

1ST EDITION

# The Handbook of NLP with Gensim

Leverage topic modeling to uncover hidden patterns,  
themes, and valuable insights within textual data



**CHRIS KUO**

# The Handbook of NLP with Gensim

Leverage topic modeling to uncover hidden patterns, themes, and valuable insights within textual data

**Chris Kuo**



BIRMINGHAM—MUMBAI

# The Handbook of NLP with Gensim

Copyright © 2023 Packt Publishing

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Group Product Manager:** Niranjan Naikwadi

**Publishing Product Manager:** Anant Jain

**Book Project Manager:** Hemangi Lotlikar

**Senior Editor:** Rohit Singh

**Technical Editor:** Rahul Limbachiya

**Copy Editor:** Safis Editing

**Proofreader:** Safis Editing

**Indexer:** Rekha Nair

**Production Designer:** Vijay Kamble

**DevRel Marketing Executive:** Vinishka Kalra

First published: October 2023

Production reference: 2131023

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80324-494-5

[www.packtpub.com](http://www.packtpub.com)

*To God be the glory*

*- Chris Kuo*

# Contributors

## About the author

**Chris Kuo** is a data scientist and an adjunct professor with over 23 years of experience. He led various data science solutions including customer analytics, health analytics, fraud detection, and litigation. He is also an inventor of a U.S. patent. He has worked at several Fortune 500 companies in the insurance and retail industries.

Chris teaches at Columbia University and has taught at Boston University and other universities. He has published articles in economic and management journals and served as a journal reviewer. He is the author of *The eXplainable A.I.*, *Modern Time Series Anomaly Detection*, *Transfer Learning for Image Classification*, and *The Handbook of Anomaly Detection*. He received his undergraduate degree in Nuclear Engineering and Ph.D. in Economics.

## About the reviewers

**Amreth Chandrasehar** is a director at Informatica, where he is responsible for ML engineering, observability, and SRE teams. Over the last few years, he has played a key role in cloud migration, generative AI, observability, and ML adoption at various organizations. He is also a co-creator of the Conducktor Platform, serving T-Mobile's 100 million+ customers, and a Tech/Customer Advisory board member at various companies on observability.

Amreth has also co-created the open source Kardio.io, a service health dashboard tool. He has been invited to speak at several key conferences and has won several awards within the company and was recently awarded 3 Gold Awards at Globiee, Stevie, and International Achievers Awards for his contributions in observability and generative AI.

*I would like to thank my wife, Ashwinya Mani, and my son, Athvik A, for their patience and support provided during my review of this book.*

**Devashish Deshpande** is a Machine Learning Engineer with expertise in Natural Language Processing. He possesses an undergraduate degree in Computer Science as well as an advanced master's degree in Artificial Intelligence. Contributing early on to popular open source libraries, such as Gensim and Scikit-Learn, helped him gain experience in how research papers can be translated to code and deployed and used by large communities. Later, his corporate experience helped in developing strong software engineering fundamentals. Currently, Devashish works on researching and developing better machine learning models and deploying them in a performant and practical way.



# Table of Contents

## Preface

xvii

---

## Part 1: NLP Basics

1

Introduction to NLP	3
Introduction to natural language processing	12
4	LDA
4	Ensemble LDA
NLU + NLG = NLP	13
5	Topic modeling with BERTopic
5	Common NLP Python modules included in this book
6	14
Gensim and its NLP modeling techniques	14
8	spaCy
8	NLTK
BoW and TF-IDF	15
9	Summary
LSA/LSI	15
10	Questions
Word2Vec	15
10	References
Doc2Vec	16
12	

2

Text Representation	17
Technical requirements	20
17	Bag-of-N-grams
What word embedding is	20
18	What TF-IDF is
Simple encoding methods	21
18	Shining applications of BoW
One-hot encoding	21
18	and TF-IDF
BoW	22
19	Coding – BoW

Gensim for BoW	22	<b>Coding – TF-IDF</b>	31
scikit-learn for BoW (CountVectorizer)	24	Gensim for TF-IDF	31
<b>Coding – Bag-of-N-grams</b>	<b>26</b>	scikit-learn for TF-IDF	33
Gensim for N-grams	26	<b>Summary</b>	34
scikit-learn for N-grams	28	<b>Questions</b>	34
NLTK for N-grams	29	<b>References</b>	34

## 3

<b>Text Wrangling and Preprocessing</b>	<b>35</b>		
Technical requirements	35	NLTK for tokenization	42
Key steps in NLP preprocessing	36	NLTK for stop-word removal	43
Tokenization	36	NLTK for lemmatization	43
Lowercase conversion	36	<b>Coding with Gensim</b>	44
Stop word removal	36	Gensim for preprocessing	44
Punctuation removal	37	Gensim for stop-word removal	45
Stemming	37	Gensim for stemming	45
Lemmatization	37	<b>Building a pipeline with spaCy</b>	45
<b>Coding with spaCy</b>	<b>38</b>	Summary	49
spaCy for lemmatization	40	<b>Questions</b>	49
spaCy for PoS	41	<b>References</b>	49
Coding with NLTK	42		

## Part 2: Latent Semantic Analysis/Latent Semantic Indexing

## 4

<b>Latent Semantic Analysis with scikit-learn</b>	<b>53</b>		
Technical requirements	54	<b>Understanding a transformation matrix</b>	55
<b>Understanding matrix operations</b>	<b>54</b>	A transformation matrix in daily life examples	56
An orthogonal matrix	54		
The determinant of a matrix	55		

---

<b>Understanding eigenvectors and eigenvalues</b>	<b>58</b>	<b>Using TruncatedSVD for LSI with real data</b>	<b>65</b>
<b>An introduction to SVD</b>	<b>59</b>	Loading the data	65
Truncated SVD	61	Creating TF-IDF	66
Truncated SVD for LSI	61	Using TruncatedSVD to build a model	67
		Interpreting the outcome	67
<b>Coding truncatedSVD with scikit-learn</b>	<b>62</b>	<b>Summary</b>	<b>70</b>
Using TruncatedSVD	62	<b>Questions</b>	<b>70</b>
randomized_SVD	63		

## 5

<b>Cosine Similarity</b>		<b>71</b>	
Technical requirements	72	<b>Summary</b>	<b>76</b>
What is cosine similarity?	72	<b>Questions</b>	<b>76</b>
How cosine similarity is used in images	73	<b>References</b>	<b>76</b>
How to compute cosine similarity with scikit-learn	74		

## 6

<b>Latent Semantic Indexing with Gensim</b>		<b>77</b>	
Technical requirements	77	<b>Using the model as an information retrieval tool</b>	<b>92</b>
Performing text preprocessing	78	Loading the dictionary list	93
Performing word embedding with BoW and TF-IDF	79	Preprocessing the new document	94
BoW	80	Scoring the document to get the latent topic scores	95
TF-IDF	81	Calculating the similarity scores with the new document	96
<b>Modeling with Gensim</b>	<b>81</b>	Finding documents with high similarity scores	96
BoW	82	<b>Summary</b>	<b>98</b>
TF-IDF	83	<b>Questions</b>	<b>98</b>
Using the coherence score to find the optimal number of topics	85	<b>References</b>	<b>98</b>
Saving the model for production	88		

## Part 3: Word2Vec and Doc2Vec

7

### Using Word2Vec 101

Technical requirements	102	Visualizing Word2Vec with TensorBoard	115
Introduction to Word2Vec	102	Training your own Word2Vec model in CBOW and Skip-Gram	117
Advantages of Word2Vec	103	Load the data	117
Reviewing the real-world applications of Word2Vec	104	Text preprocessing	117
Introduction to Skip-Gram (SG)	105	Training your own Word2Vec model in CBOW	118
Data preparation	105	Training your own Word2Vec model in Skip-Gram	119
The input and output layers	107	Visualizing your Word2Vec model with t-SNE	120
The hidden layer	108	Comparing Word2Vec with Doc2Vec, GloVe, and fastText	123
Should I remove stop words for training Word2Vec?	108	Word2Vec versus Doc2Vec	123
Model computation	108	Word2Vec versus GloVe	124
Introduction to CBOW	109	Word2Vec versus FastText	124
Using a pretrained model for semantic search	110	Summary	124
Adding and subtracting words/concepts	113	Questions	125
Example 1	114	References	125
Example 2	114		

8

### Doc2Vec with Gensim 127

Technical requirements	128	Model optimization	132
From Word2Vec to Doc2Vec	129	PV-DM	132
PV-DBOW	129	The real-world applications of Doc2Vec	134
The input layer	131	Doc2Vec modeling with Gensim	134
The hidden layer	131		
The output layer	131		

---

Text preprocessing for Doc2Vec	135	Use case 2 – find relevant documents based on keywords	140
Modeling	136		
Saving the model	137	<b>Tips on building a good Doc2Vec model</b>	142
Saving the training data	137		
<b>Putting the model into production</b>	<b>138</b>	<b>Summary</b>	<b>142</b>
Loading the model	138	<b>Questions</b>	142
Loading the training data	138	<b>References</b>	143
Use case 1 – find similar articles	138		

## Part 4: Topic Modeling with Latent Dirichlet Allocation

### 9

---

<b>Understanding Discrete Distributions</b>	<b>147</b>		
Technical requirements	148	<b>Beta distributions</b>	158
The basics of discrete probability distributions	148	The real-world examples	159
Bernoulli distributions	149	The formal definition of a beta distribution	159
The formal definition of a Bernoulli distribution	149	What does it look like?	160
What does it look like?	149	The beta distribution in Bayesian inference	165
Fun facts	150	Fun fact	165
<b>Binomial distributions</b>	<b>150</b>	<b>Dirichlet distributions</b>	<b>166</b>
The real-world examples	150	Real-world examples	166
The formal definition of a binomial distribution	151	The formal definition of a Dirichlet distribution	166
What does it look like?	151	What is a simplex?	167
Plotting it with Python	153	What does the Dirichlet distribution look like?	168
Fun facts	155	The Dirichlet distribution in Bayesian inference	173
<b>Multinomial distributions</b>	<b>156</b>	Fun fact	174
The real-world examples	156	<b>Summary</b>	<b>174</b>
The formal definition of a multinomial distribution	156	<b>Questions</b>	<b>174</b>
What does it look like?	156	<b>References</b>	<b>174</b>
Fun facts	158		

## 10

<b>Latent Dirichlet Allocation</b>	<b>175</b>
What is generative modeling?	176
Discriminative modeling	176
Generative modeling	176
Bayes' theorem	176
Expectation-Maximization (EM)	178
Understanding the idea behind LDA	179
Dirichlet distribution of topics	180
Understanding the structure of LDA	180
Variational inference	183
Variational E-M	185
Gibbs sampling in LDA	185
Variational E-M versus Gibbs sampling	186
Summary	186
Questions	187
References	187

## 11

<b>LDA Modeling</b>	<b>189</b>
Technical requirements	189
Text preprocessing	190
Preprocessing	190
Experimenting with LDA modeling	192
A model built on BoW data	192
A model built on TF-IDF data	194
Building LDA models with a different number of topics	196
Models built on BoW data	196
Models built on TF-IDF data	197
Determining the optimal number of topics	197
Using the model to score new documents	200
Text preprocessing	200
Scoring new texts	201
Outcome	201
Summary	203
Questions	203
References	203

## 12

<b>LDA Visualization</b>	<b>205</b>
Technical requirements	205
Designing an infographic	205
Data visualization with pyLDAvis	206
The interactive graph	207

---

<b>Summary</b>	<b>212</b>	<b>References</b>	<b>212</b>
<b>Questions</b>	<b>212</b>		

## 13

---

<b>The Ensemble LDA for Model Stability</b>	<b>213</b>
---	------------

<b>Technical requirements</b>	<b>214</b>	Preprocessing the training data	<b>218</b>
<b>From LDA to Ensemble LDA</b>	<b>214</b>	Creating text representation with BOW and TF-IDF	<b>219</b>
<b>The process of Ensemble LDA</b>	<b>215</b>	Saving the dictionary	<b>219</b>
<b>Understanding DBSCAN and CBDBSCAN</b>	<b>215</b>	Building the Ensemble LDA model	<b>220</b>
DBSCAN	215	Scoring new documents	222
CBDBSCAN (Checkback DBSCAN)	217	<b>Summary</b>	<b>223</b>
<b>Building an Ensemble LDA model with Gensim</b>	<b>218</b>	<b>Questions</b>	<b>223</b>
		<b>References</b>	<b>223</b>

## Part 5: Comparison and Applications

## 14

---

<b>LDA and BERTopic</b>	<b>227</b>
-------------------------	------------

<b>Technical requirements</b>	<b>228</b>	Modeling	<b>234</b>
<b>Understanding the Transformer model</b>	<b>228</b>	<b>Reviewing the results of BERTopic</b>	<b>235</b>
<b>Understanding BERT</b>	<b>230</b>	Getting the topic information	<b>236</b>
<b>Describing how BERTopic works</b>	<b>231</b>	Inspecting the keywords of a single topic	<b>237</b>
BERT – word embeddings	231	Getting document information	237
UMAP – reduce the dimensionality of embeddings	232	Getting representative documents	239
HDBSCAN – cluster documents	232	<b>Visualizing the BERTTopic model</b>	<b>239</b>
c-TFIDF – create a topic representation	233	Visualizing topics	240
Maximal Marginal Relevance	233	Visualizing the hierarchy of topics	241
<b>Building a BERTTopic model</b>	<b>233</b>	Visualizing the top words of topics	241
Loading the data – no text preprocessing	234	Visualizing on a heatmap	242
		<b>Predicting new documents</b>	<b>242</b>

<b>Using the modular property of BERTopic</b>	<b>243</b>	Language understanding	247
Word embeddings	244	Topic clarity	247
Dimensionality reduction	244	Determination of the number of topics	247
Clustering	245	Determination of word significance in a topic	247
<b>Comparing BERTopic with LDA</b>	<b>246</b>	<b>Summary</b>	<b>247</b>
Approach	246	<b>Questions</b>	<b>248</b>
Word embeddings	246	<b>References</b>	<b>248</b>
Text preprocessing	246		

## 15

<b>Real-World Use Cases</b>		<b>249</b>	
<b>Word2Vec for medical fraud detection</b>	<b>250</b>	<b>Interpretable text classification from electronic health records</b>	<b>255</b>
Background	250	Background	255
Questions	251	Questions	256
NLP solution	251	NLP solution	256
Takeaways	251	Takeaways	256
Background	252	<b>BERTopic for legal documents</b>	<b>256</b>
Questions	252	Background	256
NLP solution	252	Questions	257
Takeaways	253	NLP solution	257
Background	253	Takeaways	257
Questions	253	<b>Word2Vec for 10-K financial documents to the SEC</b>	<b>257</b>
NLP solution	253	Background	257
Takeaways	254	Questions	258
<b>Comparing LDA/NMF/BERTopic on Twitter/X posts</b>	<b>254</b>	NLP solution	258
Background	254	Takeaways	258
Questions	254	<b>Summary</b>	<b>258</b>
NLP solution	254	<b>References</b>	<b>259</b>
Takeaways	254		

<b>Assessments</b>	<b>261</b>
Chapter 1 – Introduction to NLP	261
Chapter 2 – Text Representation	261
Chapter 3 – Text Wrangling and Preprocessing	262
Chapter 4 – Latent Semantic Analysis with scikit-learn	262
Chapter 5 – Cosine Similarity	263
Chapter 6 – Latent Semantic Indexing with Gensim	263
Chapter 7 – Using Word2Vec	264
Chapter 8 – Doc2Vec with Gensim	265
Chapter 9 – Understanding Discrete Distributions	265
Chapter 10 – Latent Dirichlet Allocation	266
Chapter 11 – LDA Modeling	267
Chapter 12 – LDA Visualization	267
Chapter 13 – The Ensemble LDA for Model Stability	268
Chapter 14 – LDA and BERTopic	268
<b>Index</b>	<b>271</b>
<b>Other Books You May Enjoy</b>	<b>282</b>



# Preface

With the arrival of ChatGPT in late 2022 and GPT-4 in early 2023, there is an ignited interest in **natural language processing (NLP)** including **large language models (LLMs)**. You will find this book very helpful if you are picking it up hoping to get a start with NLP, learn and build the NLP techniques that have matured in the past few decades, or understand the differences between pre-LLM and LLM techniques. With the NLP development in the past four decades, there have been many commercial NLP products built on pre-LLM techniques, such as Word2Vec, Doc2Vec, **Latent Semantic Analysis (LSA)** or called **Latent Semantic Indexing (LSI)**, **Latent Dirichlet Allocation (LDA)**, and Ensemble LDA.

With the help of this book, you will not only get started with NLP to build NLP models but also get equipped with some background knowledge of LLMs. We believe the concepts covered in this book will be the necessary bridge for anyone new who comes to NLP, who wants to build NLP products, and who wants to learn LLMs.

## Why read this book?

To assist you in learning fundamental NLP concepts and building your NLP applications, we will start with NLP concepts and techniques that enable commercial NLP applications. This guide covers both theories and code practices. It presents NLP topics, so beginners as well as experienced data scientists can benefit from it.

Many of the techniques mentioned earlier, such as Word2Vec, Doc2Vec, LSA, LDA, and Ensemble LDA, are included in the Python Gensim module. Gensim is an open source Python library widely used by NLP researchers and developers, together with other NLP open source modules, including NLTK, Scikit-learn, and spaCy. We will learn how to build models using these modules. In addition, you will also learn about the Transformer-based topic modeling BERTopic in a separate chapter, and a BERTopic use case in the last chapter for NLP use cases.

You will also get to practice implementing your model for scoring and predictions. This implementation perspective enables you to work with data engineers closely in model deployment. We'll conclude the book with a study of selected large-scale NLP use cases. We believe these use cases can inspire you to build your NLP applications.

## What is Gensim

New NLP learners may find the Gensim library cited in many tutorials. Gensim is an open source Python library to process unstructured texts using unsupervised machine learning algorithms. It was first created by Radim Řehůřek in 2011 and is now developed and maintained continually by 400+ contributors. It has been used in over 2000 research papers and student theses.

One of Gensim's merits is its fast execution speed. Gensim attributes this advantage to its use of low-level BLAS libraries through NumPy, highly optimized Fortran/C, and multithreading under the hood. Memory independence is also one of their design objectives. Gensim enables data streaming to process large corpora without the need to load a whole training corpus in RAM.

## Who this book is for

This book does not assume any prior linguistic knowledge or NLP techniques, so it is suitable for anyone who wants to learn NLP. Data scientists and professionals who want to develop NLP applications will also find it helpful. If you are an NLP practitioner, you can consider this book as a code reference when working on your projects. Those practicing for an upper-class level NLP course can also use this book.

## What this book covers

*Chapter 1, Introduction to NLP*, is an introductory chapter that explains the development from **Natural Language Understanding (NLU)** and **Natural Language Generation (NLG)** to NLP. It briefs the core techniques including text pre-processing, LSA/LSI, Word2Vec, Doc2Vec, LDA, Ensemble LDA, and BERTopic. It presents the open source NLP modules Gensim, Scikit-learn, and Spacy.

*Chapter 2, Text Representation*, starts with the basic step of text representation. It explains the motivation from one-hot encoding to **Bag-of-words (BoW)** and **Term Frequency-Inverse Document Frequency (TF-IDF)**. It demonstrates how to perform BoW and TF-IDF with Gensim, Scikit-learn, and NLTK.

*Chapter 3, Text Wrangling and Preprocessing*, presents the essential text pre-processing tasks: (a) tokenization, (b) lowercase conversion, (c) stop words removal, (d) punctuation removal, (e) stemming, and (f) lemmatization. It guides you to perform the pre-processing tasks with Gensim, spaCy, and NLTK.

*Chapter 4, Latent Semantic Analysis with scikit-learn*, presents the theory of LSA/LSI. This chapter introduces **Singular Vector Decomposition (SVD)**, Truncated SVD, and Truncated SVD's application to LSA/LSI. This chapter uses Scikit-learn to illustrate the transition of Truncated SVD to LSA/LSI explicitly.

*Chapter 5, Cosine Similarity*, is dedicated to explaining this fundamental measure in NLP. Cosine similarity, among other metrics such as Euclidean distance or Manhattan distance, measures the similarity between embedded data in the vector space. This chapter also indicates the applications of cosine similarity for image comparison and querying.

*Chapter 6, Latent Semantic Indexing with Gensim*, builds an LSA/LSI model with Gensim. This chapter introduces the concept of coherence score that determines the optimal number of topics. It shows how to score new documents with the use of cosine similarity to add to an information retrieval tool.

*Chapter 7, Using Word2Vec*, introduces the milestone Word2Vec technique and its two neural network architectural variations: **Continuous Bag-of-Word (CBOW)** and **Skip Gram (SG)**. It illustrates the concept and operation for word embedding in the vector space. It guides you to build a word2Vec model and prepares it as part of an informational retrieval tool. It visualizes word vectors of a Word2Vec model with t-SNE and TensorBoard (by TensorFlow). This chapter ends with the comparisons of Word2Vec with Doc2Vec, GloVe, and FastText.

*Chapter 8, Doc2Vec with Gensim*, presents the evolution from Word2Vec to Doc2Vec. It details the two neural network architectural variations: **Paragraph Vector with Distributed Bag-of-words (PV-DBOW)** and **Paragraph Vectors with Distributed Memory (PV-DM)**. It guides you to build a Doc2Vec model and prepares it as part of an informational retrieval tool

*Chapter 9, Understanding Discrete Distributions*, introduces the discrete distribution family including Bernoulli, binomial, multinomial, beta, and Dirichlet distribution. Because the complex distributions are the generalization of the simple distributions, this sequence helps you to understand Dirichlet distribution. The fact that ‘Dirichlet’ is in the title of LDA tells us its significance. This chapter helps you understand LDA in the next chapter.

*Chapter 10, Latent Dirichlet Allocation*, presents the LDA algorithm, including the structural design of LDA, generative modeling, and Variational Expectation-Maximization.

