u oni...'],
 ["Free entry in 2 a wkly comp to win FA Cup final tkts 21st
  May 2005. Text FA to 87121 to receive entry question(std txt
  rate)T&C's apply 08452810075over18's"],
 ['U dun say so early hor... U c already then say...'],
 ["Nah I don't think he goes to usf, he lives around here though"],
 ["FreeMsg Hey there darling it's been 3 week's now and no word
   back! I'd like some fun you up for it still? Tb ok! XxX std
   chgs to send, å£1.50 to rcv"],
 ['Even my brother is not like to speak with me. They treat me
   like aids patent.'],
 ["As per your request 'Melle Melle (Oru Minnaminunginte
   Nurungu Vettam)' has been set as your callertune for all
   Callers. Press *9 to copy your friends Callertune"],
```

```
['WINNER!! As a valued network customer you have been selected
 to receivea å£900 prize reward! To claim call 09061701461.
 Claim code KL341. Valid 12 hours only.'] ,
['Had your mobile 11 months or more? U R entitled to Update
 to the latest colour mobiles with camera for Free! Call The
 Mobile Update Co FREE on 08002986030'],
```

Step 3-3 Importing and installing necessary libraries

Here are the libraries:

```
import numpy as np
import random
import pandas as pd
import sys
import os
import time
import codecs
import collections
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils
from nltk.tokenize import sent_tokenize, word_tokenize
import scipy
from scipy import spatial
from nltk.tokenize.toktok import ToktokTokenizer
import re
tokenizer = ToktokTokenizer()
```

Step 3-4 Processing the data

Now we process the data:

```
#Converting list to string
from collections import Iterable

def flatten(items):
    """Yield items from any nested iterable"""
    for x in items:
        if isinstance(x, Iterable) and not isinstance(x,
        (str, bytes)):
            for sub_x in flatten(x):
                yield sub_x
        else:
            yield x

TextData=list(flatten(list_data))
TextData = ''.join(TextData)

# Remove unwanted lines and converting into lower case
TextData = TextData.replace('\n','')
TextData = TextData.lower()

pattern = r'[^a-zA-z0-9\s]'
TextData = re.sub(pattern, '', ''.join(TextData))

# Tokenizing

tokens = tokenizer.tokenize(TextData)
tokens = [token.strip() for token in tokens]

# get the distinct words and sort it

word_counts = collections.Counter(tokens)
word_c = len(word_counts)
```

```
print(word_c)

distinct_words = [x[0] for x in word_counts.most_common()]
distinct_words_sorted = list(sorted(distinct_words))

# Generate indexing for all words

word_index = {x: i for i, x in enumerate(distinct_words_sorted)}

# decide on sentence length

sentence_length = 25
```

Step 3-5 Data preparation for modeling

Here we are dividing the mails into sequence of words with a fixed length of 10 words (you can choose anything based on the business problem and computation power). We are splitting the text by words sequences. When creating these sequences, we slide this window along the whole document one word at a time, allowing each word to learn from its preceding one.

```
#prepare the dataset of input to output pairs encoded as integers
# Generate the data for the model

#input = the input sentence to the model with index
#output = output of the model with index

InputData = []
OutputData = []

for i in range(0, word_c - sentence_length, 1):
    X = tokens[i:i + sentence_length]
    Y = tokens[i + sentence_length]
    InputData.append([word_index[char] for char in X])
    OutputData.append(word_index[Y])
```

```
print (InputData[:1])
print ("\n")
print(OutputData[:1])

#output

[[5086, 12190, 6352, 9096, 3352, 1920, 8507, 5937, 2535, 7886,
5214, 12910, 6541, 4104, 2531, 2997, 11473, 5170, 1595, 12552,
6590, 6316, 12758, 12087, 8496]]

[4292]

# Generate  X
X = numpy.reshape(InputData, (len(InputData), sentence_length, 1))

# One hot encode the output variable
Y = np_utils.to_categorical(OutputData)

Y
#output
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

Step 3-6 Model building

We will now define the LSTM model. Here we define a single hidden LSTM layer with 256 memory units. This model uses dropout 0.2. The output layer is using the softmax activation function. Here we are using the ADAM optimizer.

```
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2])))
model.add(Dropout(0.2))
model.add(Dense(Y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy',
optimizer='adam')

#define the checkpoint
file_name_path="weights-improvement-{epoch:02d}-{loss:.4f}.hdf5"
checkpoint = ModelCheckpoint(file_name_path, monitor='loss',
verbose=1, save_best_only=True, mode='min')
callbacks = [checkpoint]
```