*Chapter 11, LDA Modeling*, demonstrates how to build an LDA model, perform hyperparameter tuning, and determine the optimal number of topics. You will learn the steps to apply an LDA model to score new documents as part of an informational retrieval tool.

*Chapter 12, LDA Visualization*, presents the visualization for LDA. This chapter starts with a design thinking for the rich content of a topic model. Then it shows how to use pyLADviz for visualization.

*Chapter 13, The Ensemble LDA for Model Stability*, investigates the root causes of the instability of LDA. It explains the Ensemble approach for LDA and the use of Checkback DBSCAN, a clustering algorithm, to deliver a stable set of topics.

*Chapter 14, LDA and BERTopic*, presents the BERTopic modeling technique that uses an LLM-based BERT algorithm for word embeddings, UMAP for dimensionality reduction for word embedding, HDBSCAN for topic clustering, c-TFIDF for word presentation for topics, and MMR to fine-tune the word representation for topics. It guides you through BERT modeling, visualization, and scoring new documents for topics.

*Chapter 15, Real-World Use Cases*, presents seven NLP projects in healthcare, medical, legal, finance, and social media. By learning these NLP solutions, you will be motivated to apply code notebooks of this book to perform similar jobs or apply to your future applications.

## To get the most out of this book

You'll need to make sure that you have the following setup requirements fulfilled in order to follow the instructions given in this book:

Software/hardware covered in the book	Operating system requirements
Python version $\geq 3.7$	Windows, macOS, or Linux
Gensim	

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

## Download the example code files

The Python notebooks are available for download at <https://github.com/PacktPublishing/The-Handbook-of-NLP-with-Gensim>. If there's an update to the code, it will be updated in the GitHub repository. You are encouraged to use Google Colab. Google Colab is a free Jupyter Notebook environment that runs entirely in the cloud. Google Colab has already pre-installed popular machine-learning libraries such as pandas, NumPy, TensorFlow, Keras, and OpenCV.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

## Data for this book

The AG's corpus of news articles, made public by A. Gulli, is a collection of more than 1 million news articles from more than 2,000 news sources. Zhang, Zhao, and LeCun sampled news articles from on “world”, “sports”, “business”, and “Science” categories. This dataset `ag_news` is a frequently used dataset and is available in Kaggle, PyTorch, Huggingface, and Tensorflow. There are 120,000 and 7,600 news articles in the training and testing samples respectively. This dataset is used throughout the book.

## Conventions used

There are a number of text conventions used throughout this book.

**Code in text:** Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “Mount the downloaded WebStorm-10\*.dmg disk image file as another disk in your system.”

A block of code is set as follows:

```
import gensim
from gensim.utils import simple_preprocess
from gensim.corpora import Dictionary
import pprint
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
from gensim.summarization import keywords
```

Any command-line input or output is written as follows:

```
pip install gensim==3.8.3
```

**Bold:** Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: “Without **natural language processing (NLP)** tools, the marketing team can only do basic operations with these text messages and data.”

### Tips or important notes

Appear like this.

## Get in touch

Feedback from our readers is always welcome.

**General feedback:** If you have questions about any aspect of this book, email us at [customercare@packtpub.com](mailto:customercare@packtpub.com) and mention the book title in the subject of your message.

**Errata:** Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit [www.packtpub.com/support/errata](http://www.packtpub.com/support/errata) and fill in the form.

**Piracy:** If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [copyright@packtpub.com](mailto:copyright@packtpub.com) with a link to the material.

**If you are interested in becoming an author:** If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit [authors.packtpub.com](http://authors.packtpub.com).

## Share Your Thoughts

Once you've read *The Handbook of NLP with Gensim*, we'd love to hear your thoughts! Please click [here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

## Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781803244945>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

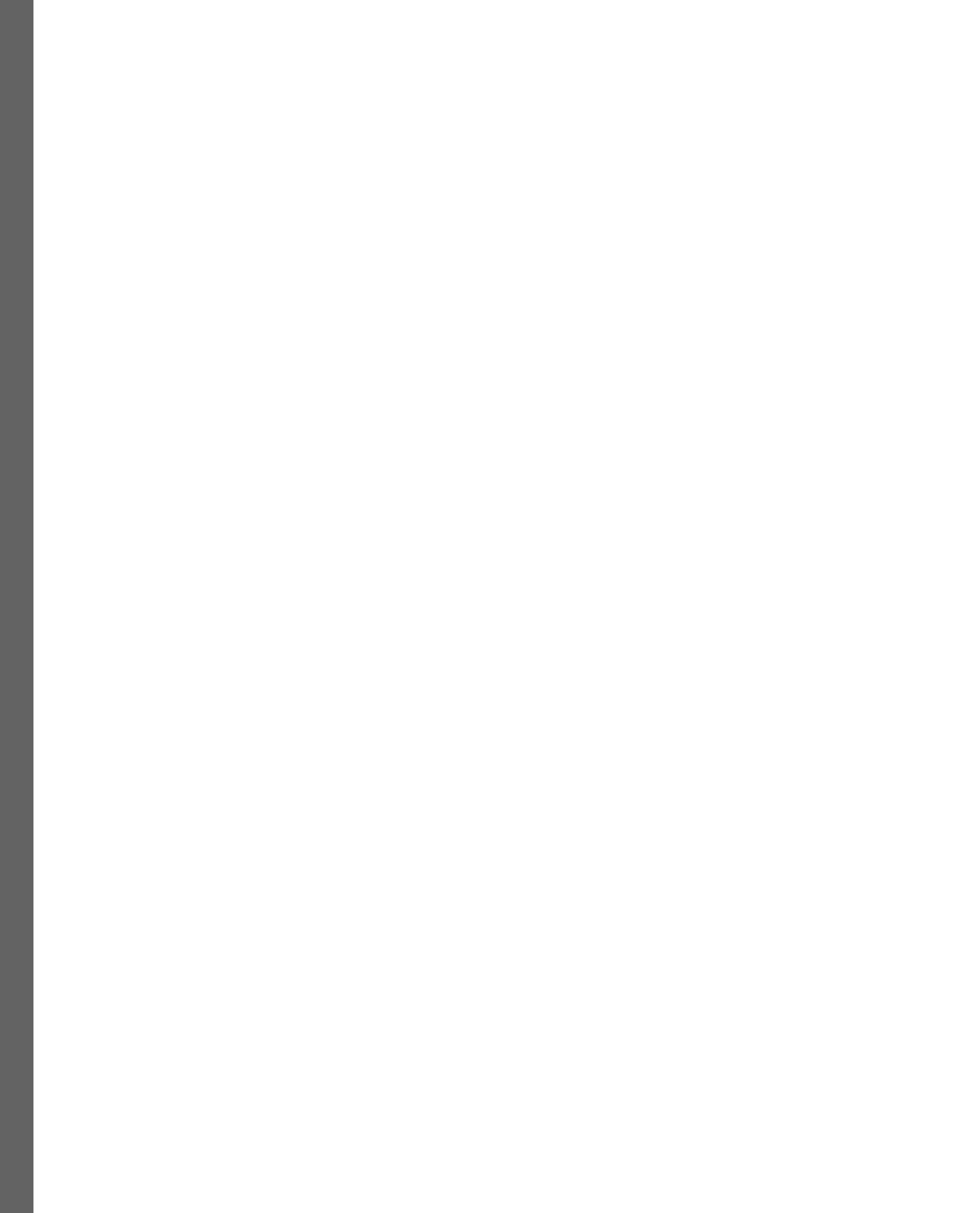


# Part 1: NLP Basics

In this part, you will get an overview of NLP. You will understand the concept of text representation and learn two basic forms of word embeddings. You will learn the key steps in NLP preprocessing including tokenization, lowercase conversion, stop words removal, punctuation removal, stemming, and lemmatization. You will learn how to do coding with spaCy, NLTK, and Gensim, and know how to build a pipeline applicable for any NLP preprocessing in the future.

This part contains the following chapters:

- *Chapter 1, Introduction to NLP*
- *Chapter 2, Text Representation*
- *Chapter 3, Text Wrangling and Preprocessing*



# 1

## Introduction to NLP

“Why do we need NLP?” You may ask this question as you’ve witnessed the advancement of **natural language processing (NLP)** in recent years. Let’s see how NLP helped a well-established investment firm named “Harmony Investments.” For decades, Harmony Investments had been renowned for its astute financial strategies and portfolio management, ranging from stocks and bonds to real estate and alternative investments. However, the sheer volume and variety of data sources, including news articles, earnings reports, social media posts, and financial statements, made it nearly impossible to manually analyze all the information. The firm’s analysts were spending an excessive amount of time collecting and reviewing data. Recognizing the need for a more efficient and data-driven approach, the firm partnered with a leading AI solutions provider to implement NLP-driven solutions into their business operations. They used NLP algorithms to review news articles, press releases, and social media platforms in real time. This analysis enabled the firm to react swiftly. They used NLP tools that automatically summarized lengthy earning reports. This reduced the time the analysts spent on manual document review. They used NLP-powered sentiment analysis to gauge public sentiment surrounding specific stocks or market segments. Analysts had more time for strategic research and developing innovative investment strategies. As a result, Harmony Investments not only retained its reputation as a leading investment firm but also attracted new clients and expanded its portfolio.

Joe is a data scientist who is new to NLP. He and his data analyst colleague, Jacob, are interested in learning NLP techniques. They want to acquire the NLP techniques that can deliver the NLP benefits as discussed. They have certainly heard of **ChatGPT** and all the news about **large language models (LLMs)**. They want to learn NLP systematically, from concepts to practice, and want to find a textbook that can bridge them to LLMs without diving into LLMs first. If you are like Joe or Jacob, then this book is for you.

A fundamental step in NLP for computers to understand texts is **text representation**, which convert a collection of text documents into numerical values. Each document is represented as a vector in a high-dimensional space, where each dimension corresponds to a unique word in the entire corpus. This helps computers understand what words mean and how they relate to each other in sentences. This book starts with **bag-of-words (BoW)**, **bag-of-N-grams**, **term frequency-inverse document frequency (TF-IDF)**. An advance to text representation is the **word embedding** techniques. Word embeddings are dense vector representations of words that capture semantic relationships between

words based on their context in a large dataset. Word embeddings, like Word2Vec, create continuous vector representations where words with similar meanings have similar vector representations, and they capture semantic and syntactic relationships.

Topic modeling is a significant NLP subject. It classifies documents into topics for document retrieval, categorization, tagging, or annotation. This book gives more insight into the milestone topic modeling technique, **Latent Dirichlet Allocation (LDA)**. In addition, another milestone topic modeling technique is **BERTopic**. Let me briefly describe the development history of **Bidirectional Encoder Representations from Transformers (BERT)**. The seminal paper “*Attention is all you need*” by Vaswani et al. [2] enables many transformer-based word embeddings and LLMs. One of the word embeddings is BERT. Can we do topic modeling to classify documents based on BERT word embeddings? That’s the origin of BERTopic. I have included BERTTopic in this book together with LDA so you get to see the differences. This will provide a bridge to the transformer-based NLP techniques.

This book is a practical handbook with code snippets. I will cover many techniques in the **Gensim** library. Gensim is an open source Python library for topic modeling, document clustering, and other unsupervised learning tasks on collections of textual documents. It provides a high-level interface for building and training a variety of models. Gensim stands for **generate similar**. It finds the similarities between documents to summarize texts or to classify documents into topics.

In this chapter, we will cover the following topics:

- Introduction to natural language processing
- NLU + NLG = NLP
- Gensim and its NLP modeling techniques
- Topic modeling with BERTopic
- Common NLP Python modules included in this book

After completing this chapter, you will get to know the development history of NLP. You will be able to explain the key NLP techniques that Gensim covers. You will also understand other popular NLP Python libraries that are often used together.

## Introduction to natural language processing

NLP is based on 50 years of rich research into linguistics and processing algorithms. It is a branch of computer science or **artificial intelligence (AI)** that uses computer algorithms to analyze, understand, and generate human language data. The algorithms process human language to “understand” its full meaning. NLP has a wide range of applications that include the following:

- **Text mining:** Extracting information from large amounts of text data, such as documents, emails, and social media posts.
- **Information retrieval:** Searching for relevant information in large text databases. In this book, you will learn many techniques for information retrieval.

- **Question answering:** Answering questions posed in natural language.
- **Machine translation:** Translating text from one language to another.
- **Sentiment analysis:** Identifying the tone and emotion of text data.
- **Natural language generation (NLG):** Generating text that mimics human language.

As I said before, NLP has a long development history. Let's look into it briefly.

## NLU + NLG = NLP

NLP is an umbrella term that covers **natural language understanding (NLU)** and NLG. We'll go through both in the next sections.

### NLU

Many languages, such as English, German, and Chinese, have been developing for hundreds of years and continue to evolve. Humans can use languages artfully in various social contexts. Now, we are asking a computer to understand human language. What's very rudimentary to us may not be so apparent to a computer. Linguists have contributed much to the development of computers' understanding in terms of syntax, semantics, phonology, morphology, and pragmatics.

NLU focuses on understanding the meaning of human language. It extracts text or speech input and then analyzes the syntax, semantics, phonology, morphology, and pragmatics in the language. Let's briefly go over each one:

- **Syntax:** This is about the study of how words are arranged to form phrases and clauses, as well as the use of punctuation, order of words, and sentences.
- **Semantics:** This is about the possible meanings of a sentence based on the interactions between words in the sentence. It is concerned with the interpretation of language, rather than its form or structure. For example, the word “table” as a noun can refer to “a piece of furniture having a smooth flat top that is usually supported by one or more vertical legs” or a data frame in a computer language.

Let's elaborate more on semantics with two jokes. The first example is as follows:

Patient: “*Doctor, doctor! I've broken my arm in three places!*”

Doctor: “*Well, stop going to those places, then.*”

The patient uses the word *places* to mean the spots on the arm; the doctor uses *places* to mean physical locations.

The second example is as follows:

My coworker: “*Do you ever think about working from home?*”

Me: “*I don't even think about work at work!*”

The first *work* in my reply means the tasks in my work. The second *work* in *at work* means *at one's place of employment*.

NLU can understand the two meanings of a word in such jokes through a technique called **word embedding**. We will learn more about this in *Chapter 2, Text Representation*.

- **Phonology:** This is about the study of the sound system of a language, including the sounds of speech (phonemes), how they are combined to form words (morphology), and how they are organized into larger units such as syllables and stress patterns. For example, the sounds represented by the letters “p” and “b” in English are distinct phonemes. A phoneme is the smallest unit of sound in a language that can change the meaning of a word. Consider the words “pat” and “bat.” The only difference between these two words is the initial sound, but their meanings are different.
- **Morphology:** This is the study of the structure of words, including the way in which they are formed from smaller units of meaning called morphemes. It originally comes from “morph,” the shape or form, and “ology,” the study of something. Morphology is important because it helps us understand how words are formed and how they relate to each other. It also helps us understand how words change over time and how they are related to other words in a language. For example, the word “unkindness” consists of three separate morphemes: the prefix “un-,” the root “kind,” and the suffix “-ness.”
- **Pragmatics:** This is the study of how language is used in a social context. Pragmatics is important because it helps us understand how language works in real-world situations, and how language can be used to convey meaning and achieve specific purposes. For example, if you offer to buy your friend a McDonald’s burger, a large fries, and a large drink, your friend may reply “no” because he is concerned about becoming fat. Your friend may simply mean the burger meal is high in calories, but the conversation can also imply he may be fat in a social context.

Now, let's understand NLG.

## NLG

While NLU is concerned with *reading* for a computer to comprehend, NLG is about *writing* for a computer to write. The term *generation* in NLG refers to an NLP model generating meaningful words or even articles. Today, when you compose an email or type a sentence in an app, it presents possible words to complete your sentence or performs automatic correction. These are applications of NLG. As a result, the term **generative AI** is coined for generative models including language, voice, image, and video generation. ChatGPT and GPT-4 from **OpenAI** are probably the most famous examples of generative AI. I will briefly introduce ChatGPT, GPT-4, and other open source products, such as gpt.h2o.ai, and present the **HuggingFace.co** open source community.

## **ChatGPT and GPT-4**

You can enter a prompt for ChatGPT to generate a poem, a prompt, a story, and so on. With its use of the **reinforcement learning from human feedback (RLHF)** technique, ChatGPT is designed to respond to questions in a way that sounds natural and human-like, making it easier for people to communicate with the model. It is also able to remember the conversations prior to the current conversation. If you ask ChatGPT, “*Should I wear a blue shirt or a white shirt for tomorrow’s outdoor company meeting when it is likely to be hot and sunny?*”, it will formulate a response by inferring a sequence of words that are likely to come next. It answers, “*When it is hot and sunny, you may want to wear a white shirt.*” If you reply, “*How about on a gloomy day?*”, ChatGPT understands that you mean to ask, in continuation to your prior question, about what to wear for tomorrow’s outdoor company meeting. It does not take the second question as an independent question and answer it randomly. This ability to remember and contextualize inputs is what gives ChatGPT the ability to carry on some semblance of a human conversation rather than give naïve, one-off answers. Hence, having the memory for long context is the key for next-generation language models such as GPT-4.

**Generative Pre-trained Transformer 4 (GPT-4)** was released by OpenAI on March 14, 2023. It is a transformer-based model that can predict the next word or token. It can answer questions, summarize text, translate text to other languages, and generate code, blog posts, stories, conversations, and other content types. The ability to remember and contextualize inputs, known as the **context window** is key for a language model. The context window of GPT-4 has been increased to roughly 8,000 words. Given the English conversation speed is about 120 words per minute and 7,200 words per hour, a context window of 8,000 words is more than an hour’s conversation.

What is the token or transformer mentioned in the previous paragraph? A **token** is the NLP term referring to a word separated from a sentence, or **tokenized**. A **transformer** is the milestone language model invented in 2017 by Vaswani et al [2]. In *Chapter 2, Text Representation*, I will explain how tokenization works in text preprocessing. In *Chapter 13, The Ensemble LDA for Model Stability*, I will give high-level explanations of a transformer model and others including the BERTopic model that builds on it.

## **Gpt.h2o.ai**

**GPT-H2O.ai** is an LLM by H2O.ai. It is aimed at enterprise-grade LLMs such as ChatGPT, but is open source. I let gpt.h2o.ai define itself by asking “What is gpt.h2o.ai?” It answered: “gpt.h2o.ai is a large language model that was trained on a combination of web crawl data and Reddit comments. The model can generate human-like text and has been used for a variety of tasks, including generating code, answering questions, and summarizing text.”

Let's look at its storytelling capability. I entered: "Write a fable on a lion that rescued a boy when the boy was under the attack of aliens." It returned the following interesting story:

"Once upon a time, there was a young boy named Jack who loved to explore the woods near his home. One day, while he was out on one of his adventures, he came across a strange sight. There, in the middle of the forest, was a spaceship. It was unlike anything Jack had ever seen before. As he approached the ship, he heard a strange noise coming from inside. Jack peered through the window and saw a group of aliens inside. They were holding a young boy captive and seemed to be doing something to him. Jack knew he had to act quickly. Just then, a lion appeared out of nowhere. The lion roared loudly, scaring the aliens away. The boy was freed and the spaceship took off into the sky. The boy thanked the lion for saving him and asked how he knew to come to his rescue. The lion explained that it had been watching over him for some time and knew that he was in danger. The boy and the lion became friends. Jack never forgot the day that he saw the lion save the boy from the aliens."

### ***HuggingFace.co***

New research on LLMs has burgeoned over the past few years. New LLMs, datasets, and solutions become hard to keep up with and create a new knowledge barrier for anyone who wants to enter the world of generative AI. Hugging Face provides a platform called HuggingFace.co that allows everyone to access open source LLMs, academic papers, and various datasets that trained the LLMs on its platform. You can also share your LLMs and datasets.

You may ask why LLMs can answer questions, write book reports, draft notes, or summarize documents. An important factor is the data on which they were trained or with which they were fine-tuned. These large-scale datasets include "News/Wikipedia," "Web crawling," "Questions and Answers (Q&A)," "Books," and "Reading comprehension." In "Large Language Model Datasets" [3], I have given a more detailed review of the datasets that trained prominent LLMs such as GPT-2, GPT-3, GPT-4, and so on.

I trust you have gained a better understanding of NLU and NLG. Next, I will introduce the NLP techniques covered by Gensim.

## **Gensim and its NLP modeling techniques**

Gensim is actively maintained and supported by a community of developers and is widely used in academic research and industry applications. It covers many important NLP techniques that make up the workforce of today's NLP. That's one of the reasons why I have developed this book to help data scientists.

Last year, I was at a company's year-end party. The ballroom was filled with people standing in groups with their drinks. I walked around and listened for conversation topics where I could chime in. I heard one group talking about the FIFA World Cup 2022 and another group talking about stock markets. I joined the stock markets conversation. In that short moment, my mind had performed "word extractions," "text summarization," and "topic classifications." These tasks are the core tasks of NLP and what Gensim is designed to do.

We perform serious text analyses in professional fields including legal, medical, and business. We organize similar documents into topics. Such work also demands “word extractions,” “text summarization,” and “topic classifications.” In the following sections, I will give you a brief introduction to the key models that Gensim offers so you will have a good overview. These models include the following:

- BoW and TF-IDF
- Latent semantic analysis/indexing (LSA/LSI)
- Word2Vec
- Doc2Vec
- Text summarization
- LDA
- Ensemble LDA

## BoW and TF-IDF

Texts can be represented as a bag of words, which is the count frequency of a word. Consider the following two phrases:

- **Phrase 1:** All the stars we steal from the night sky
- **Phrase 2:** Will never be enough, never be enough, never be enough for me

The BoW presents the word count frequency as shown in *Figure 1.1*. For example, the word *the* in the first sentence appears twice, so it is coded as 2; the word *be* in the second sentence appears three times, so it is coded as 3:

Count Vector	All	be	enough	from	for	me	Never	night	sky	stars	steal	the	we	Will
All the stars we steal from the night sky	1	0	0	1	0	0	0	1	1	1	1	2	1	0
Will never be enough, Never be enough, Never be enough for me	0	3	3	0	1	1	3	0	0	0	0	0	0	1

Figure 1.1 – BoW encoding (also presented in the next chapter)

BoW uses the word count to reflect the significance of a word. However, this is not very intuitive. Frequent words may not carry special meanings depending on the type of document. For example, in clinical reports, the words *physician*, *patient*, *doctor*, and *nurse* appear frequently. The high frequency of these words may overshadow specific words such as *bronchitis* or *stroke* in a patient’s document. A better encoding system is to compare the relative word appearance in a document to its appearance throughout the corpus. TF-IDF is designed to reflect the importance of a word in a document by calculating its relevance to a corpus. We will learn the details of this in *Chapter 2, Text Representation*.

At this moment, you just need to know that both BoW and TF-IDF are variations of text representation. They are the building blocks of NLP.

Although BoW and TF-IDF appear simple, they already have real-world applications in different fields. An important application of BoW and TF-IDF is to prevent spam emails from going to the inbox folder of an email account. Spam emails are ubiquitous, unavoidable, and quickly fill up the spam folder. BoW or TF-IDF will help to distinguish the characteristics of a spam email from regular emails.

## LSA/LSI

Suppose you were a football fan and searching in a library using the keywords *famous World Cup players*, and that the system can only do exact key word match. The old computer system returned all articles that contained *famous*, *world*, *cup*, or *players*. It also returned a lot of other unrelated articles such as *famous singer*, *150 most famous poems*, and *world-renowned scientist*. This is terrible, isn't it? A simple keyword match cannot serve as a search engine.

**Latent semantic analysis (LSA)** was developed in the 1990s. It's an NLP solution that far surpasses naïve keyword matching and has become an important search engine algorithm. Prior to that, in 1988, an LSA-based information retrieval system was patented (US Patent #4839853, now expired) and named "latent semantic indexing," so the technique is also called **latent semantic indexing (LSI)**. Gensim and many other reports name LSA as LSI so as not to confuse LSA with LDA. In this book, I will adopt the same naming convention. In *Chapter 6, Latent Semantic Indexing with Gensim*, I will show you the code example to build an LSI model.

You can search with keywords such as the following:

*Crude prices inflation the economy outlook earnings*

This can return relevant news articles. One of the results is as follows:

*A huge jump in wholesale prices sent stocks falling yesterday as investors worried that rising oil prices were taking a toll on the overall economy. (Data source: AG news data)*

Notice it searches by meaning but not by word matching.

## Word2Vec

The Word2Vec technique developed by Mikolov et al. [4] in 2014 was a significant milestone in NLP. Its idea was ground-breaking—it embeds words or phrases from a text corpus as dense, continuous-valued vectors, hence the name word-to-vector. These vector representations capture semantic relationships and contextual information between words. Its applications are prevalent in many recommendation systems. *Figure 1.2* shows other words that are close to the word *iron* including *gunpowder*, *metals*, and *steel*; words far from *iron* are *organic*, *sugar*, and *grain*:

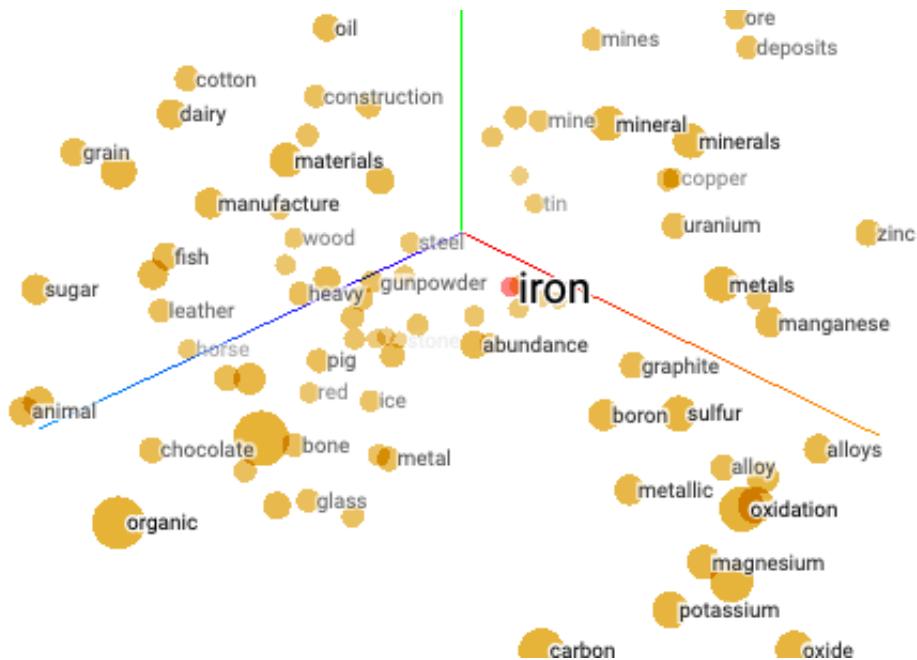


Figure 1.2 – An overview of Word2Vec (also presented in Chapter 7)

Also, the relative distance of words measures the similarity of meanings. Word2Vec enables us to measure and visualize the similarities or dissimilarities of words or concepts. This is a fantastic innovation.

Can you see how this idea can also apply to movie recommendations? Each movie can be considered a word in someone's watching history. I Googled the words *movie recommendations*, and it returned many movies under "Top picks for you":



Figure 1.3 – An overview of Word2Vec as a movie recommendation system (also presented in Chapter 7)

## Doc2Vec

Word2Vec represents a word with a vector. Can we represent a sentence or a paragraph with a vector? **Doc2Vec** is designed to do so. Doc2Vec transforms articles into vectors and enable semantic search for related articles. Doc2Vec has enabled many commercial products. For example, when you search for a job on LinkedIn.com or Indeed.com, you see similar job postings presented next to your target job posting. It is done by Doc2Vec. In *Chapter 8, Doc2Vec with Gensim*, you will build a real Doc2Vec model with code examples.

## LDA

When documents are tagged by topic, we can retrieve the documents easily. In the old days, if you went to a library for books of a certain genre, you used the indexing system to find them. Now, with all the digital content, documents can be tagged systematically by topic modeling techniques.

The preceding library example may be an easier one, if compared to all sorts of social media posts, job posts, emails, news articles, or tweets. Topic models can tag digital content for effective searching or retrieving. LDA is an important topic modeling technique and has many commercial use cases. *Figure 1.4* shows a snapshot of the LDA model output that we will build in *Chapter 11, LDA Modeling*:

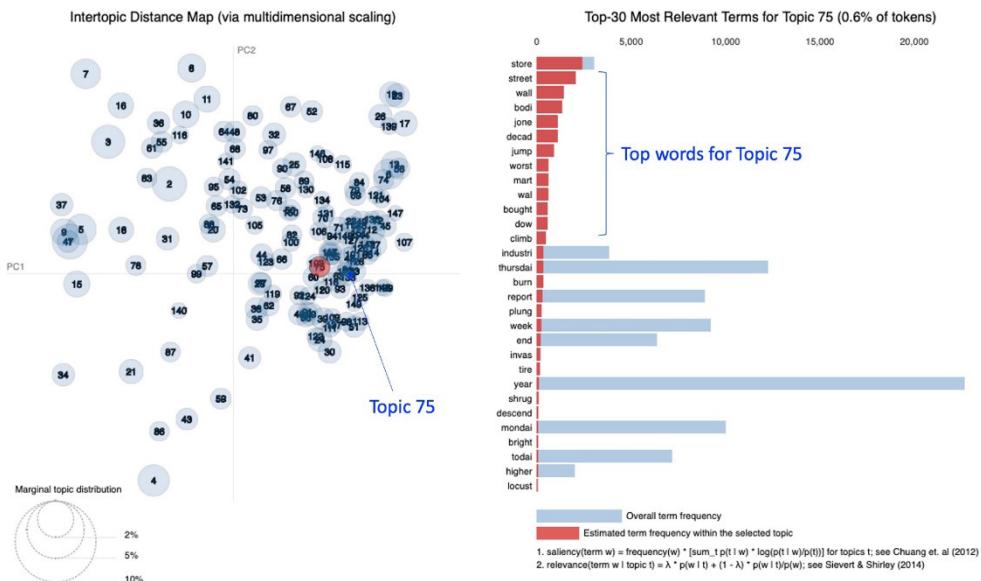


Figure 1.4 – pyLDAvis (also presented in Chapter 11)

Each bubble represents a topic. The distance between any two bubbles represents the difference between the two topics. The red bars on top of the blue bars represent the estimated frequency of a word for a chosen topic. Documents on the same topic are similar in their content. If you are reading

a document that belongs to **Topic 75** and want to read more related articles, LDA can return other articles on **Topic 75**.

## Ensemble LDA

The goal of topic modeling for a set of documents is to find topics that are reliable and reproducible. If you replicate the same modeling process for the same documents, you expect to produce the same set of topics. However, past experiments have shown that while iterations for the same model produce the same set of topics, some iterations can produce extra topics. This creates a serious issue in practice: Which model outcome is the correct one to use? This issue seriously limits the applications of LDA. So, we think of the ensemble method in machine learning. In machine learning, an ensemble is a technique that combines multiple individual models to improve predictive accuracy and generalization performance. **Ensemble LDA** builds many models to identify a core set of topics that is reliable and reproducible all the time. In *Chapter 13, The Ensemble LDA for Model Stability*, I will explain the algorithm with visual aids in more detail. We also will build our own model with code examples.

## Topic modeling with BERTopic

BERTopic is a topic modeling algorithm that is based on the BERT word embeddings. In *Chapter 14, LDA and BERTopic*, we will learn the key components of BERTopic and build our own model. In addition, the BERTopic modeling has its own visualization functions that are similar to pyLDAvis, as seen in *Figure 1.4*. We will learn to use all the visualization functions as well.

*Figure 1.5* shows the top words for eight topics:

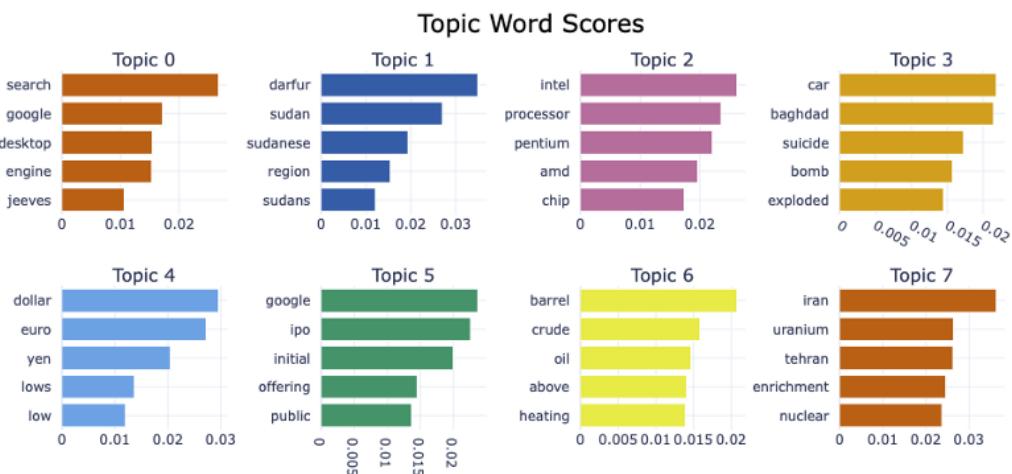


Figure 1.5 – An overview of topic modeling results by BERTopic (also presented in Chapter 14)

I trust these introductions have given you a strong appetite to dive into each chapter and apply the models discussed in your future work. Now, let's get familiar with the terminology commonly used in NLP.

## Common NLP Python modules included in this book

This book includes a few Python modules for the best learning outcomes. If an NLP task can be performed by other libraries, such as `scikit-learn` or `NLTK`, I will show you the code examples for comparison. The libraries included in this book are detailed in the following sections.

### **spaCy**

`spaCy` is by far the best production-level, open source library for NLP. It makes many processing tasks easy with reliable code and outcomes. If you work with a large volume of texts for text preprocessing, `spaCy` is an excellent choice. It is designed to be a simple and concise alternative to C.

It can perform a wide range of NLP operations well. These NLP operations include the following tasks:

- **Tokenization:** This breaks text into individual words or tokens. To a computer, a sentence is just a string of characters. The string has to be separated into words.
- **Part-of-speech (PoS) tagging:** This assigns grammatical labels to each word in a sentence. For example, the sentence “She loves the beautiful flower” has a pronoun (“she”), a verb (“loves”), an adjective (“beautiful”), and a noun (“flower”). The labeling for the pronoun, verb, adjective, and noun is called PoS tagging.
- **Named entity recognition (NER):** This identifies named entities such as names, organizations, locations, and so on. For example, in the sentence “I went to New York City on July 4<sup>th</sup>,” the named entities would be “New York City” (a place), and “July 4<sup>th</sup>” (a date). It is worth mentioning that `spaCy`’s built-in NER models are based on the BERT architecture. As we will learn about BERT in this book, it is helpful to be aware of this.
- **Lemmatization:** This reduces words to their base or dictionary form. We will learn more about lemmatization in *Chapter 3, Text Wrangling and Preprocessing*.
- **Rule-based matching:** This can find sequences of words based on user-defined rules.
- **Word vectors:** These represent words as numerical vectors. When two words become vectors, they can be compared in the vector space. Word embedding and vectorization is an important step in NLP. `spaCy` provides the functions to do so. We will learn about the concept and practice of word vectorization in *Chapter 7, Using Word2Vec*.

`spaCy` can be easily integrated with other libraries such as `Gensim` and `NLTK`. That’s why in many code examples you see that `spaCy`, `Gensim`, and `NLTK` are used together.

These are just some of the main capabilities of spaCy, and it offers many more features and functionalities for NLP tasks.

## NLTK

NLTK is an open source Python library for natural language processing. It provides a suite of tools for working with text data, including tokenization, PoS tagging, and NER. It provides interfaces to over 50 corpora and lexical resources, such as **WordNet**. NLTK also includes a number of pre-trained models for tasks such as sentiment analysis and topic modeling. It is widely used in academia and industry for research and development in NLP. NLTK can perform a range of NLP tasks too, including PoS, NER, sentiment analysis, text classification, and text summarization.

## Summary

This chapter provided a landscape view of the NLP topics covered in this book. We learned that the development of NLP was due to the success of NLU and NLG. Then, we surveyed the NLP techniques that are covered by Gensim. The main techniques include BoW, TF-IDF, LSA/LSI, Word2Vec, Doc2Vec, LDA, and Ensemble LDA. We were also introduced to BERTopic modeling. We then learned about the other two popular NLP Python libraries, spaCy and NLTK.

As we all know, a computer operates on zeros and ones but cannot comprehend the great works of Shakespeare. So, how do ChatGPT and other language models understand language? The very first step is to convert words to numerical values. The next chapter will teach you about text representation.

## Questions

1. Describe natural language processing (NLP).
2. What is natural language understanding (NLU)?
3. What is natural language generation (NLG)?
4. List some of the NLP modeling techniques used by Gensim.
5. List some of the most used NLP Python modules.
6. Once you have answered the previous questions, let's access <https://chat.openai.com/> to search for answers. This time, let's key in a question, called a "prompt," to get answers. You are encouraged to experiment with ChatGPT with variations of the questions. For example, you can test the following for Question 1:
  - I. "Please describe natural language processing."
  - II. "Please describe NLP to a high schooler."
  - III. "Please describe NLP in one paragraph."
  - IV. "Please describe NLP with an analogy."

## References

1. Wei, Low De (2022, December 2). This AI Chatbot Is Blowing People's Minds. Here's What It's Been Writing. Bloomberg.com. <https://www.bloomberg.com/news/articles/2022-12-02/chatgpt-openai-s-new-essay-writing-chatbot-is-blowing-people-s-minds?leadSource=uverify%20wall>
2. Vaswani, A., Shazeer, N.M., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need. ArXiv, abs/1706.03762.
3. Kuo, Chris, (2023) Large Language Model Datasets, May 9, 2023, <https://dataman-ai.medium.com/large-language-model-datasets-95df319a110>
4. Mikolov, T., Chen, K., Corrado, G.S., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *International Conference on Learning Representations*. <https://arxiv.org/abs/1310.4546>
5. Devlin, J., Chang, M., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *ArXiv*, *abs/1810.04805*.
6. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T.J., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., & Amodei, D. (2020). Language Models are Few-Shot Learners. *ArXiv*, *abs/2005.14165*.

# 2

## Text Representation

A computer operates on zeros and ones, and algorithms operate on numerical values. A computer does not understand beautiful texts such as the plays by William Shakespeare or the books by Leo Tolstoy. So, raw texts need to be converted to numerical values for a computer to process. The first step in NLP is converting texts to numerical values.

In this chapter, we will learn about the basic text representation – Bag-of-Words, Bag-of- $N$ -grams, and TF-IDF. This chapter is for absolute NLP beginners. In this chapter, we will learn how to code with Gensim, scikit-learn, and NLTK. We will cover the following topics:

- What text representation is
- The transition from one-hot encoding to Bag-of-Words to Bag-of- $N$ -grams
- What TF-IDF is
- How to perform **Bag-of-Words (BoW)** and TF-IDF encoding in Gensim
- The real-world applications of BoW and TF-IDF

By the end of this chapter, you will be able to describe the BoW, Bag-of- $N$ -grams, and TF-IDF methods and their advantages or disadvantages. You will be able to name some real-world applications of BoW and TF-IDF, and you will be able to program raw texts using these techniques in Gensim, scikit-learn, and NLTK accordingly.

### Technical requirements

You can find the code files for this chapter on GitHub at <https://github.com/PacktPublishing/The-Handbook-of-NLP-with-Gensim/tree/main/Chapter02>.

## What word embedding is

Word embedding is an innovative representation of text, where it is represented by a real-valued vector. The term *embed* means to “fix an object firmly and deeply in a surrounding mass” according to Dictionary . com. This interesting term was first coined by Bengio et al. in 2003 [1]. The ability to represent words and documents in a vector form is a key breakthrough of NLP. Texts can be computed in very sophisticated neural networks for addition, subtraction, comparison, or generation.

Let’s take text comparison as an example. When texts become vectors, we can calculate the distances between them. Texts that are closer in the vector space are expected to be similar in meaning. In *Chapter 7, Using Word2Vec*, we will learn that the words “gun,” “gunpowder,” and “steel” are closely related because their vectors are also close, while the words “egg,” “gun,” and “grass” are distant from each other because their vectors are quite different.

Now that we know what word embedding is, let’s start with the basic encoding methods.

## Simple encoding methods

As the saying goes, “*All great endeavors commence from ground zero*,” and NLP’s ground zero is encoding. There are many encoding techniques to effectively represent words with the right contexts or NLU meaning. Let’s start with the three simplest encoding methods – one-hot encoding, BoW, and Bag of *N*-grams.

### One-hot encoding

We can do one-hot encoding for texts. It is also called count vectorizing. The idea is very simple – we create a vector whose length is the number of unique words in the entire text. At the time of writing this chapter, on a quiet evening, I am listening to the song *Never Enough* from *The Greatest Showman* (<https://www.imdb.com/title/tt1485796/>). Let me use its lyric as an example:

“All the stars we steal from the night sky

Will never be enough, never be enough, never be enough for me.”

Here, there are two sentences. Each sentence will be converted to a vector. The length of the vector is the total number of unique words in the entire article. We will first create a set of unique words, as shown in *Figure 2.1*. Then, we will perform one-hot encoding on the two sentences. If a word is present, it is coded as 1; otherwise, it is 0.

One-Hot Encoding	All	be	enough	from	for	me	Never	night	sky	stars	steal	the	we	Will
All the stars we steal from the night sky	1	0	0	1	0	0	0	1	1	1	1	1	1	0
Will never be enough, Never be enough, Never be enough for me	0	1	1	0	1	1	1	0	0	0	0	0	0	1

Figure 2.1 – One-hot encoding

However, something is missing when the values are only binary. Some words actually appear more than once and can convey important information. We can improve the encoding by using BoW. Let's do that now.

## BoW

Texts can be represented as a BoW. This is the count frequency of a word, as shown in *Figure 2.2*. The word “the” in the first document appears twice so is coded as 2; the word “be” in the second document appears three times so is coded as 3. We will practice the code in the *Coding – BoW* section.

Count Vector	All	be	enough	from	for	me	Never	night	sky	stars	steal	the	we	Will
All the stars we steal from the night sky	1	0	0	1	0	0	0	1	1	1	1	2	1	0
Will never be enough, Never be enough, Never be enough for me	0	3	3	0	1	1	3	0	0	0	0	0	0	1

Figure 2.2 – BoW encoding

Similar to the issue with one-hot encoding, the BoW method ignores the order of the words as well as any grammar rules. However, although it appears primitive, it can effectively classify different kinds of documents. Different types of documents should have different words. The occurrence and frequency of a word in a document should characterize the type of article.

Let's take phrases. The words in a phrase such as “New York” work better together. If they are separated as “New” and “York,” they lose their special meaning. Let's see how the Bag-of-N-grams method can improve upon BoW.

## Bag-of-N-grams

Two consecutive words in a sentence are called a *bigram*, and three consecutive words are called a *trigram*. Not all two or three words in a sequence are meaningful, but sometimes they are. When this happens, it is more informative to use Bag-of-N-grams rather than BoW. For example, let's take the sentence "Let us join the New York City marathon in 2022":

- The unigrams are "Let," "us," "join," "the," "New," "York," "City," "marathon," "in," and "2022"
- The bigrams are "Let us," "us join," "join the," "the New," "New York," "York City," "City marathon," "marathon in," and "in 2022"
- The trigrams are "Let us join," "us join the," "join the New," "the New York," "New York City," "York City marathon," "City marathon in," and "marathon in 2022"

The phrases "Let us join," "New York," and "New York City" are common phrases that could appear in other documents. Bigrams or trigrams help to capture additional information beyond unigrams. In theory, we can create combinations of words. However, typically more than three words are not that meaningful. It is usually enough to do up to 3 grams. We will do this in `models.phrases` in Gensim.