We can now fit the model to the data. Here we use 5 epochs and a batch size of 128 patterns. For better results, you can use more epochs like 50 or 100. And of course, you can use them on more data.

```
#fit the model
model.fit(X, Y, epochs=5, batch_size=128, callbacks=callbacks)
```

Note We have not split the data into training and testing data. We are not interested in the accurate model. As we all know, deep learning models will require a lot of data for training and take a lot of time to train, so we are using a model checkpoint to capture all of the model weights to file. We will use the best set of weights for our prediction.

```
#output
```

```
Epoch 1/5
13312/13335 [===========================>.] - ETA: 0s - loss: 7.9041
Epoch 00001: loss improved from inf to 7.90363, saving model to weights-improvement-01-7.9036.hdf5
13335/13335 [==============================] - 30s 2ms/step - loss: 7.9036
Epoch 2/5
13312/13335 [===========================>.] - ETA: 0s - loss: 7.1114
Epoch 00002: loss improved from 7.90363 to 7.11067, saving model to weights-improvement-02-7.1107.hdf5
13335/13335 [==============================] - 28s 2ms/step - loss: 7.1107
Epoch 3/5
13312/13335 [===========================>.] - ETA: 0s - loss: 7.0211
Epoch 00003: loss improved from 7.11067 to 7.02179, saving model to weights-improvement-03-7.0218.hdf5
13335/13335 [==============================] - 26s 2ms/step - loss: 7.0218
Epoch 4/5
13312/13335 [===========================>.] - ETA: 0s - loss: 6.9316
Epoch 00004: loss improved from 7.02179 to 6.93116, saving model to weights-improvement-04-6.9312.hdf5
13335/13335 [==============================] - 26s 2ms/step - loss: 6.9312
Epoch 5/5
13312/13335 [===========================>.] - ETA: 0s - loss: 6.8516
Epoch 00005: loss improved from 6.93116 to 6.85182, saving model to weights-improvement-05-6.8518.hdf5
13335/13335 [==============================] - 28s 2ms/step - loss: 6.8518
```

After running the above code, you will have weight checkpoint files in your local directory. Pick the network weights file that is saved in your working directory. For example, when we ran this example, below was the checkpoint with the smallest loss that we achieved with 5 epochs.

```
# load the network weights
file_name = "weights-improvement-05-6.8213.hdf5"
model.load_weights(file_name)
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Step 3-7 Predicting next word

We will randomly generate a sequence of words and input to the model and see what it predicts.

```
# Generating random sequence
start = numpy.random.randint(0, len(InputData))
input_sent = InputData[start]

# Generate index of the next word of the email

X = numpy.reshape(input_sent, (1, len(input_sent), 1))
```

```
predict_word = model.predict(X, verbose=0)
index = numpy.argmax(predict_word)

print(input_sent)
print ("\n")
print(index)

# Output
[9122, 1920, 8187, 5905, 6828, 9818, 1791, 5567, 1597, 7092,
11606, 7466, 10198, 6105, 1837, 4752, 7092, 3928, 10347, 5849,
8816, 7092, 8574, 7092, 1831]

5849

# Convert these indexes back to words

word_index_rev = dict((i, c) for i, c in enumerate(tokens))
result = word_index_rev[index]
sent_in = [word_index_rev[value] for value in input_sent]

print(sent_in)
print ("\n")
print(result)

Result :

['us', 'came', 'use', 'respecthe', 'would', 'us', 'are', 'it',
'you', 'to', 'pray', 'because', 'you', 'do', 'me', 'out', 'youre',
'thk', 'where', 'are', 'mrng', 'minutes', 'long', '500', 'per']

shut
```

So, given the 25 input words, it's predicting the word “shut” as the next word. Of course, its not making much sense, since it has been trained on much less data and epochs. Make sure you have great computation power and train on huge data with high number of epochs.

Index

A. Kulkarni and A. Shivananda, *Natural Language Processing Recipes*,
https://doi.org/10.1007/978-1-4842-4267-4

J, K

L

M

N

O

P, Q

R

S

T

U

V

W, X, Y, Z