## What TF-IDF is

One-hot encoding simply records the presence of a word but does not reflect any of its relative importance. BoW is an improvement over one-hot encoding by measuring word frequency. However, word frequency does not imply word importance. For example, in *Figure 2.2*, the word "the" appears twice in the first sentence, and "be" appears three times in the second sentence, but they do not add any specific color to the poem. In linguistics, it is often the case that words that appear less carry more distinctive meanings. The terms "shining," "steal," "night," and "sky" paint a picture vividly in our poem. Can we improve upon one-hot encoding or BoW?

**Term frequency-inverse document frequency (TD-IDF)** is designed to reflect the importance of a word in a document of a corpus. Many frequently used words such as "the," "he," "she," "we," and "they" do not convey special meanings. If word frequency is used to refer to the significance of a word, it cannot be captured well. Also, professional documents have frequently used words. For example, in a clinical document, the words "patient," "doctor," and "nurse" could appear frequently and overshadow more informative words that refer to the diagnosis of a patient, such as "bronchitis" or "stroke."

TD-IDF calculates the relevance of a word in a document to a corpus. TF is the term frequency of a word in a document. IDF is the inverse document frequency of the word throughout a corpus. When the IDF of a word is close to 0, the word is a common word throughout the corpus. Otherwise, it will approach 1. It is calculated as follows:

$$w_{ij} = \text{TF} \times \text{IDF} = tf_{ij} \cdot \log\left(\frac{N}{df_i}\right) \quad \text{Eq. (1)}$$

Here:

- $w_{ij}$  = the TF-IDF score of token  $i$  in document  $j$
- $tf_{ij}$  = the number of occurrences of token  $i$  in document  $j$
- $df_i$  = the number of documents that contain token  $i$
- $N$  = the total number of documents

Multiplying the TF and IDF of a word will result in its TF-IDF score in a document. The higher the score, the more relevant that word is in that document. Later, in the coding section, we will learn how to produce the TF-IDF score.

## Shining applications of BoW and TF-IDF

Although BoW and TF-IDF may appear simple, they already have real-world applications. Both techniques can capture the appearance and frequency of a word in a document. Different types of documents will have different word appearance and word frequency, so they can be applied to classify documents into different types. One important application is to prevent spam emails from going to the inbox folder of an email account. Spam emails are ubiquitous, unavoidable, and can quickly fill up the spam folder. BoW or TF-IDF helps to distinguish the characteristics of a spam email from regular emails. You may ask, if BoW and TF-IDF are effective, why do we still receive spam emails? This is because spam email writers try to compose spam emails that are as close as possible to regular emails, so an algorithm cannot distinguish them from regular emails.

Besides text classification, BoW has been expanded to **Bag-of-Visual-Words (BoVW)** to classify images. Although BoVW has grown into its own line of research and is beyond the scope of this book, let me explain the basic idea. An image can be considered as a collection of many small patches that carry distinctive color information. Csurka et al. (2004) considered these small patches as visual words [2]. Hence, an image can be viewed as a BOVW. The BoVW method enables us to compare their similarities to classify them. This approach can provide reliable results for image classification. Similarly, the BoW method for visual words can be replaced by the TF-IDF method to classify images.

Having learned about the basics of these methods, let's now turn to implementing them by coding.

## Coding – BoW

I will use the classic song *Theme from New York, New York* by Frank Sinatra. Its repetition of “New York” is a good choice for NLP. At the time of writing this chapter, the jazz music of the song accompanies the calm and snowy midnight in New York.

“Start spreading the news

You’re leaving today (tell him friend)

I want to be a part of it, New York, New York

Your vagabond shoes, they are longing to stray

And steps around the heart of it, New York, New York”

Let’s learn how to do BoW with Gensim.

### Gensim for BoW

Let’s import several modules. The Gensim `simple_preprocess` function converts a document into a list of tokens. The Gensim `Dictionary()` class implements the concept of a dictionary in Gensim. It maps a tokenized word to a unique ID. I will also import `pprint` for `prettyprint`. It will print output in a prettier form:

```
import gensim
from gensim.utils import simple_preprocess
from gensim.corpora import Dictionary
import pprint
```

The opening of *Theme from New York, New York* has five sentences. I will assign the lyrics to a list called `doc_list`. The list contains five strings.

```
doc_list = [
    "Start spreading the news",
    "You're leaving today (tell him friend)",
    "I want to be a part of it, New York, New York",
    "Your vagabond shoes, they are longing to stray",
    "And steps around the heart of it, New York, New York"
]
```

I will use `pprint` and set the indent to four spaces:

```
pp = pprint.PrettyPrinter(indent=4)
pp pprint(doc_list)
```

It prints the lyrics in the original form – five lines with indents:

```
[ 'Start spreading the news',
  "You're leaving today (tell him friend)",
  'I want to be a part of it, New York, New York',
  'Your vagabond shoes, they are longing to stray',
  'And steps around the heart of it, New York, New York']
```

Note that if I use the regular Python print function, the output is not presented as nicely as pprint:

```
print(doc_list)
```

This outputs the following:

```
['Start spreading the news', "You're leaving today (tell him friend)",  

 'I want to be a part of it, New York, New York', 'Your vagabond shoes,  

 they are longing to stray', 'And steps around the heart of it, New  

 York, New York']
```

The doc\_list list has five strings. Tokenization means separating a string into a list of individual words. We will create a loop to tokenize the five strings into five lists; each list has individual words:

```
doc_tokenized = []
for doc in doc_list:
    doc_tokenized.append(simple_preprocess(doc))
doc_tokenized
```

This loop can be written more succinctly as follows:

```
doc_tokenized = [simple_preprocess(doc) for doc in
                 doc_list]
doc_tokenized
```

Now, the result changes to the following:

```
[['start', 'spreading', 'the', 'news'],
 ['you', 're', 'leaving', 'today', 'tell', 'him', 'friend'],
 ['want', 'to', 'be', 'part', 'of', 'it', 'new', 'york', 'new',
 'york'],
 ['your', 'vagabond', 'shoes', 'they', 'are', 'longing', 'to',
 'stray'],
 ['and', 'steps', 'around', 'the', 'heart', 'of', 'it', 'new', 'york',
 'new', 'york']]
```

A nice feature of Gensim is that it issues a unique ID for a token. This concept comes from a dictionary where each word has a unique ID. We will declare a dictionary as follows:

```
dictionary = Dictionary()  
dictionary
```

Then, we will perform BoW by using `doc2bow()`. This function expects the input as a list of tokenized words. It converts the list into the BoW 2-tuple format as `list of (token_id, token_count)`. The `allow_update` parameter is set as `True`. It will update the dictionary and create IDs for new words. Because the `doc_tokenized` list is a list of lists, the following code loops through the lists:

```
BoW_corpus = [dictionary.doc2bow(doc, allow_update=True)  
              for doc in doc_tokenized]  
BoW_corpus
```

The output of the first two lists looks like this:

```
[(0, 1), (1, 1), (2, 1), (3, 1)]  
[(4, 1), (5, 1), (6, 1), (7, 1), (8, 1), (9, 1), (10, 1)]
```

This output is hard to read, isn't it? Let's use the `dictionary[id]` function to show the word of an ID:

```
id_words = [(dictionary[id], count) for id, count in line]  
for line in BoW_corpus]  
print(id_words)
```

This will present an enhanced output. The first two lists are as follows:

```
[('news', 1), ('spreading', 1), ('start', 1), ('the', 1)],  
[('friend', 1), ('him', 1), ('leaving', 1), ('re', 1), ('tell', 1),  
('today', 1), ('you', 1)],
```

Note the count of a word is the same as the count in a document (in this case, a list). The word "new" in the third sentence, "I want to be a part of it, New York, New York," appears twice. If the five sentences (or, to use the NLP term, "documents") were entered as a very long string, the word "new" would be counted four times. You have to be careful not to do so, as it loses the structure of documents.

## scikit-learn for BoW (CountVectorizer)

The `sklearn.feature_extraction` module is a useful module that can extract features from datasets of text or image formats. Here, we are not dealing with images. We will use the `CountVectorizer` function. This converts a collection of text documents into a matrix of token counts:

```
from sklearn.feature_extraction.text import CountVectorizer
```

Following the syntax of scikit-learn, we will first declare the object and call it cv:

```
cv.fit_transform():
cv = CountVectorizer()
```

Then, we perform the task by using the `.fit()` function:

```
cv_fit = cv.fit_transform(doc_list)
```

The word names can be retrieved by `cv.get_feature_names_out()`:

```
word_list = cv.get_feature_names_out()
word_list
```

The words are as follows:

```
['and', 'are', 'around', 'be', 'friend', 'heart', 'him', 'it',
'leaving', 'longing', 'new', 'news', 'of', 'part', 're', 'shoes',
'spreading', 'start', 'steps', 'stray', 'tell', 'the', 'they', 'to',
'today', 'vagabond', 'want', 'york', 'you', 'your']
```

The CountVectorizer module counts the words of a collection of text documents and saves them in a matrix. Let's see that matrix now:

```
cv_fit.toarray()
```

There are five sentences in the list, so the output should have five arrays. Each array is the corresponding count to the previous word list. For example, the word “new” in the word list is the 11<sup>th</sup> element of the list. The third array shows that the 11<sup>th</sup> element is “2.” This means the word count for the word “new” in the third sentence is 2.

The first two lists are as follows:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0]
```

I can sum up the word count of all the sentences by using `.sum(axis=0)` for `cv_fit.toarray()`:

```
count_list = cv_fit.toarray().sum(axis=0)
```

Let's print out the words and the counts in a dictionary:

```
pp pprint( dict(zip(word_list, count_list)) )
```

Now, we will see the following result:

```
{ 'and': 1, 'are': 1, 'around': 1, 'be': 1, 'friend': 1, 'heart': 1,
  'him': 1, 'it': 2, 'leaving': 1, 'longing': 1, 'new': 4, 'news': 1,
  'of': 2, 'part': 1, 're': 1, 'shoes': 1, 'spreading': 1, 'start':
  1, 'steps': 1, 'stray': 1, 'tell': 1, 'the': 2, 'they': 1, 'to': 2,
  'today': 1, 'vagabond': 1, 'want': 1, 'york': 4, 'you': 1, 'your': 1}
```

Let's summarize what we have done. We have learned how to code BoW with Gensim and scikit-learn. Now, we are ready to code Bag-of-N-grams with Gensim and scikit-learn.

## Coding – Bag-of-N-grams

Any consecutive  $N$  words become an  $N$ -gram. Not all  $N$ -grams are meaningful. For example, there are uncommon phrases in this list: “Let us,” “us join,” “join the,” “the New,” “New York,” “York City,” “City marathon,” “marathon in,” and “in 2022.” The novelty of Gensim is that it tries to create common phrases for  $N$ -grams. To make this use case interesting, in the following code, I have added two more sentences to make the lyrics longer. It has the 3-gram phrase “come and visit”:

```
doc_list_Ngrams = [
    "Start spreading the news",
    "You're leaving today",
    "I want to be a part of it, New York, New York",
    "Your vagabond shoes, they are longing to stray",
    "And steps around the heart of it, New York, New York",
    "Come and visit us",
    "Come and visit the city",
]
```

Next, let's see how to code Bag-of-N-grams with Gensim.

### Gensim for N-grams

First, let's tokenize the strings in the list:

```
doc_tokenized_Ngrams = [simple_preprocess(doc) for doc in
                        doc_list_Ngrams]
```

Then, let's build the bigrams by using Gensim's `Phrases()` function. Gensim can produce “smart” output that has one word or common phrase in a document:

```
from gensim.models import Phrases
from gensim.models.phrases import Phraser
```

We will build the bigram models:

```
bigram = gensim.models.phrases.Phrases(
    doc_tokenized_Ngrams, min_count=3, threshold=10)
bigram = Phrases(doc_tokenized_Ngrams, min_count=1,
    threshold=2, delimiter=' ')
bigram_phraser = Phraser(bigram)
```

Let's print them out:

```
for sent in doc_tokenized_Ngrams:
    tokens_ = bigram_phraser[sent]
    print(tokens_)
```

The outputs are as follows:

```
['start', 'spreading', 'the', 'news']
['you', 're', 'leaving', 'today']
['want', 'to', 'be', 'part', 'of it', 'new york', 'new york']
['your', 'vagabond', 'shoes', 'they', 'are', 'longing', 'to', 'stray']
['and', 'steps', 'around', 'the', 'heart', 'of it', 'new york', 'new
york']
['come and', 'visit', 'us']
['come and', 'visit', 'the', 'city']
```

Now, let's continue to build trigrams with Gensim. Building a trigram with Gensim is not so straightforward. It is built first by bigrams and then uses the output of bigrams to build trigrams:

```
from gensim.models import Phrases
```

First, let's build bigrams:

```
bigram = Phrases(doc_tokenized_Ngrams, min_count=1,
    delimiter=b' ')
```

Second, we use the output of bigrams to build trigrams:

```
trigram = Phrases(bigram[doc_tokenized_Ngrams],
    min_count=1, delimiter='')
```

Let's print out the results:

```
for sent in doc_tokenized_Ngrams:
    bigrams_ = [b for b in bigram[sent] if b.count(' ') ==
        1]
    trigrams_ = [t for t in trigram[bigram[sent]] if
```

```
t.count(' ') == 2]
print(trigrams_)
```

The outputs look like this. Of the seven sentences, Gensim only identifies the last two sentences as having trigrams:

```
[] []
[]
[]
[]
[]
['come and visit']
['come and visit']
```

Now, we will code Bag-of-N-grams in `scikit-learn`.

## scikit-learn for N-grams

`scikit-learn` simply creates consecutive words for N-grams without checking whether they are common phrases in a document. It is done using the same `CountVectorizer` function we used in BoW. To create bigrams, we specify 2 in the range for  $N$ -gram – `CountVectorizer(ngram_range=[2, 2])`. I have reused the same code we used in the previous *scikit-learn for BoW (CountVectorizer)* subsection:

```
doc_list = [
    "Start spreading the news",
    "You're leaving today (tell him friend)",
    "I want to be a part of it, New York, New York",
    "Your vagabond shoes, they are longing to stray",
    "And steps around the heart of it, New York, New York"
]
from sklearn.feature_extraction.text import CountVectorizer
ngram_vectorizer = CountVectorizer(ngram_range=(2, 2))
ngram_fit = ngram_vectorizer.fit_transform(doc_list)
```

Let's put the bigrams in a list:

```
word_list = ngram_vectorizer.get_feature_names_out()
```

And then we'll put the count of the bigrams in a list:

```
count_list = ngram_fit.toarray().sum(axis=0)
```

Then, we'll print the bigrams and counts in a dictionary for readability:

```
pp.pprint( dict(zip(word_list, count_list)) )
```

The results look like this:

```
{ 'and steps': 1, 'are longing': 1, 'around the': 1, 'be part': 1, 'heart of': 1, 'him friend': 1, 'it new': 2, 'leaving today': 1, 'longing to': 1, 'new york': 4, 'of it': 2, 'part of': 1, 're leaving': 1, 'shoes they': 1, 'spreading the': 1, 'start spreading': 1, 'steps around': 1, 'tell him': 1, 'the heart': 1, 'the news': 1, 'they are': 1, 'to be': 1, 'to stray': 1, 'today tell': 1, 'vagabond shoes': 1, 'want to': 1, 'york new': 2, 'you re': 1, 'your vagabond': 1}
```

Similarly, we can create trigrams:

```
from sklearn.feature_extraction.text import CountVectorizer
ngram_vectorizer = CountVectorizer(ngram_range=(3, 3))
ngram_fit = ngram_vectorizer.fit_transform(doc_list)
```

We'll put the trigrams in a list:

```
word_list = ngram_vectorizer.get_feature_names_out()
```

We'll put the count of the trigrams in a list:

```
count_list = ngram_fit.toarray().sum(axis=0)
pp.pprint( dict(zip(word_list, count_list)) )
```

Now the output will be as follows:

```
'and steps around': 1, 'are longing to': 1, 'around the heart': 1, 'be part of': 1, 'heart of it': 1, 'it new york': 2, 'leaving today tell': 1, 'longing to stray': 1, 'new york new': 2, 'of it new': 2, 'part of it': 1, 're leaving today': 1, 'shoes they are': 1, 'spreading the news': 1, 'start spreading the': 1, 'steps around the': 1, 'tell him friend': 1, 'the heart of': 1, 'they are longing': 1, 'to be part': 1, 'today tell him': 1, 'vagabond shoes they': 1, 'want to be': 1, 'york new york': 2, 'you re leaving': 1, 'your vagabond shoes': 1}
```

Next, we will learn how to code Bag-of-N-grams with NLTK.

## NLTK for N-grams

**Natural Language Toolkit (NLTK)** is an important NLP Python package. It has a suite of text-processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, and wrappers for industrial-strength NLP libraries. It also contains more than 50 corpora and lexical resources.

To generate  $N$ -grams, we will combine all five lists into a long string:

```
flat_list = []
for sublist in doc_tokenized:
    for item in sublist:
        flat_list.append(item)
flat_list
```

This outputs as follows:

```
['start', 'spreading', 'the', 'news', 'you', 're', 'leaving', 'today',
'tell', 'him', 'friend', 'want', 'to', 'be', 'part', 'of', 'it',
'new', 'york', 'new', 'york', 'your', 'vagabond', 'shoes', 'they',
'are', 'longing', 'to', 'stray', 'and', 'steps', 'around', 'the',
'heart', 'of', 'it', 'new', 'york', 'new', 'york']
```

Now, we will create the Bag-of- $N$ -grams by using the following:

```
from nltk.util import bigrams, trigrams
```

The NLTK `bigrams()` function creates bigrams as follows:

```
nltk_bigrams = list(bigrams(flat_list))
nltk_bigrams
[('start', 'spreading'), ('spreading', 'the'), ('the', 'news'),
('news', 'you'), ('you', 're'), ('re', 'leaving'), ('leaving',
'today'), ('today', 'tell'), ('tell', 'him'), ('him', 'friend'),
('friend', 'want'), ('want', 'to'), ('to', 'be'), ('be', 'part'),
('part', 'of'), ('of', 'it'), ('it', 'new'), ('new', 'york'), ('york',
'new'), ('new', 'york'), ...]
```

To create trigrams with NLTK, we'll just use `trigrams()`:

```
nltk_trigrams = list(trigrams(flat_list))
nltk_trigrams
```

Now, the output is as follows:

```
[('start', 'spreading', 'the'), ('spreading', 'the', 'news'), ('the',
'news', 'you'), ('news', 'you', 're'), ('you', 're', 'leaving'),
('re', 'leaving', 'today'), ('leaving', 'today', 'tell'), ('today',
'tell', 'him'), ('tell', 'him', 'friend'), ('him', 'friend', 'want'),
('friend', 'want', 'to'), ...]
```

We have learned about the coding for Bag-of- $N$ -grams with Gensim, `scikit-learn`, and NLTK. Now, we will learn how to code TF-IDF with Gensim and `scikit-learn`.

## Coding – TF-IDF

TF-IDF is a score between 0 and 1. A high TF-IDF value indicates that a term is meaningful and a small value indicates that a term is a common one. Let's start with Gensim to see how it codes for TF-IDF.

### Gensim for TF-IDF

The Gensim module for TF-IDF is `TfidfModel`. TF-IDF is built on BoW. Because we already built BoW In the *Coding – BoW* section, all we need to do now is input the BoW results to the TF-IDF function. I will reprint the *Coding – BoW* section so you do not need to search back and forth. There are five strings in `doc_list`:

```
doc_list = [
    "Start spreading the news",
    "You're leaving today (tell him friend)",
    "I want to be a part of it, New York, New York",
    "Your vagabond shoes, they are longing to stray",
    "And steps around the heart of it, New York, New York"
]
```

The first number in the 2-tuple is the word index, and the second number is the word count:

```
BoW_corpus
```

The first two lists are as follows:

```
[(0, 1), (1, 1), (2, 1), (3, 1)],
[(4, 1), (5, 1), (6, 1), (7, 1), (8, 1), (9, 1), (10, 1)]
```

Next, we will perform the TF-IDF transformation:

```
from gensim.models import TfidfModel
tfidf = TfidfModel(BoW_corpus, smartirs='ntc')
```

There are five lists in `BoW_corpus`. So, when I use `tfidf[BoW_corpus]`, it will produce the TF-IDF values for the five lists. Let's just print out the TF-IDF of the third sentence, which is 2 in the list:

```
tfidf[BoW_corpus][2]
```

It produces the following result:

```
[(11, 0.390273182711427), (12, 0.2221916716729212), (13,
0.4443833433458424), (14, 0.2221916716729212), (15,
0.390273182711427), (16, 0.2221916716729212), (17, 0.390273182711427),
(18, 0.4443833433458424)]
```

The last 2-tuple is  $(18, 0.4443833433458424)$ . It means the TF-IDF value for the 18th word is 0.4443833433458424. From `BoW_corpus`, we know the 18th word is “York” from “*I want to be a part of it, New York, New York.*” It has a relatively higher value than others because it carries more specific information in this document when compared to the entire corpus.

There is one minor point about TF-IDF. Did you notice the `smartirs='ntc'` parameter in `TfidfModel()`? There was a lot of research on the variations of TF-IDF in *Eq. (1)*. These variations will result in slightly different TF-IDF values. Let me explain the variations. The first term in *Eq. (1)*, TF, can be defined as the raw count, or the logarithmic or even double logarithmic values of the raw count. The second term in *Eq. (2)*, IDF, can be defined as IDF or probabilistic IDF. In addition, whether the length of a document should be normalized or not will influence the TF-IDF values. Why are we discussing normalizing the length of a document? It is because a long document usually has the same terms repeating over and over, or because a few documents in a corpus may simply cover more materials than other documents. Hence, it is a good idea to normalize the length of documents.

All these variations have been summarized into a three-letter system called the “SMART system” (that’s why the parameter is `smartirs='ntc'`). It includes “term frequency weighting,” “document frequency weighting,” and “document normalization” and has corresponding notations for them. The coding system is shown here. The default notations are “ntc,” which means the term frequency is “n” for the raw count, the document frequency is “t” for zero-corrected IDF, and the documentation normalization will use a cosine transformation. Let’s see other variants:

- **Term frequency weighting:**
  - b: binary
  - t or n: raw
  - a: augmented
  - l: logarithm
  - d: double logarithm
  - L: log average
- **Document frequency weighting:**
  - x or n: none
  - f: idf
  - t: zero-corrected idf
  - p: probabilistic idf

- **Document normalization:**

- x or n: none
- c: cosine
- u: pivoted unique
- b: pivoted character length

The default “ntc” is proven in past research, and you do not need to worry too much about different variants.

We have learned about the Gensim method for TF-IDF. Next, let’s code TF-IDF with scikit-learn.

## scikit-learn for TF-IDF

The scikit-learn module for TF-IDF is `TfidfVectorizer`. It follows the standard scikit-learn syntax style. We will first declare the object as `tfidf_vectorizer` and then use the `.fit()` function to perform it:

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer()
tfidf_vectorizer.fit(doc_list)
```

scikit-learn outputs the TF-IDF in a matrix. The unique words are the columns and the rows are the documents. Remember the order of the columns is the order of the unique words but not the original sentences:

```
import numpy as np
np.set_printoptions(precision=2)
print(tfidf_vectorizer.transform(doc_list).toarray())
```

Here is the matrix:

```
[[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.
0.42 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. 0.38 0. 0.38 0. 0.38 0. 0. 0. 0. 0. 0. 0.38 0. 0. 0. 0. 0. 0. 0.
0.38 0. 0. 0. 0. 0.38 0. 0. 0. 0. 0. 0.38 0. ],
[0. 0. 0. 0. 0.31 0. 0. 0. 0.25 0. 0. 0.51 0. 0. 0.25 0.31 0. 0. 0. 0. 0. 0.
0. 0. 0. 0.25 0. 0. 0.31 0.51 0. 0. ],
[0. 0.36 0. 0. 0. 0. 0. 0. 0. 0.36 0. 0. 0. 0. 0. 0. 0.36 0. 0. 0. 0. 0.36
0. 0. 0.36 0.29 0. 0.36 0. 0. 0. 0.36],
[0.3 0. 0.3 0. 0. 0.3 0. 0.24 0. 0. 0.48 0. 0.24 0. 0. 0. 0. 0. 0. 0.3 0.
0. 0.24 0. 0. 0. 0. 0.48 0. 0. ]]
```

Most of the elements are zeros. In the first array, there are only four non-zero values. This is because there are four words in the first sentence. Let's get the words as the column names:

```
tfidf_vectorizer.get_feature_names_out()
```

Here is the output:

```
['and', 'are', 'around', 'be', 'friend', 'heart', 'him', 'it',
'leaving', 'longing', 'new', 'news', 'of', 'part', 're', 'shoes',
'spreading', 'start', 'steps', 'stray', 'tell', 'the', 'they', 'to',
'today', 'vagabond', 'want', 'york', 'you', 'your']
```

It means the first column of the matrix is “and.” In the first sentence, there is no “and,” so the TF-IDF in the first array is 0.0.

With this, we have completed the coding for all text classification methods.

## Summary

In this chapter, we learned about the basic forms of text representation, including BoW, Bag-of-N-grams, and TF-IDF methods, to represent raw text. The advantage of BoW is its simplicity. The Bag-of-N-grams method enhances BoW because it captures phrases. TF-IDF can enhance BoW by measuring the importance of a word in a document relative to the entire corpus. Words that are rare in a document will have a high score in the TF-IDF vector. The common disadvantage of BoW, Bag-of-N-grams, and TF-IDF is they create a very sparse matrix. Also, they do not take into consideration the order of words in an article. In this chapter, we also learned how to perform BoW and TF-IDF in Gensim, scikit-learn, and NLTK.

As we become more hands-on with texts, we'll need to deal with words in uppercase or lowercase, or documents with punctuation, numbers, and special characters. We'll also need to distinguish meaningful words from common words and annotate them with grammatical notations. All of these are known as text preprocessing, which we'll discuss in the next chapter.

## Questions

1. What is the advantage of Bag-of-N-words over BoW?
2. What is the advantage of TF-IDF over BoW?
3. What are the real-world applications of the BoW model and TF-IDF model?

## References

1. Bengio, Y., Ducharme, Ré., Vincent, P. & Janvin, C. (2003). *A Neural Probabilistic Language Model*. J. Mach. Learn. Res., 3, 1137–1155.
2. G. Csurka; C. Dance; L.X. Fan; J. Willamowski & C. Bray (2004). *Visual categorization with bags of keypoints*. Proc. of ECCV International Workshop on Statistical Learning in Computer Vision.

# 3

## Text Wrangling and Preprocessing

Almost 80% of NLP is data preprocessing. When we do topic modeling using TF-IDF, LDA, LSA, or similar models, we need to prepare the texts. Without text preprocessing, the quality of the model outcome will suffer, and latent information may be buried in the ocean of texts. The well-known phrase **garbage in, garbage out (GIGO)** refers to this. In this chapter, we will learn the key steps in NLP preprocessing: **tokenization, lowercase conversion, stop word removal, punctuation removal, stemming, and lemmatization**. The first two of these are very basic, so we will spend more time on the rest. We will learn how to code these steps in spaCy, NLTK, and Gensim. Later, we will build a pipeline for NLP preprocessing applicable to any NLP preprocessing in the future.

Specifically, we will cover the following topics:

- Steps in NLP preprocessing
- Coding with spaCy
- Coding with NLTK
- Coding with Gensim
- Building a pipeline with spaCy

By the end of this chapter, you will be able to explain the steps in text preprocessing. You will code the preprocessing procedures with spaCy, NLTK, and Gensim, and be able to build a pipeline with spaCy.

### Technical requirements

You can find code files for this chapter on GitHub at <https://github.com/PacktPublishing/The-Handbook-of-NLP-with-Gensim/tree/main/Chapter03>.

## Key steps in NLP preprocessing

NLP has accumulated much knowledge in preprocessing texts. The key steps in NLP preprocessing are tokenization, lowercase conversion, stop word removal, punctuation removal, stemming, and lemmatization. These steps help to ensure the text quality for modeling and further analyses. Let's learn about them in detail.

### Tokenization

While we see a sentence consisting of individual words, computers see a sentence as an inseparable string. Tokenization is the process of splitting a string into a list of tokens. For example, one line of the song "Theme from New York, New York" that we used in *Chapter 2, Text Representation*, is: "I want to be a part of it, New York, New York." After tokenization, it becomes a list:

```
['I', 'want', 'to', 'be', 'a', 'part', 'of', 'it', ',', 'New', 'York',
', 'New', 'York']
```

### Lowercase conversion

Many computer languages including Python are case-sensitive, so it is always helpful to convert the text to lowercase. Thus, the previous list becomes the following:

```
['i', 'want', 'to', 'be', 'a', 'part', 'of', 'it', ',', 'new', 'york',
', 'new', 'york']
```

### Stop word removal

Human language contains many common words that help to establish sentence structure, clarify relationships between words, and emphasize the tone of a statement. Common words such as "to," "how," "at," "for," "the," "in," "although," "but," and so on play a crucial role in structuring sentences. However, because they appear so often, they do not add much value to NLP. For example, if you search for "What is a wind turbine?" the search engine only looks for the term "wind turbine." Removing stop words will reveal the more meaningful words that actually carry the message in a sentence.

It is important to define what stop words are so we do not unwittingly drop a meaningful word. Is there a stop word dictionary? The **Natural Language Toolkit (NLTK)** stop word list is probably the most recognized and frequently used. It is available in the NLTK library, and we will use it later. Another popular dictionary is the spaCy stop word list, and we will use it later as well.

Stop word lists can change and grow. Relatively recently, some commonly used words sequences or symbols on the internet such as "www," "com," and "HTTP" have become stop words too. In many professional fields such as biomedical or engineering, technical jargon is frequent and may be uninformative. Sarica & Luo (2021) [1] suggest using **technical language processing (TLP)** that focuses on the common words in technical fields beyond the stop words in general texts.

## Punctuation removal

Punctuation such as periods, commas, and parentheses are the marks to help us understand the message clearly. In most applications, these marks become individual marks after tokenization and do not carry much meaning. The common characters include !, “#, \$, %, &, ;, (,), \*, +, /, :, <, =, >, ?, @, [ , \ ], ^, \_, ` , { , | , }, and ~, except apostrophe ('). In some rare applications, punctuation may carry a hidden meaning and we need to retain it. For example, an exclamation mark in an email subject line may carry emotional meaning. We may need to test whether it adds value.

## Stemming

When we see the root of a word, we can guess its meaning. A root word, also called the stem of a word, is the base form of a word. For example, the stem of the words “playing,” “plays,” and “played” is “play.” Different derived forms can be created by adding prefixes, suffixes, or other morphological changes. For example, the word “friendships” is “ship” attached to “friend” and “s” attached to “friendship.”

The technique to extract the root form of words is called stemming. There are three common rules:

- If the word ends in “ed,” remove the “ed”
- If the word ends in “ing,” remove the “ing”
- If the word ends in “ly,” remove the “ly”

With that, let’s see a few other stemming examples:

- The word “significant” becomes “signifi”
- The word “tensely” becomes “tens”
- The word “magnitude” becomes “magnitud”
- The word “improve” becomes “improv”
- The word “populated” becomes “popul”

There is a slight difference between lemmatization and stemming. Stemming converts a word to its root form that may not be a common formal word (such as “populated” becoming “popul”). Lemmatization converts a word to a meaningful base word, where the base word is still an actual word. Let’s understand lemmatization in more detail.

## Lemmatization

As we learned, stemming removes the affixes to return the root of a word. After stemming, the root of a word may not be a regular word. For example, the stems of the words “tensely” and “unequal” are “tens” and “unequ.” In contrast to stemming, lemmatization returns the dictionary meaning of

the word after a word is converted into its root form. Lemmatization in linguistics is the process that turns a word into its *lemma*, which is the canonical form of a word in a dictionary.

It is not necessary to perform stemming and lemmatization together. Often, it is sufficient just to perform one or the other. In many textbooks, the two tasks are combined into one section. Note that spaCy does not perform stemming but lemmatization, while NLTK can perform both.

Let's start with spaCy.

## Coding with spaCy

Since the previously mentioned text preprocessing steps are fundamental to NLP, the NLP community has long sensed the demand for an open source library to benefit more researchers. Thus, spaCy was developed and open sourced. It is designed particularly for production use. Researchers can build applications that process massive volumes of text efficiently. Its NLP pipeline handles all the assigned NLP tasks and then stores the results as attributes to each tokenized word.

Figure 3.1 shows how the `nlp()` pipeline of spaCY works. It takes the raw text, tokenizes the text with its tokenizer, tags each tokenized word with its tagger, and so on. The results are stored as attributes:

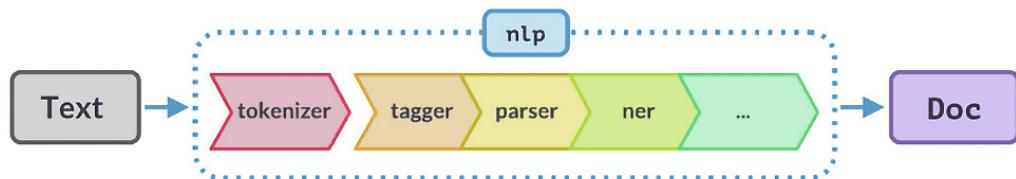


Figure 3.1 – The spaCy pipeline

Let's see what they are:

- **tokenizer:** This tokenizes the text and turns a string of text into an NLP object.
- **tagger and parser:** This assigns **part-of-speech (PoS)** tags and dependency labels. The PoS function is a very strong feature of spaCy. I will describe it later in the chapter.
- **ner:** This is the EntityRecognizer function that detects and labels named entities. In NLP information extraction, we deal with many named entities such as people, locations, organizations, products, and so on. This function is a strong feature of spaCy. We will discuss it in later chapters.
- **lemmatizer:** This performs lemmatization.

Let's apply `nlp()` to a string to see how it works:

```
import spacy
text = "The US economy has been looking solid lately that Federal
Reserve officials will probably double their projection for growth
next year."
nlp = spacy.load("en_core_web_sm")
doc = nlp(text)
```

These few code lines have performed all the NLP tasks. They have tokenized words and attached the attributes to each word. Let's first examine the results of stop-word removal. The `is_stop` stop-word attribute is a Boolean value that states whether or not the word is a stop word. Let's print out the words in the text used previously and their stop-word attributes:

```
for token in doc:
    print(token.text,token.is_stop)
```

The output is as follows:

```
The --> True
US --> True
economy --> False
has --> True
been --> True
looking --> False
solid --> False
lately --> False
that --> True
Federal --> False
Reserve --> False
officials --> False
will --> True
probably --> False
double --> False
their --> True
projection --> False
for --> True
growth --> False
next --> True
year --> False
. --> False
```

spaCy does not automatically remove stop words but lets users have full control of stop-word removal. Let's learn to print out the original sentence, and the sentence without stop words:

```
import spacy
#loading the english language small model of spacy
en = spacy.load('en_core_web_sm')
stopwords = en.Defaults.stop_words
```

spaCy lets you add your own stop words. Suppose we consider "year" a stop word in this context. We will add it to the stop-word list by doing the following:

```
nlp = spacy.load("en_core_web_sm")
nlpDefaults.stop_words.add("year")
```

Let's do an experiment. The preceding code checks whether a lowercase word is in the stop-word list or not. If not, the word will be added to the empty list, named `lst`:

```
lst=[]
for token in text.split():
    if token.lower() not in stopwords: lst.append(token)
print(' '.join(lst))
```

The output is as follows:

```
economy looking solid lately Federal Reserve officials probably double
projection growth.
```

Notice that stop words in the original text "the," "us," "has," "been," "that," "will," "their," "for," "next," and our newly added stop word "year" have all been removed. You may have noticed that "U.S." is spelled as "US" in the text, and becomes the lowercase "us" and is considered a stop word. There are nuances such as this in NLP and you may need to make special corrections to the input text.

## spaCy for lemmatization

The `lemma_` attribute stores the lemmatized form of a word. Let's print out each word and the lemmatized word:

```
for token in doc:
    print(token.text, "-->", token.lemma_)
```

Here, I will just show the lemmatized words that are different from the original words:

```
has --> have
been --> be
looking --> look
```

## spaCy for PoS

PoS labels the correct meaning of a word in a sentence according to its context. It is a system where a word is assigned a syntactic function such as noun, pronoun, adjective, verb, and so on. PoS tagging is a significant advance in NLP development. It is the fruit of the linguistic and statistical research of the past two decades. The English language has more than 30 PoS tags based on linguistic research. The following are some examples of these tags, with their descriptions:

- **NNP:** Proper noun singular
- **NNPS:** Proper noun, plural
- **VB:** Verb, base form
- **VBD:** Verb, past tense
- **VBG:** Verb, gerund/present participle
- **VBN:** Verb, past

For example, the sentence “Frank is 54 years old” will be labeled as the following:

- (Frank: NNP): “Proper noun”
- (is: VBZ): “Singular verb, third person singular present”
- (54: CD): “Cardinal Number”
- (years: NNS): “Noun”
- (old: JJ): “Adjective”

Let's see what PoS tags spaCy applies to our text:

```
for token in doc:
    print(token.text, "---->", token.pos_)
```

Here, I have just printed out a few outputs:

```
The ----> DET
US ----> PROPN
economy ----> NOUN
has ----> AUX
looking ----> VERB
solid ----> ADJ
lately ----> ADV
that ----> SCONJ
Federal ----> PROPN
for ----> ADP
. ----> PUNCT
```

I trust you like the pipeline idea that processes all NLP tasks together. Now we have learned how spaCy operates, let's turn to another popular library – NLTK.

## Coding with NLTK

NLTK is the oldest and most widely used library in NLP. It has many easy-to-use interfaces and stores over 50 corpora and lexical resources such as **WordNet**. WordNet is a large database for the semantic relations between nouns, verbs, adjectives, and adverbs. It can be seen as a digital dictionary and thesaurus. See *Transfer Learning for Image Classification – (2) Trained Image Models* [5] for more detail. NLTK has a suite of text-processing libraries. It performs NLP tasks including tokenization, tagging, parsing, and stemming. It also includes libraries that perform semantic reasoning, and wrappers for industrial-grade NLP tasks. It has been used by researchers, linguists, engineers, educators, researchers, and industry professionals.

**Google Colab** already has the popular NLTK functions installed. You just need to run the following syntax in Google Colab:

```
import nltk  
nltk.download("popular")
```

Now, I am going to use NLTK to perform tokenization, stop-word removal, and lemmatization, step by step.

### NLTK for tokenization

To tokenize raw text with NLTK, you just need to use `word_tokenize()`:

```
import nltk  
nltk.download('wordnet')  
from nltk.stem import WordNetLemmatizer  
from nltk.tokenize import word_tokenize  
text_tokenized = word_tokenize(text)  
print(text_tokenized)
```

Now, we will see the output:

```
['The', 'US', 'economy', 'has', 'been', 'looking', 'solid', 'lately',  
'that', 'Federal', 'Reserve', 'officials', 'will', 'probably',  
'double', 'their', 'projection', 'for', 'growth', 'next', 'year', '.']
```

## NLTK for stop-word removal

NLTK has a list of stop words. The following code loads the list and filters out any tokenized words belonging to the stop-word list:

```
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))
filtered_sentence = [w for w in text_tokenized if not
    w.lower() in stop_words]
#with no lowercase conversion
filtered_sentence = []
for w in text_tokenized:
    if w not in stop_words:
        filtered_sentence.append(w)
print(filtered_sentence)
```

Here is the output without the stop words:

```
['The', 'US', 'economy', 'looking', 'solid', 'lately', 'Federal',
'Reserve', 'officials', 'probably', 'double', 'projection', 'growth',
'next', 'year', '.']
```

Or, let's just print out the stop words that have been removed:

```
removed_words = []
for w in text_tokenized:
    if w in stop_words:
        removed_words.append(w)
print(removed_words)
```

The output is as follows:

```
['has', 'been', 'a', 'which', 'the', 'is', 'just', 'the', 'that',
'what', 'to', 'have', 'been']
```

## NLTK for lemmatization

The `WordNetLemmatizer()` function performs lemmatization based on WordNet:

```
removed_words = []
for w in text_tokenized:
    if w in stop_words:
        removed_words.append(w)
print(removed_words)
```

NLTK lemmatization only identifies one word whose lemmatized form is different from its original form:

```
has --> ha
```

The lemmatization by spaCy and NLTK can be different. You are advised to inspect them carefully before adopting one of them.

Now we have learned about both spaCy and NLTK, let's learn Gensim's way of performing these NLP tasks.

## Coding with Gensim

Gensim already builds in the common preprocessing tasks including tokenization, stemming, and stop word removal in its `preprocess_string()` function. I am going to show you the preprocessing function. I'll also separately demonstrate Gensim's stop word removal and stemming tasks.

### Gensim for preprocessing

Gensim's `preprocess_string()` class is an effective and powerful class that can perform all text preprocessing tasks such as stop-word removal, tagging, punctuation removal, and stemming. Let's see the code:

```
from gensim.parsing.preprocessing import remove_stopwords, preprocess_
string
remove_stopwords(text)
preprocess_string(text)
```

The output is as follows:

```
['econom', 'look', 'solid', 'late', 'feder', 'reserv', 'offici',
'probabl', 'doubl', 'project', 'growth', 'year']
```

The `preprocess_string()` function performs the following NLP functions:

- `strip_tags()`: Removes HTML tags such as `<p>` in a sentence
- `strip_punctuation()`: Removes punctuation
- `strip_multiple_whitespaces()`: Removes multiple white spaces
- `strip_numeric()`: Removes numeric values
- `remove_stopwords()`: Removes stop words specified in the list
- `strip_short()`: Removes words with less than three characters
- `stem_text()`: Performs stemming using **PorterStemmer**

## Gensim for stop-word removal

The `remove_stopwords()` function performs stop-word removal. Remember, the stop-word removal step is already included in the `preprocess_string()` function. Let's see it in action:

```
stopword_removed = remove_stopwords(text)  
stopword_removed
```

We will see the following results:

```
The US economy looking solid lately Federal Reserve officials probably  
double projection growth year.
```

## Gensim for stemming

The `stem_text()` function derives the stem of a word. As with stop-word removal, the stemming step is already included in the `preprocess_string()` function:

```
from gensim.parsing.preprocessing import stem_text  
stem_text(text)
```

The output is as follows:

```
the us economi ha been look solid late that feder reserv offici will  
probabl doubl their project for growth next year.
```

spaCy was developed to perform all these preprocessing steps. Let's learn how spaCy can perform all of these in a pipeline.

## Building a pipeline with spaCy

As mentioned earlier, spaCy can perform all the NLP functions such as PoS, name entity recognition, lemmatization, and so on. *Table 3.1* shows these functions:

String name	Description
Tagger	Assigns PoS tags
Parser	Assigns dependency labels
Ner	Assigns named entities
entity_linker	Assigns knowledge base IDs to named entities; should be added after the entity recognizer
entity_ruler	Assigns named entities based on pattern rules and dictionaries

String name	Description
Textcat	Assigns text categories; exactly one category is predicted per document
textcat_multilabel	Assigns text categories in a multi-label setting: zero, one, or more labels per document
Lemmatizer	Assigns base forms to words using rules and lookups
trainable_lemmatizer	Assigns base forms to words
Morphologizer	Assigns morphological features and coarse-grained PoS tags
attribute_ruler	Assigns token attribute mappings and rule-based exceptions
Senter	Assigns sentence boundaries
Sentencizer	Adds rule-based sentence segmentation without the dependency parse
tok2vec	Assigns token-to-vector embeddings
Transformer	Assigns the tokens and outputs of a transformer model

Table 3.1 – Functionalities of spaCy

Now, let's learn to use spaCy to build our NLP preprocessing pipeline on our data. We will use the AG news data as described in *Chapter 1, Introduction to NLP*:

```
import pandas as pd
pd.set_option('display.max_colwidth', -1)
train = pd.read_csv("/content/gdrive/My Drive/data/ag_news_train.csv")
train.head()
```

The first few records are shown in *Table 3.2*:

Class index	Title	Description	Tokenized
3	Wall St. Bears Claw Back Into the Black (Reuters)	Reuters - Short-sellers, Wall Street's dwindling\band of ultra-cynics, are seeing green again.	[reuters, -, short, -, sellers, ,, wall, street, dwindling\band, ultra, -, cynics, ,, seeing, green, .]

Class index	Title	Description	Tokenized
3	Carlyle Looks Toward Commercial Aerospace (Reuters)	Reuters - Private investment firm Carlyle Group,\which has a reputation for making well-timed and occasionally\ controversial plays in the defense industry, has quietly placed\its bets on another part of the market.	[reuters, -, private, investment, firm, carlyle, group,\which, reputation, making, -, timed, occasionally\ controversial, plays, defense, industry, „ quietly, placed\its, bets, market, .]
3	Oil and Economy Cloud Stocks' Outlook (Reuters)	Reuters - Soaring crude prices plus worries\about the economy and the outlook for earnings are expected to\hang over the stock market next week during the depth of the\summer doldrums.	[reuters, -, soaring, crude, prices, plus, worries\ about, economy, outlook, earnings, expected, to\ hang, stock, market, week, depth, the\summer, doldrums, .]
3	Iraq Halts Oil Exports from Main Southern Pipeline (Reuters)	Reuters - Authorities have halted oil export\flows from the main pipeline in southern Iraq after\ intelligence showed a rebel militia could strike\ infrastructure, an oil official said on Saturday.	[reuters, -, authorities, halted, oil, export\flows, main, pipeline, southern, iraq, after\intelligence, showed, rebel, militia, strike\infrastructure, „ oil, official, said, saturday, .]
3	Oil prices soar to all-time record, posing new menace to US economy (AFP)	AFP - Tearaway world oil prices, toppling records and straining wallets, present a new economic menace barely three months before the US presidential elections.	[afp, -, tearaway, world, oil, prices, „ toppling, records, straining, wallets, „ present, new, economic, menace, barely, months, presidential, elections, .]

Table 3.2 – First few samples in the data

The NLP pipeline of spaCy can let users turn off functions. In order to speed up the process, we will turn off 'tok2vec', 'tagger', 'parser', 'attribute\_ruler', and 'transformer'. We put them in a list called `to_be_disabled`:

```
from scipy.special import k0
to_be_disabled = ['tok2vec', 'tagger', 'parser',
                  'attribute_ruler', 'transformer']
```

Our `train['Description']` raw text is a list of strings. We will feed the list to the `nlp()` function. The `for` loop will iterate through the strings of the list, tokenize each string, and tag each word with attributes. We will check whether a word is a stop word or not by using the `.is_stop` attribute. If it is a stop word, we will not include it. We collect the tokenized words without the stop words in the `text_tokenized` list:

```
nlp = spacy.load("en_core_web_sm")
stopwords = en.Defaults.stop_words
text_tokenized = []
for doc in nlp.pipe(train['Description']), disable =
    to_be_disabled):
    k = [token.lemma_ for token in doc if not
          token.is_stop]
    text_tokenized.append(k)
```

Up to this point, the preprocessing is complete! We can save the processed text as a new Tokenized column for our future studies. Let's print out one example of the raw text and its processed text:

```
train['Tokenized'] = text_tokenized
train.to_csv("/content/gdrive/My Drive/data/ag_news_train_samples.csv")
```

Let's print out the first text list and its tokenized form:

```
train[['Description', 'Tokenized']].head(1)
```

The results are as follows:

```
"British health authorities on Tuesday suspended the manufacturing
license of flu vaccine maker Chiron Corp., a major supplier to the
United States."
'british', 'health', 'authorities', 'tuesday', 'suspended',
'manufacturing', 'license', 'flu', 'vaccine', 'maker', 'chiron',
'corp.', ',', 'major', 'supplier', 'united', 'states', '.'
```

The preceding results are amazing. spaCy has tokenized all the words and converted them all to lowercase. The stop words such as "on" and "the" have been removed. Nevertheless, the period after an abbreviated word such as "corp." is kept. spaCy knows the period should be part of the abbreviated word.

## Summary

Text preprocessing is a critical step in NLP modeling. With proper text preprocessing, the quality of the NLP model's outcome will be satisfactory. This is a fundamental step that experienced NLP data scientists pay attention to.

This chapter presents the three popular Python libraries for text preprocessing: spaCy, Gensim, and NLTK. In each library, we demonstrated how to perform tokenization, stop-word removal, punctuation removal, stemming, and lemmatization. We also saw the strengths of each library and learned how to use them.

In the next chapter, we will start to learn about the concept of latent semantics and **latent semantic analysis (LSA)**.

## Questions

1. What is tokenization?
2. What is stemming?
3. What is the difference between stemming and lemmatization?
4. How does spaCy remove stop words?
5. What is the **part-of-speech (PoS)** system?
6. What does Gensim's `preprocessing_string()` do?

## References

1. Sarica, Serhad, and Jianxi Luo. 2021. "Stopwords in Technical Language Processing." *PLOS ONE* 16 (8): e0254937. <https://doi.org/10.1371/journal.pone.0254937>.
2. Qiao, Yifan, Chenyan Xiong, Zhenghao Liu, and Zhiyuan Liu. 2019. "Understanding the Behaviors of BERT in Ranking," April. <https://doi.org/10.48550/arxiv.1904.07531>.
3. Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." *ArXiv:1810.04805 [Cs]*, May. <https://arxiv.org/abs/1810.04805v2>.
4. Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. "Attention Is All You Need." ArXiv.org. December 5, 2017. <https://doi.org/10.48550/arXiv.1706.03762>.
5. Kuo, Chris. 2022. *Transfer Learning for Image Classification*.



# Part 2:

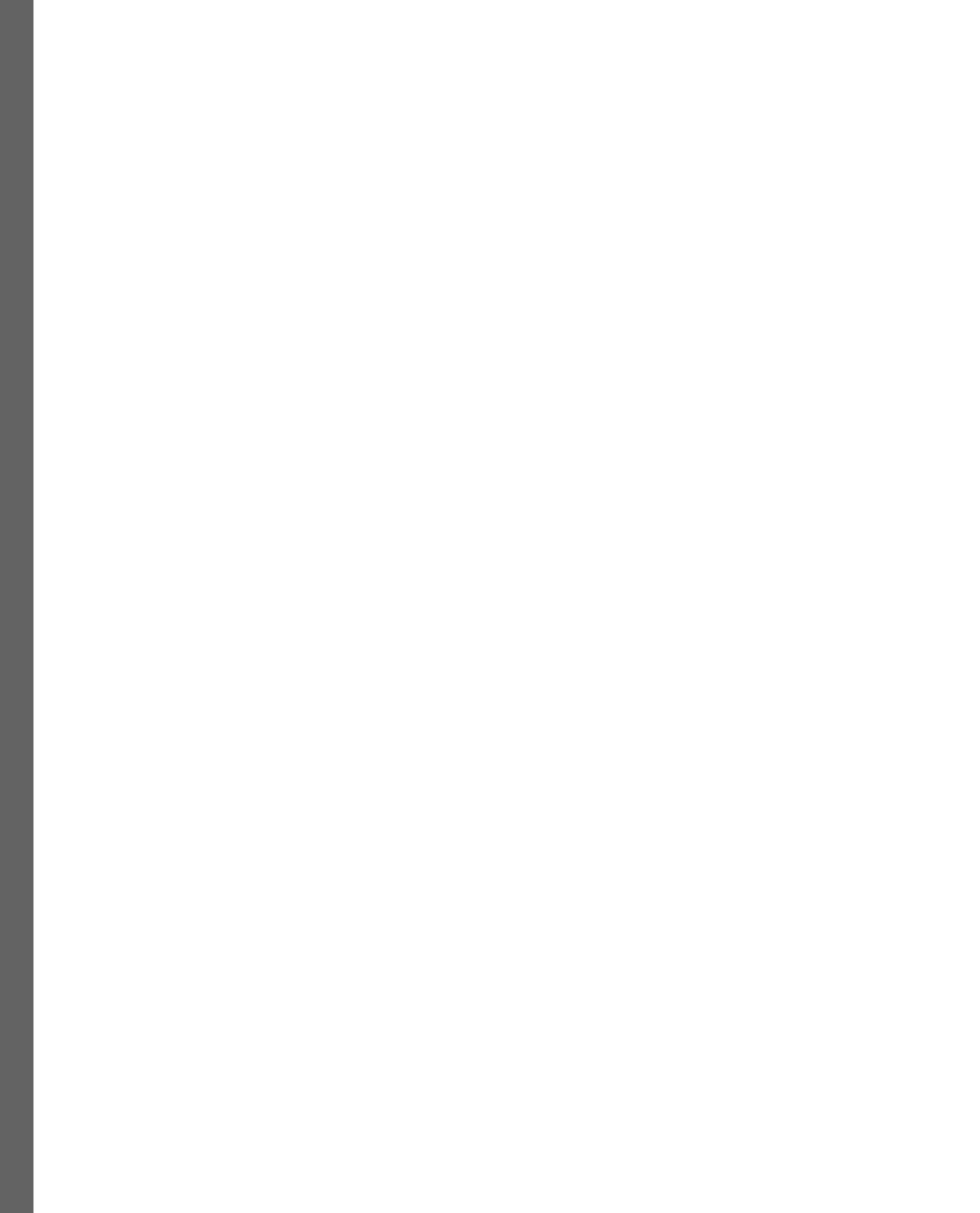
## Latent Semantic Analysis/ Latent Semantic Indexing

In this part, you will understand how **Latent Semantic Analysis (LSA)** was developed from **Sigular Value Decomposition (SVD)** and Truncated SVD. We will use Scikit-learn to show you the development from Truncated SVD to LSA.

This book follows the conventional name LSA/LSI because LSA is also called **Latent Semantic Indexing (LSI)**. In this part, we will introduce one of the core measures for vector similarity – cosine similarity. We will then use Gensim to build an LSA/LSI model. You will learn how to determine the optimal number of topics, set up your model as an informational retrieval tool, and use cosine similarity to search for the most relevant documents according to a set of keywords.

This part contains the following chapters:

- *Chapter 4, Latent Semantic Analysis with scikit-learn*
- *Chapter 5, Cosine Similarity*
- *Chapter 6, Latent Semantic Indexing with Gensim*



# 4

## Latent Semantic Analysis with scikit-learn

Searching for keywords has become part of our information lives. We use keyword searches to find articles. When we type in words, we do not mean to retrieve articles with the exact words but the concepts. For example, when we search for “Tesla” and “electric car,” we want to get articles about the company Tesla and its electric cars, rather than historical information about the scientist Nicola Tesla. This is called a “semantic search,” which means interpreting a word for its intent and contextual meaning. We use keywords as cues to get the concept and the documents that are associated with that concept, rather than using the keywords at their face values.

**Latent Semantic Analysis (LSA)** is a milestone solution that was developed in the 1990s. When words are entered for a keyword search, LSA finds the underlying topics that the words in documents associate with and retrieves those documents. LSA is also called **Latent Semantic Indexing (LSI)**, as we discussed in *Chapter 1, Introduction to NLP*. It is called so because, in 1988, an LSA information retrieval system named *Latent Semantic Indexing* was patented. Gensim and other reports call LSA LSI so as not to confuse LSA with **LDA (Latent Dirichlet Analysis)**. For the same reason, I will adopt the same naming convention in this book.

LSI uses **Singular Value Decomposition (SVD)**. SVD is a popular technique in data science, image operations, physics, and engineering for dimensionality reduction. For these important reasons, this chapter will explain SVD, how it works, and how it is applied to LSI. We will start gently with a basic matrix operation, including eigenvectors and eigenvalues, and a transformation matrix.

LSI was programmed in Gensim with just a few code lines for production. However, I don’t want you to miss the opportunity to understand the SVD algorithm for LSI. Therefore, I will cover LSI in two chapters. I will use `scikit-learn` in this chapter to explain the SVD algorithm for LSI and dedicate the next chapter to LSI modeling with Gensim. This organization will help you to understand the algorithm itself and how to develop production code with Gensim.

The following topics are covered in this chapter:

- Understanding basic matrix operations
- Understanding a transformation matrix
- Understanding eigenvectors and eigenvalues
- An introduction to SVD
- Coding `TruncatedSVD` with `scikit-learn`
- Using `TruncatedSVD` for LSI with real data

With the completion of this chapter, you will learn about or refresh your knowledge of matrix operations, and you'll then be able to explain LSI using SVD and the use of `TruncatedSVD` for LSI.

## Technical requirements

We will use `numpy` and `scikit-learn` in this chapter. The main module for SVD is `TruncatedSVD`:

```
import numpy as np
from sklearn.decomposition import TruncatedSVD
```

You can find the code files for this chapter on GitHub at <https://github.com/PacktPublishing/The-Handbook-of-NLP-with-Gensim/tree/main/Chapter04>.

## Understanding matrix operations

I will start with basic matrix definitions, including an orthogonal matrix and determinant. They will help you to understand a transformation matrix and eigenvalues and eigenvectors.

### An orthogonal matrix

Let me start with a few matrix definitions. A vector is called “normal” if it has a length of one. Two vectors are “orthogonal” if they are at right angles to each other. A matrix is called an “orthonormal matrix” if it is a real square matrix whose columns and rows are orthonormal vectors. *Figure 4.1* gives some examples:

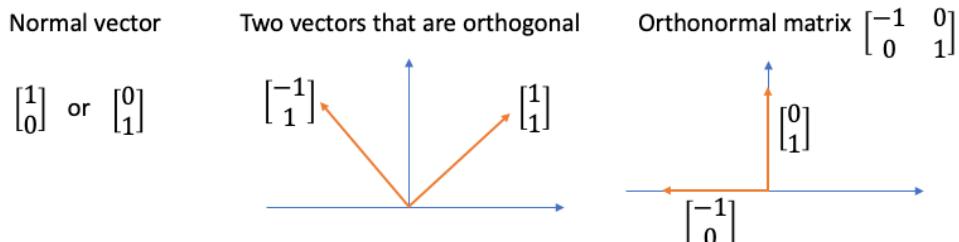


Figure 4.1 – An orthogonal matrix

Why are we interested in an orthogonal matrix? The product of two orthogonal matrices will also be an orthogonal matrix. The transpose of the orthogonal matrix will also be an orthogonal matrix. These properties will be used to derive SVD.

## The determinant of a matrix

The determinant of a matrix is the scalar value of a square matrix. It helps us to derive the inverse of a matrix. For a  $2 \times 2$  matrix,  $\mathbb{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ , its determinant is  $ad-bc$ , as shown here:

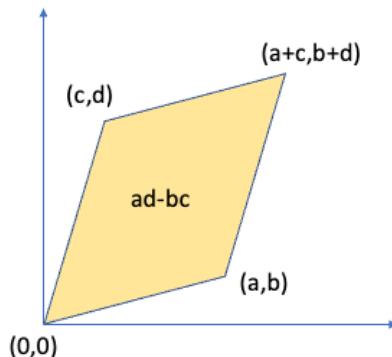


Figure 4.2 – The determinant of a matrix

A matrix is *singular* if its determinant is zero. A  $2 \times 2$  matrix is singular if its determinant  $ad-bc = 0$ . Singular matrices do not have an inverse. You cannot convert the inverse of a singular matrix back to the original matrix. Invertible matrices are crucial to solving systems of linear equations and many other mathematical and engineering applications. Further, they are needed in calculating eigenvalues and eigenvectors, which we will learn about shortly.

Now that we know about the basic properties of matrix operations, let's understand what transformation matrices are.

## Understanding a transformation matrix

A transformation matrix is a powerful mathematical tool. It can perform various geometric transformations, such as rotation, scaling, and shearing, on objects or points in space. By multiplying a vector or set of coordinates by a transformation matrix, you can achieve complex transformations with ease and precision. This simplifies tasks such as rendering 3D graphics, image processing, and robotics.

Figure 4.3 shows four common 2D transformation matrices:

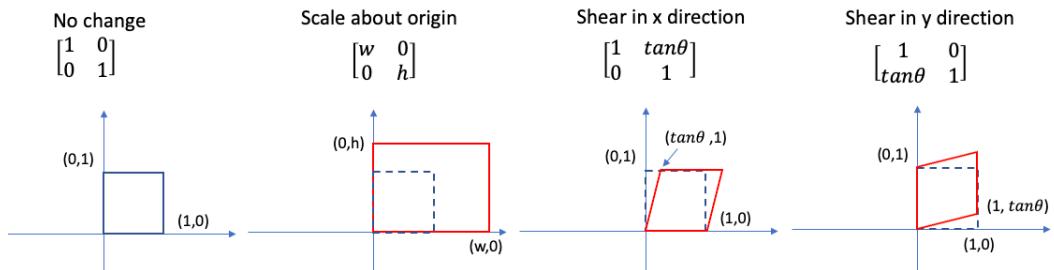


Figure 4.3 – Transformation matrices

When multiplying by a matrix, these transformation matrices will “distort” or “transform” that matrix. Let’s first learn about their mathematical properties, and then I will explain why we want to do that. If the transformation matrix is  $\begin{bmatrix} w & 0 \\ 0 & h \end{bmatrix}$ , it will scale the original matrix up or down by  $w$  and  $h$ . Let’s review each in detail:

- **The first one is “no change to the original matrix”** when the transformation matrix is  $H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ . It says any  $2 \times 2$  matrix or  $2 \times 1$  vector multiplied by the transformation matrix  $H$  will result in no change.
- **The second one is “scale the original matrix about origin.”** The transformation matrix is  $H = \begin{bmatrix} w & 0 \\ 0 & h \end{bmatrix}$ . Any  $2 \times 2$  matrix or  $2 \times 1$  vector multiplied by this transformation matrix will become a scaled-up or scaled-down square around the origin.
- **The third one is “shearing the original matrix in the x direction.”** The result becomes a sheared rectangle about the origin.
- **The fourth one is “shearing the original matrix in the y direction.”** The result becomes a sheared rectangle about the origin.

There are many examples of the use of a transformation matrix in our daily lives. Let me show you a few.

## A transformation matrix in daily life examples

One significant application of linear algebra is image transformation, which is applicable in our daily lives. Figure 4.4 shows three examples that apply matrix transformation.



Figure 4.4 – Image transformation

On the left in *Figure 4.4* is a regular mirror that reflects the original size. The middle of *Figure 4.4* shows a makeup mirror that enlarges your face. On the right of *Figure 4.4* is a convex safety mirror or security mirror that can show a wide angle of view. Consider the original image matrix. The regular mirror does not transform the image. The makeup mirror distorts the image by applying a transformation matrix, and the security mirror distorts the image by applying another kind of transformation matrix. Let me explain further, in *Figure 4.5*. It shows a black-and-white digital image of a dog. A digital image is a 2D matrix. This image is transformed in the mirror. This mirror is the transformation matrix. As shown in *Figure 4.5*, the mirror can transform the original image matrix to scale up or down, or shear around the origin, making it into a new image matrix:

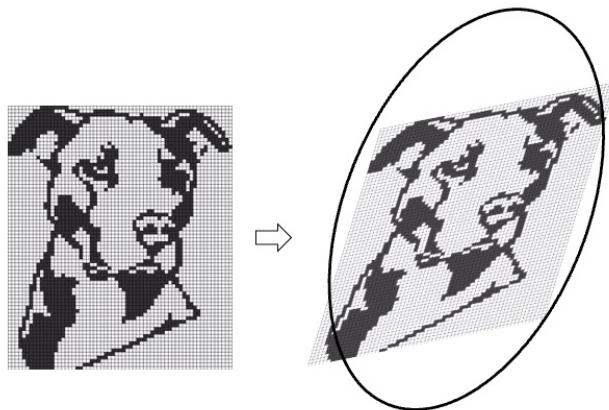


Figure 4.5 – A linear transformation of an image

These matrix operations will help us to understand eigenvectors and eigenvalues, which we will cover next.

## Understanding eigenvectors and eigenvalues

“Eigen” is the German word for “own” and can also mean “characteristic” or “proper.” An eigenvector is like an arrow in space that doesn’t change its direction when you apply a transformation to it, such as stretching or squishing. Imagine you have a rubber band (the eigenvector) and you stretch it or squish it using a magic machine (the transformation). If the rubber band still points in the same direction after the magic, it’s an eigenvector. These special arrows are really helpful in math and science to understand how things change and stay the same in different situations. Refer to *Figure 4.6* for more information on this particular property, and then I will introduce the mathematical properties. The left image (A) in *Figure 4.6* is a 2D image. After we apply a transformation to this image, it becomes the right image (B). Note that the vertical axis (yellow vector) does not change but the direction of the horizontal axis (white vector) does change. Because the vertical yellow vector does not change after the transformation, it is an eigenvector. Not all vectors will stay the same after transformation. The horizontal white vector is not an eigenvector. Further, the vertical yellow vector in the example could become longer or shorter. The size change is determined by its eigenvalue. In *Figure 4.6*, the size of the vertical vector does not change, so its eigenvalue is 1.0. In short, if a vector does not change its direction after applying a transformation, the vector is an eigenvector. However, the size of the vector can scale up or down. This scalar change is the eigenvalue.



Figure 4.6 – Eigenvectors and eigenvalues

The mathematical definition for eigenvector is this – given a transformation matrix  $H$ , a non-zero vector  $X$  is the eigenvector if:

$$HX = \lambda X \quad \text{Eq. (1)}$$

where the scalar  $\lambda$  is the eigenvalue of  $H$ . The direction of  $HX$  in *Eq. (1)* is the same as that of  $X$ . The eigenvalue  $\lambda$  simply becomes the amount of “stretch” or “shrink” to vector  $X$ . Most vectors will not satisfy *Eq. (1)*. Only eigenvectors can satisfy *Eq. (1)*.

Let's practice this concept with the transformation matrix in *Figure 4.3*. Let's examine whether  $X = [0,1]$  is an eigenvector for the transformation matrix  $H$ , "shear in  $y$  direction," in *Figure 4.3*. Assuming that  $\tan\theta = 1/2$  ( $\theta = 30^\circ$ ), matrix  $H$  becomes:

$$H = \begin{bmatrix} 1 & 0 \\ -1/2 & 1 \end{bmatrix}$$

When we multiply  $H$  and  $X$ , we get the following:

$$HX = \begin{bmatrix} 1 & 0 \\ -1/2 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 + 0 \\ 0 + 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 1 \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

We get  $HX = X = [0,1]$ . The direction of  $X$  and  $HX$  is the same. Hence,  $X$  is an eigenvector for the transition matrix  $H$ . Also, the scale of  $HX$  is 1.0 times  $X$ , so the eigenvalue is 1.

I have mentioned the determinant of a matrix before. How is it related to the eigenvalue? Assume  $X$  is an  $n \times n$  matrix. The matrix  $X$  has  $n$  eigenvalues. Thus, the sum of the  $n$  eigenvalues is the same as the sum of the diagonal elements of  $X$  (which is called the *trace* of  $X$ ). Finally, the product of the  $n$  eigenvalues of  $X$  is the same as the determinant of  $X$ .

Why are we spending time learning about the concept of eigenvectors and eigenvalues? It's because the concept is an important property in deriving SVD. Let's see how it works.

## An introduction to SVD

SVD utilizes the properties of eigenvectors and eigenvalues. SVD is a matrix decomposition method that reduces a large, usually sparse, matrix into three sub-matrices. We will assume the following:

- $\lambda_1, \lambda_2, \dots, \lambda_n$  is the eigenvalues of a matrix  $A$
- $x_1, x_2, \dots, x_n$  is a set of corresponding eigenvectors in vector  $V$
- $\Sigma$  denotes the  $n \times n$  diagonal matrix with the  $\lambda_j$  on the diagonal
- $X$  denotes the  $n \times n$  matrix whose  $j^{\text{th}}$  column is  $x_j$

Then, we can rewrite *Eq. (1)* as follows:

$$\mathbb{A}V = U\Sigma \quad \text{Eq. (2)}$$

*Eq. (2)* is the matrix form of *Eq. (1)*. In *Eq. (2)*, it is necessary to put  $\Sigma$  as the second term on the right-hand side. This will make sure each column of  $X$  is multiplied by its corresponding eigenvalue. Let's use a simple example to understand *Eq. (2)*:

$$\mathbb{A} = \begin{bmatrix} 4 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

This matrix has the eigenvalues  $\lambda_1 = 4$  for  $X = [1,0,0]$ ,  $\lambda_2 = 3$  for  $X = [0,1,0]$ , and  $\lambda_3 = 2$  for  $X = [0,0,1]$ . If the determinant of  $V$  is non-zero, then the inverse matrix  $V^{-1}$  exists, and we get the following:

$$\mathbb{A} = U\Sigma V^{-1} \quad Eq. (3)$$

If the values in  $V V^{-1}$  are real numbers (not complex numbers), then  $V^T$  is the transposed matrix. The inverse of an orthogonal  $m \times n$  matrix  $V$  with  $m \geq n$  will always exist and is equal to the transpose of  $V$ :

$$V^{-1} = V^T$$

Remember that both matrices  $U$  and  $V$  satisfy:

$$U^T U = I$$

and

$$V^T V = I$$

Now, *Eq. (3)* becomes the following:

$$\mathbb{A} = U\Sigma V^T \quad Eq. (4)$$

The values along the diagonal of  $\Sigma$  are called *singular values*. They are the square roots of the eigenvalues of  $A^T A$  and, thus, completely determined by  $A$ . The previous decomposition of  $A$  is SVD, which is not unique. If we choose the decomposition that the singular values in  $\Sigma$  are in descending order, then it is uniquely determined by  $A$ . *Figure 4.7* shows how SVD works:

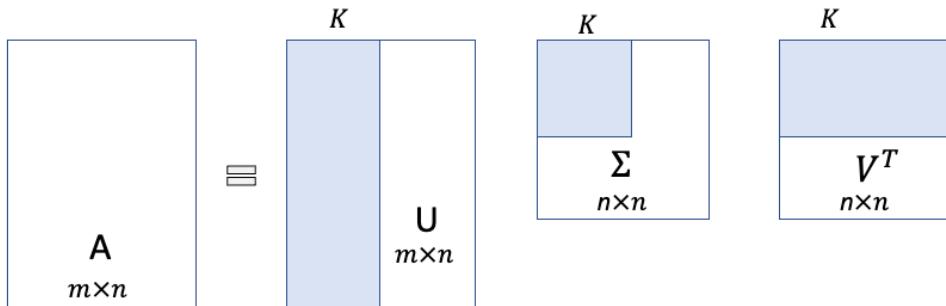


Figure 4.7 – Decomposition of a matrix into three matrices

The figure says the  $m \times n$  matrix  $A$  can be decomposed into three matrices – the  $m \times n$  matrix  $U$ , the  $n \times n$  matrix  $\Sigma$ , and the  $n \times n$  matrix  $V^T$ . The values in the diagonal matrix are called **singular values**. They are always real numbers and are arranged in descending order. From here, let's learn about an important advantage of SVD – truncated SVD.

## Truncated SVD

Since the singular values in  $\Sigma$  are ranked in descending order, the last few singular values are zeros or close to zero. The  $n \times n$  diagonal matrix  $\Sigma$  can be reduced to  $n \times k$ , where  $k < n$ . This property is important because matrix  $A$  can be decomposed into three matrices, and the three matrices can be reduced to smaller matrices. The blue shaded area in matrix  $\Sigma$  is of rank  $k \times k$ , a much smaller rank than  $n \times n$ . The rank for  $U$  becomes  $m \times k$ , and the rank for  $V^T$  becomes  $k \times n$ . The white areas in the three matrices are not needed to reconstruct  $A$ . Since the values in  $\Sigma$  are ordered in descending order and the bottom values are very small, they can be truncated without losing many values, aka the variance, of the original matrix on the left. That's why we name it truncated SVD. The rank  $k$  determines the truncation. We can find the appropriate rank  $k$  that makes the decomposed matrices small and just lose some variances of the original matrix.

How is truncated SVD helpful in LSI? Let's find out.

## Truncated SVD for LSI

In *Chapter 2, Text Representation*, we introduced the **Bag-of-Word (BoW)** matrix and the **Term Frequency-Inverse Document Frequency (TF-IDF)** matrix. They represent documents and words in a document-word matrix such that the rows are the documents and the columns are the words. The document-word matrix is matrix  $A$  in SVD. When applying SVD, matrix  $A$  can be decomposed into three smaller matrices, as shown in *Figure 4.8*.

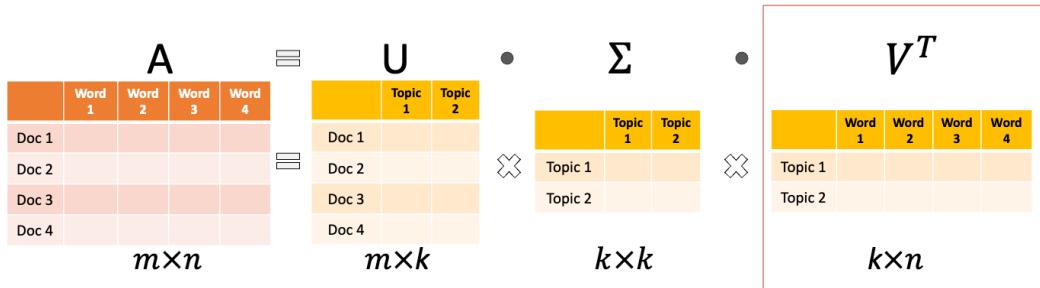


Figure 4.8 – Truncated SVD for documents, words, and topics

Matrix  $\Sigma$  is a  $k \times k$  diagonal matrix with  $V^T k$  latent topics. The first matrix  $U$  is a document-topic matrix. It represents documents with latent topics. The third matrix  $V^T$  is a topic-word matrix. It associates each topic with the words. In summary, SVD can decompose the original sparse document-word matrix into three matrices. The first matrix is a document-topic matrix, the second matrix is a square topic-topic matrix, and the third matrix is a topic-word matrix.

Knowing how SVD is applied for LSI, I will use `truncatedSVD()` of `scikit-learn` to demonstrate the coding.

## Coding truncatedSVD with scikit-learn

Using `scikit-learn` can help us to understand  $m, A, U, \Sigma, V^T$ . The main class is `TruncatedSVD()`. Let's assume matrix  $A$  is a  $5 \times 10$  matrix. In LSI, it means there are 5 documents and 10 unique words. Let's fill in random integer values between 1 and 50 (low = 1 and high = 50):

```
import numpy as np
A = np.random.randint(low=1, high=50, size = (5,10))
print(A)
```

The output looks like this:

```
[[ 2, 23, 38, 24, 32, 20, 22, 38, 4, 6],
 [35, 20, 47, 49, 29, 39, 15, 15, 8, 28],
 [35, 8, 47, 2, 40, 24, 21, 37, 12, 25],
 [43, 41, 22, 41, 27, 45, 41, 31, 36, 28],
 [19, 17, 8, 39, 40, 24, 43, 16, 33, 22]]
```

We will decompose matrix  $A$  using `TruncatedSVD`.

### Using TruncatedSVD

The `TruncatedSVD` function of `sklearn.decomposition` takes an input parameter, `n_components`. Let's declare the `TruncatedSVD` object (known as `svd`) and assume there are three topics (`n_components=3`). I will explain `n_components` later:

```
from sklearn.decomposition import TruncatedSVD
svd = TruncatedSVD(n_components=3)
```

To decompose matrix  $A$ , we just need to use the `.fit()` function:

```
svd.fit(A)
```

The `svd` object will have the three matrices mentioned in *Figure 4.8*. The singular values in the diagonal of  $\Sigma$  can be obtained as follows:

```
Sigma = svd.singular_values_
```

The dimension of  $\Sigma$  is the number of topics by the number of topics. There are three topics in our case; thus, the dimension is  $3 \times 3$ . The singular values in  $\Sigma$  can be obtained as follows:

```
np.round(Sigma,1)
```

The output is as follows:

```
[197.8, 55.2, 38.5]
```

The matrix that we are interested in is  $V^T$ . Its dimension is 3 x 10 because it is the number of topics multiplied by the number of words. It can be obtained by using `svd.components_`. I will just round the number of words to the first decimal point so as not to look too busy:

```
np.round(svd.components_, 1)
```

The 3 x 10 matrix looks like this:

```
[[0.3, 0.3, 0.4, 0.4, 0.4, 0.4, 0.3, 0.3, 0.2, 0.3],  
[-0., -0.2, 0.7, -0.4, 0.2, -0.1, -0.3, 0.3, -0.4, -0.],  
[0.3, 0., 0.2, 0.5, -0.4, 0.3, -0.4, -0.4, -0.3, 0.1]]
```

How much variance can be explained by the three topics? We can answer this question by obtaining the explained variance ratio. Again, I will round it to two decimal points so they do not look too busy:

```
np.round(svd.explained_variance_ratio_, 2)
```

The results are as follows:

```
[0.14, 0.42, 0.2]
```

The word “Truncated” in TruncatedSVD implies something will be truncated and some small values will be lost. Obviously, if the truncated values are all zeros, there is nothing to lose. The number of topics (`n_components=3`) in  $\Sigma$  determines the truncation. We hope to find a smaller number of topics to represent the information (or, mathematically, variance) in the original five documents. Here, we represent the original five documents with three topics. If we represent the five documents with five topics, basically there is nothing to lose, but it does not serve our purpose. How much information in the five documents is represented by the three topics? Let's sum up the explained variance ratio to find out:

```
svd.explained_variance_ratio_.sum()
```

The output is 0.7638465040646414. The three topics can explain 76.3% of the five documents.

The TruncatedSVD() function is a wrapper for randomized\_SVD(). Let me add an optional section to perform randomized\_SVD().

## randomized\_SVD

Let's dig deeper. You may want to see each matrix U, Sigma, and  $V^T$ . We can use `randomized_svd` because the TruncatedSVD function in `scikit-learn` is just a wrapper function of `randomized_svd`:

```
from sklearn.utils.extmath import randomized_svd  
U, Sigma, VT = randomized_svd(A,  
n_components=3,
```

```
n_iter=10,  
random_state=2)
```

The sizes of the matrices can be obtained using the following:

```
U.shape, Sigma.shape, VT.shape
```

This shows the following output:

```
(5, 3), (3, 3), (3, 10))
```

U is a  $5 \times 3$  matrix:

```
np.round(U, 1)
```

The results are as follows:

```
[[ 0.3, 0.4, -0.3],  
 [ 0.5, 0.2, 0.8],  
 [ 0.4, 0.6, -0.3],  
 [ 0.6, -0.4, 0. ],  
 [ 0.4, -0.5, -0.4]]
```

Let's print out the diagonal values in  $\Sigma$ . Note that they are the same as the result of TruncatedSVD. Again, I round the output to the first decimal point:

```
np.round(Sigma, 1)
```

The results are as follows:

```
[197.8, 55.2, 38.5]
```

Let's print out  $V^T$ . Again, I round the output to the first decimal point. Note that it is the same as the result of TruncatedSVD:

```
np.round(VT, 1)
```

The results are as follows:

```
[[0.3, 0.3, 0.4, 0.4, 0.4, 0.4, 0.3, 0.3, 0.2, 0.3],  
 [-0., -0.2, 0.7, -0.4, 0.2, -0.1, -0.3, 0.3, -0.4, -0.],  
 [0.3, 0., 0.2, 0.5, -0.4, 0.3, -0.4, -0.4, -0.3, 0.1]]
```

That's the framework of SVD. In the next section, we will apply TruncatedSVD to the real AG news data.

## Using TruncatedSVD for LSI with real data

In this section, we will build a model using TruncatedSVD and real data. Let me outline the tasks first. This task list is a general procedure when you build an LSI:

- Loading the data
- Creating TF-IDF
- Using TruncatedSVD to build a model
- Interpreting the outcome

For an effective learning outcome, we will just use five documents in the data so we can print out the words. Once you know how the process works, you can replicate it for the entire data.

### Loading the data

In the *Preface* of the book, we said that we will use the sampled AG corpus of news articles throughout the book. Using one dataset will help you to focus on the techniques rather than orient yourself to different data, although there is still some value in exposing it to different datasets. The original AG corpus of news articles is a large collection of more than 1 million news articles from more than 2,000 news sources. A smaller collection that sampled news articles on “world,” “sports,” “business,” and “science” was constructed and used as a text classification benchmark in the paper by Zhang, Zhao, and LeCun [1]. This book uses this dataset. There are 120,000 and 7,600 news articles in the training and testing samples respectively. The data has four classes – class “1” is news about “world,” “class “2” is news about “sports,” “3” is “business,” and “4” is “sci/tech.” Let’s load the data:

```
import pandas as pd
import numpy as np
pd.set_option('display.max_colwidth', -1)
path = '/content/gdrive/My Drive/data/gensim'
train = pd.read_csv(path + "/ag_news_train.csv")
```

Let’s take a look at the first five documents:

```
docs = train['Description'][0:5]
print(docs)
```

The first five texts of this dataset are as follows:

```
0    Reuters - Short-sellers, Wall Street's dwindling band of ultra-cynics, are seeing green again.
1    Reuters - Private investment firm Carlyle Group, which has a reputation for making well-timed and occasionally controversial plays in the defense industry, has quietly placed its bets on another part of the market.
```

```

2    Reuters - Soaring crude prices plus worries about the economy and
the outlook for earnings are expected to hang over the stock market
next week during the depth of the summer doldrums.

3    Reuters - Authorities have halted oil export flows from the main
pipeline in southern Iraq after intelligence showed a rebel militia
could strike infrastructure, an oil official said on Saturday.

4    AFP - Tearaway world oil prices, toppling records and straining
wallets, present a new economic menace barely three months before the
US presidential elections.

```

So far so good! Now, we will convert the raw texts to TF-IDF.

## Creating TF-IDF

We will create TF-IDF for the five documents:

```

TfidfVectorizer:
from nltk.tokenize import RegexpTokenizer
from sklearn.feature_extraction.text import TfidfVectorizer

```

We want to separate a sentence into words without punctuation. Tokenization with regular expressions helps us do that. The \w+ notation searches for alphabets or numbers:

```
tokenizer = RegexpTokenizer(r'\w+')
```

With the tokenizer, we will create TF-IDF by using `TfidfVectorizer()`:

```

tfidf = TfidfVectorizer(lowercase=True,
stop_words='english',
ngram_range = (1,1),
tokenizer = tokenizer.tokenize)

```

Now, we will apply `tfidf` to the documents:

```
train_data = tfidf.fit_transform(docs)
```

Let's see the unique words in our example:

```

terms = tfidf.get_feature_names_out()
print(terms)

```

The output looks like the following:

```

['afp', 'authorities', 'band', 'barely', 'bets', 'carlyle',
'controversial', 'crude', 'cynics', 'defense', 'depth', 'doldrums',
'dwindling', 'earnings', 'economic', 'economy', 'elections',
'expected', 'export', 'firm', 'flows', 'green', 'group', 'halted',
'hang', 'industry', 'infrastructure', 'intelligence', 'investment',
'iraq', 'main', 'making', 'market', 'menace', 'militia', 'months',
'months', 'oil', 'outlook', 'over', 'pipeline', 'rebel', 'stocks',
'southern', 'strike', 'summer', 'tearaway', 'toppling', 'world']

```

```
'new', 'occasionally', 'official', 'oil', 'outlook', 'pipeline',
'placed', 'plays', 'plus', 'present', 'presidential', 'prices',
'private', 'quietly', 'rebel', 'records', 'reputation', 'reuters',
's', 'said', 'saturday', 'seeing', 'sellers', 'short', 'showed',
'soaring', 'southern', 'stock', 'straining', 'street', 'strike',
'summer', 'tearaway', 'timed', 'toppling', 'ultra', 'wall', 'wallets',
'week', 'world', 'worries']
```

There are 77 unique words in these five documents. We can get the word count by doing the following:

```
len(tfidf.get_feature_names_out())
```

The output is as follows:

```
77
```

We are making good process. Now, we are ready to apply TruncatedSVD.

## Using TruncatedSVD to build a model

We will declare a model, `lsi`. There are only five documents, so we will just set the number of topics to 3:

```
from sklearn.decomposition import TruncatedSVD
lsi = TruncatedSVD(n_components=3)
```

`sklearn` has two ways to execute TruncatedSVD. One is `.fit(X)`, which fits the model only, and the other one is `.fit_transform(X)`, which fits the model and performs dimensionality reduction on `X`. Because we want to see the reduced `X` later, I will use `.fit_transform(X)`:

```
lsi.fit_transform(train_data)
```

The model performs truncatedSVD, as shown in *Figure 4.8*, and breaks the `train_data` matrix into three matrices –  $U$ ,  $\Sigma$ , and  $V^T$ . Let's interpret the outcome.

## Interpreting the outcome

First, let's print out  $\Sigma$ . Its dimension is the number of topics multiplied by the number of topics. We have three topics, so its dimension is  $3 \times 3$ . The singular values in matrix  $\Sigma$  are as follows:

```
Sigma = lsi.singular_values_
print(Sigma)
```

The output looks like this:

```
[1.05609076, 1.01734796, 0.99447695]
```

Next, let's print out the topic-word matrix  $V^T$ . Its dimension is 3 x 77 because of 3 topics and 77 unique words:

```
V_transpose = lsi.components_.T
V_transpose.shape
(3, 77)
```

I will convert matrix  $V^T$  to a DataFrame and attach the column names to show you the topic-word matrix:

```
VT_names = pd.DataFrame(VT, columns=terms)
print(VT_names)
```

The rows shown in *Table 4.1* are the topics, and the columns are the words. The values are the weights of the words for the topics.

	<b>afp</b>	<b>authorities</b>	<b>band</b>	<b>barely</b>	<b>bets</b>	<b>carlyle</b>	<b>...</b>	<b>world</b>	<b>worries</b>
0	0.12	0.11	0.08	0.12	0.09	0.09	...	0.12	0.12
1	-0.13	-0.1	0.1	-0.13	0.13	0.13	...	-0.13	0.08
2	-0.03	0.05	0.26	-0.03	-0.06	-0.06	...	-0.03	-0.09

Table 4.1 – Matrix  $V^T$

Note that the columns are sorted by the words in alphabetical order. To see the words of a topic from the most to the least important, let's take a topic and then sort the words by the weights in descending order. Let's see topic 0 by taking row 0 (`VT_names[0:1]`) and sorting the columns in descending order:

```
VT_names[0:1].sort_values(by=0, axis=1, ascending=False)
```

*Table 4.2* shows the top words of topic 0 in descending order. The words include “oil,” “prices,” “market,” and “presidential,” which implies this topic is about oil prices and the political economy.

	<b>oil</b>	<b>reuters</b>	<b>prices</b>	<b>market</b>	<b>wallets</b>	<b>presidential</b>	<b>menace</b>	<b>...</b>
0	0.27	0.23	0.19	0.17	0.12	0.12	0.12	...

Table 4.2 – The top words of topic 0

Next, let's review topic 1. I will take row 1 (`VT_names[1:2]`), then sort the columns in descending order:

```
VT_names[1:2].sort_values(by=1, axis=1, ascending=False)
```

The top words in *Table 4.3* include "market," "industry," and "private." Remember, we only use five documents to show the modeling process. The results are not representative:

	<b>market</b>	<b>industry</b>	<b>timed</b>	<b>private</b>	<b>reputation</b>	<b>plays</b>	<b>placed</b>	...
1	0.17	0.13	0.13	0.13	0.13	0.13	0.13	...

Table 4.3 – The top words of topic 1

Finally, let's see the top words of topic 2. I will take row 2 (`VT_names [2 : 3]`) and then sort the columns in descending order:

```
VT_names[2:3].sort_values(by=2, axis=1, ascending=False)
```

The top words are shown in *Table 4.4*:

	<b>seeing</b>	<b>cynics</b>	<b>s</b>	<b>sellers</b>	<b>short</b>	<b>dwindling</b>	<b>street</b>	<b>ultra</b>	...
2	0.26	0.26	0.26	0.26	0.26	0.26	0.26	0.26	...

Table 4.4 – The top words of topic 2

Having reviewed the topics, let's finally understand how much these topics can represent the data – that is, how much these topics can explain the variance. We will use `explained_variance_ratio_` to get the explainability:

```
lsi.explained_variance_ratio_.sum()
```

The result is `0.5229249537209735`, which means these three topics can explain 52.2% of the variance in the five documents. Apparently, if you specify more topics, the variance explained will be higher. You should now have a good understanding of TruncatedSVD and the inputs and outputs of each step. You can apply the same process to the entire data.

You may ask how you determine the optimum number of topics. In the next chapter, we will use Gensim to determine the optimum number of topics. We will end this chapter here and summarize what we have learned.

## Summary

In this chapter, we learned how LSI was developed based on SVD. We learned a large document-term matrix can be decomposed into three matrices through SVD. We also learned about a few basic properties of matrix operations and transformation matrices, as well as eigenvectors and eigenvalues, to understand SVD. After that, we applied SVD to real data to observe the outcome.

Gensim has packaged LSI in a few lines of code for efficient production. While this chapter walked you through the theoretical construction of LSI, *Chapter 6, Latent Semantic Indexing with Gensim*, will teach you how to build an LSI model for production. However, there is an important NLP concept that you should learn about before learning about LSI with Gensim. It is cosine similarity. It is a fundamental concept used extensively in the NLP field, including modern word embeddings and large language models. Let's move on to the next chapter.

## Questions

1. What is an orthonormal matrix?
2. What is a transformation matrix?
3. Name a few applications of the transformation matrix in our daily lives.
4. What is SVD?
5. Explain how SVD is applied in LSI.
6. Which of the three matrices in SVD relate latent topics to words?
7. How do you show the percentage of the topics that can explain the variance in the texts of documents?
8. List the procedure to develop an LSI model with TruncatedSVD.

# 5

## Cosine Similarity

One significant automation of NLP is its capability to find words or documents that are semantically related. In a search engine, we want to find other words that are similar to our search words. What a search engine does is measure the similarity of two words or the similarity of two documents. We cannot compare two words or documents simply using their alphabets. But we can compare two words or documents mathematically in their latent vector space. In *Chapter 4, Latent Semantic Analysis with scikit-learn*, we learned how to represent documents as vectors. Because documents are represented as vectors, they can be compared mathematically. How do we measure the similarity of two vectors? The measure is called **cosine similarity**. This measure is widely used in NLP.

Search engines, whether powered by pre-LLM techniques (such as Word2Vec or Doc2Vec) or LLMs (such as BERT word embeddings), use cosine similarity, among other metrics, such as Euclidean distance or Manhattan distance, to measure the similarity between a user's search query and the documents in their index. Cosine similarity calculates the cosine of the angle between these vectors, providing a measure of how closely aligned the query vector is with each document vector. Documents with a smaller angle (i.e., a higher cosine similarity) are considered more relevant to the user's query, enabling search engines to rank and retrieve results in order of relevance, improving the overall search experience.

Cosine similarity is not limited to NLP; it is also widely used in image comparison. Any digital images can be presented as vectors. Since they are in vector form, we can apply cosine similarity to calculate the similarity between two images. In this chapter, I will also describe how it is applied to image comparison.

Since the vast amount of words, documents, and images are stored in high-dimensional vector spaces for information retrieval, it is worth mentioning that a practical challenge is **search efficiency**, which gives rise to the **vector database**. A vector database is a type of database that is specifically designed to store and manage vector data. These databases are optimized for efficient storage, retrieval, and querying of vector data. Modern recommendation systems on large-scale e-commerce platforms employ vector databases to store user and item embeddings. **Geographic information systems (GIS)** also use vector databases for spatial data indexing, enabling the rapid retrieval of spatially related information and facilitating location-based services.

All of the above applications lie within the concept of cosine similarity. So, the following topics will be covered in this chapter:

- What is cosine similarity?
- How cosine similarity is used in image comparison
- How to use compute cosine similarity with scikit-learn

By the end of this chapter, you will be able to explain cosine similarity and its use cases in NLP and other fields.

## Technical requirements

We will use `numpy`, `sciPy.stats`, and `matplotlib` for its 3D plotting capacity. Let's load the module:

```
from sklearn.metrics.pairwise import cosine_similarity
```

Code files are available on GitHub: <https://github.com/PacktPublishing/The-Handbook-of-NLP-with-Gensim/tree/main/Chapter05>.

## What is cosine similarity?

The similarity of two vectors can be measured using cosine similarity. So, let's start with vector properties. Given two vectors, one vector can be projected onto another to show “how much” a vector is pointing in the same direction as the other. *Figure 5.1* shows a 2D graph of the projection of vector  $a$  onto vector  $b$ .

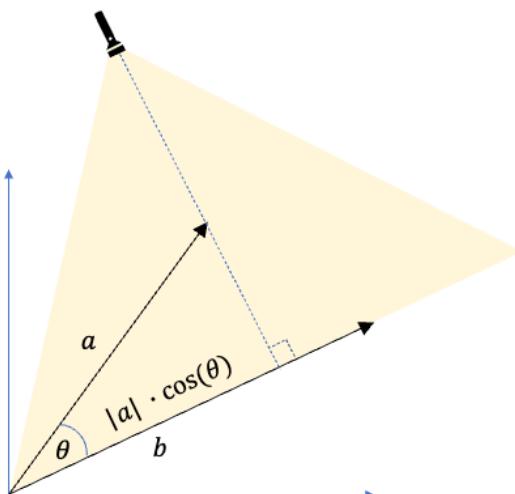


Figure 5.1 – The projection of vector  $a$  on vector  $b$

It is the shadow of vector  $a$  being cast on vector  $b$ . If the angle is small, the shadow will be long. It means the two vectors are very close. If the angle is as large as 90 degrees, the shadow is almost 0. It means the two vectors are not related at all. Therefore, the angle between the two vectors can measure the similarity. The length of the shadow is the length of  $a$  times the cosine of the angle between the two vectors. We will use the dot product of two vectors to mathematically define the similarity.

The **dot product** of vectors is equal to the product of the magnitudes of the two vectors, and the cosine of the angle between the two vectors. This relationship is shown as:

$$\vec{a} \cdot \vec{b} = |\vec{a}| |\vec{b}| \cdot \cos(\theta) \quad Eq. (1)$$

*Eq. (1)* can be re-arranged as *Eq. (2)*:

$$\cos(\theta) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| |\vec{b}|} \quad Eq. (2)$$

If the angle between vector  $a$  and vector  $b$  is small, the two vectors point in similar directions, and the cosine value will be close to 1.0. If the two vectors point in the exact opposite direction, the cosine value will be -1.0. If the two vectors are orthogonal, the cosine value will be 0 and the two vectors are unrelated. The cosine value, called cosine similarity, thus defines the similarity of two vectors. The value of cosine similarity always lies in the interval [-1,1]. A high value in cosine similarity indicates a high level of similarity between two vectors. Given two n-dimensional vectors,  $A$  and  $B$ , the cosine similarity becomes:

$$\cos(\theta) = \frac{\vec{A} \cdot \vec{B}}{|\vec{A}| |\vec{B}|} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}}$$

Cosine similarity, therefore, gives a useful measure of how similar two words are to each other. In the next chapter, when we learn about Word2Vec, we will see an example that shows the similarity of `car` and `vehicle` is 0.78, close to 1.0. It means the two words are closely related. Similarly, as in the previous chapter on LSI and future chapters on LDA and text summarization, documents can be represented as vectors. The relatedness of two documents can be measured by their cosine similarity between the vectors.

## How cosine similarity is used in images

Any digital images can be converted into vectors. The vectors can be compared in the latent vector space by the cosine similarity score to indicate the “resemblance” of two images. *Figure 5.2* first shows the digital image of the number 5 as a 2D matrix.

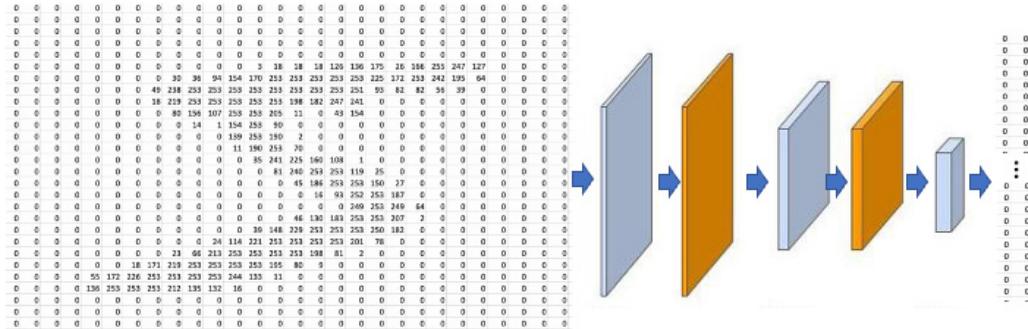


Figure 5.2 – The image of “5” (image from [1])

This 2D matrix can be converted by a neural network such as a **Convolutional Neural Network (CNN)** to become vectors. The square boxes in *Figure 5.2* represent a series of layers, including convolutional, pooling, and fully connected layers, to convert an image to a long vector. Let me describe the process a little bit. Initially, the CNN applies multiple convolutional layers to extract hierarchical features from the image. These layers detect patterns such as edges, textures, and shapes. As the network progresses through convolution and pooling layers, it captures increasingly complex features. Subsequently, the CNN flattens the resulting feature maps into a one-dimensional vector. This vector now represents the image’s learned features in a format suitable for further processing by fully connected layers or other machine learning models. If you want to know how a CNN model works, in the book *Transfer Learning for Image Classification: With Python* [1], I have detailed the function of each layer in a CNN model and the output of a CNN model.

I trust you understand the concept of cosine similarity well now. Let’s practice it with scikit-learn.

## How to compute cosine similarity with scikit-learn

In this section, I will use the `cosine_similarity()` function in scikit-learn to show computation.

First, let’s use the same song, “New York, New York” by Frank Sinatra, that was used in *Chapter 2, Text Representation*:

```
doc_list = [
    "Start spreading the news",
    "You're leaving today (tell him friend)",
    "I want to be a part of it, New York, New York",
    "Your vagabond shoes, they are longing to stray",
    "And steps around the heart of it, New York, New York"
]
```

This document has five sentences. Let's create the bag of words for this document with the `CountVectorizer()` function in scikit-learn:

```
from sklearn.feature_extraction.text import CountVectorizer
import pandas as pd
cv = CountVectorizer()
cv_fit = cv.fit_transform(doc_list)
word_list = cv.get_feature_names_out()
count_list = cv_fit.toarray().sum(axis=0)
```

Then let's print out the result. It has five lists:

```
cv_fit.toarray()
array([
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
0, 1, 0, 0, 0, 1, 0],
[0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 2, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
1, 0, 0, 1, 2, 0, 0],
[0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
1, 0, 1, 0, 0, 0, 1],
[1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 2, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1,
0, 0, 0, 2, 0, 0]
])
```

Let's also make the arrays a data frame to feed in the `cosine_similarity()` function to get the cosine similarity:

```
from sklearn.metrics.pairwise import cosine_similarity
df = pd.DataFrame(cv_fit.toarray())
print(cosine_similarity(df, df))
```

The outputs are as follows:

```
[[1.          0.          0.          0.          0.12909944]
 [0.          1.          0.          0.          0.          ]
 [0.          0.          1.          0.09449112  0.69006556]
 [0.          0.          0.09449112  1.          0.          ]
 [0.12909944  0.          0.69006556  0.          1.          ]]
```

The previous symmetric  $5 \times 5$  matrix shows the cosine similarities between any two sentences. The pair that has the highest cosine similarity is [Sentence 3, Sentence 5]. This is not a surprise because Sentence 3 is "I want to be a part of it, New York, New York", and Sentence 5 is "And steps around the heart of it, New York, New York." Both sentences have two instances of "New York."

## Summary

In this chapter, we learned the definition of cosine similarity. It measures the similarity of two vectors by the angle between the two vectors. If the angle between two vectors is small, the cosine value will be close to 1.0 and the two vectors are considered “similar.” If two vectors are orthogonal, the cosine value will be 0 and the two vectors are unrelated. The value of cosine similarity always lies in the interval [-1,1]. A high value in cosine similarity indicates a high level of similarity between two vectors. This fundamental metric is used throughout NLP, including the pre-LLM techniques and LLM techniques. It is also used in other fields, such as image comparison, to identify the similarities between the image feature vectors.

In the next chapter, we will build production models with **latent semantic indexing (LSI)**.

## Questions

1. Explain the difference between a text keyword search and a semantic search.
2. Please describe cosine similarity.
3. Describe how cosine similarity is used for image comparison.

## References

- Kuo, Chris, “Transfer Learning for Image Classification: With Python Examples,” (November 2022), <https://a.co/d/cmbn8ty>
- Kuo, Chris, “Convolutional Autoencoders for Image Noise Reduction,” <https://towardsdatascience.com/convolutional-autoencoders-for-image-noise-reduction-32fce9fc1763>

# 6

## Latent Semantic Indexing with Gensim

In *Chapter 4, Latent Semantic Indexing with scikit-learn*, we learned about the construction of LSI from SVD and used scikit-learn to perform LSI. We also mentioned that the Gensim library has programmed LSI in a few lines of code for production purposes. In this chapter, we will build the LSI model with Gensim. We will also learn how to determine the right number of topics. I'll also demonstrate to you how to put the model to real use as a search engine. This production-oriented perspective will help data scientists from non-NLP areas to consider stepping into the NLP world.

This chapter covers the following topics:

- Performing text preprocessing
- Performing text representation with BoW and TF-IDF
- Modeling with Gensim
- Using the coherence score to find the optimal number of topics
- Understanding the final model
- Using the model as an information retrieval tool

After completing this chapter, you will be able to build an LSI model with Gensim and develop your model as a part of a search engine.

### Technical requirements

Make sure you install Gensim by doing `pip install gensim`.

You can find the code files at <https://github.com/PacktPublishing/The-Handbook-of-NLP-with-Gensim/tree/main/Chapter06>.

## Performing text preprocessing

As I described in the *Preface*, this book will use the sampled AG's corpus of news articles made public by Zhang, Zhao, and LeCun [3]. This dataset is a smaller collection that sampled news articles on "world," "sports," "business," and "science" [4]. It has been used extensively in many NLP modeling projects and is available in Kaggle, PyTorch, Huggingface, and TensorFlow. The data has four classes – class "1" is news about "world affairs," class "2" is news about "sports," class "3" is about "business," and class "4" is about "science/tech." Let's print out two records for each class just to understand the text data. The code is like this:

```
import pandas as pd
pd.set_option('display.max_colwidth', -1)
path = "/content/gdrive/My Drive/data/gensim"
train = pd.read_csv(path + "/ag_news_train.csv")
train.groupby('Class Index').head(2)
```

The following table shows the first two records:

Class Index	Title	Description
1	Venezuelans Vote Early in Referendum on Chavez Rule (Reuters)	Reuters - Venezuelans turned out early\and in large numbers on Sunday to vote in a historic referendum\that will either remove left-wing President Hugo Chavez from\office or give him a new mandate to govern for the next two\years.
1	S.Koreans Clash with Police on Iraq Troop Dispatch (Reuters)	Reuters - South Korean police used water cannon in\central Seoul Sunday to disperse at least 7,000 protesters\urging the government to reverse a controversial decision to\send more troops to Iraq
2	Phelps, Thorpe Advance in 200 Freestyle (AP)	AP - Michael Phelps took care of qualifying for the Olympic 200-meter freestyle semifinals Sunday, and then found out he had been added to the American team for the evening's 400 freestyle relay final. Phelps' rivals Ian Thorpe and Pieter van den Hoogenband and teammate Klete Keller were faster than the teenager in the 200 free preliminaries.
2	Reds Knock Padres Out of Wild-Card Lead (AP)	AP - Wily Mo Pena homered twice and drove in four runs, helping the Cincinnati Reds beat the San Diego Padres 11-5 on Saturday night. San Diego was knocked out of a share of the NL wild-card lead with the loss and Chicago's victory over Los Angeles earlier in the day.
3	Wall St. Bears Claw Back Into the Black (Reuters)	Reuters - Short-sellers, Wall Street's dwindling\band of ultra-cynics, are seeing green again.
3	Carlyle Looks Toward Commercial Aerospace (Reuters)	Reuters - Private investment firm Carlyle Group, which has a reputation for making well-timed and occasionally\controversial plays in the defense industry, has quietly placed\its bets on another part of the market.
4	'Madden,' 'ESPN' Football Score in Different Ways (Reuters)	Reuters - Was absenteeism a little high\on Tuesday among the guys at the office? EA Sports would like\to think it was because "Madden NFL 2005" came out that day,\and some fans of the football simulation are rabid enough to\take a sick day to play it.
4	Group to Propose New High-Speed Wireless Format (Reuters)	Reuters - A group of technology companies\including Texas Instruments Inc. (TXN.N), STMicroelectronics(STM.PA) and Broadcom Corp. (BRCM.O), on Thursday said they\will propose a new wireless networking standard up to 10 times\the speed of the current generation.

Table 6.1 – An overview of some samples of the AG news data

As we learned in *Chapter 3, Text Wrangling and Preprocessing*, the quality of an LSI model outcome depends on good text pre-processing. We learned that pre-processing includes several key steps: (a) tokenization, (b) lowercase conversion, (c) stop word removal, (d) punctuation removal, (e) stemming, and (f) lemmatization. We learned how to code these steps in spaCy, NLTK, and Gensim. Here, we will use Gensim to do text pre-processing.

Let's first declare an empty list called `text_tokenized` to collect all the preprocessed documents:

```
from gensim.parsing.preprocessing import preprocess_string
text_tokenized = []
```

Then, the code iterates each document and performs text preprocessing by using the `preprocess_string()` function:

```
for doc in train['Description']:
    k = preprocess_string(doc)
    text_tokenized.append(k)
```

The outcome, `text_tokenized`, is a list of lists. Let's take a look at the first two processed documents:

```
text_tokenized[0:2]
```

Here's the output:

```
[[['reuter', 'short', 'seller', 'wall', 'street', 'dwindl', 'band',
  'ultra', 'cynic', 'see', 'green'],
  ['reuter', 'privat', 'invest', 'firm', 'carlyl', 'group', 'reput',
  'make', 'time', 'occasion', 'controversi', 'plai', 'defens',
  'industri', 'quietli', 'place', 'bet', 'market']]
```

The outcome shows that the words of each document are tokenized, and words are lemmatized to their root form, such as “dwindle” becoming “dwindl” and “defense” becoming “defens.”

LSI modeling requires the text input as **Bag-of-Words (BoW)** or **Term Frequency-Inverse Document Frequency (TF-IDF)**. which we will do accordingly.

## Performing word embedding with BoW and TF-IDF

Let's first do BoW and TF-IDF. We learned how to prepare BoW and TF-IDF in *Chapter 2, Text Representation*. BoW is actually the count frequency of words, while its variation, TF-IDF, is designed to reflect the importance of a word in a document of a corpus.

We will first use the `Dictionary` class to build and manage dictionaries of terms (words or tokens). It creates a mapping between unique terms in a corpus and their integer IDs. This is actually the BoW:

```
from gensim.corpora import Dictionary
gensim_dictionary = Dictionary()
```

Let's examine the dictionary list object, `gensim_dictionary`. How many unique words are in it? Let's check the length of this list to get the number of words:

```
len(gensim_dictionary)
```

We get the following output:

```
40360
```

So, there are 40,360 words!

Now, we will create the BoW.

## BoW

We create the BoW by using the `.doc2bow()` function:

```
bow_corpus = [gensim_dictionary.doc2bow(doc,
    allow_update=True) for doc in text_tokenized]
```

The `bow_corpus` object is a list of lists. Each word in a list is presented in a two-tuple format (ID and count). Let's print out two records:

```
print(bow_corpus[:2])
```

The output is as follows:

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

The first two-tuple, `(0, 1)`, means the word ID = 0, occurring once, and the second tuple, `(1, 1)`, means the word ID = 1, occurring once, and so on.

This is easier to understand if we present the word IDs as words. So, let's use `gensim_dictionary[id]` to refer to the words by their IDs:

```
id_words = [[(gensim_dictionary[id], count) for id, count
    in line] for line in bow_corpus]
```

Let's print out a few examples:

```
print(id_words)
```

The output is as follows:

```
[[(('band', 1), ('cynic', 1), ('dwindl', 1), ('green', 1), ('reuter', 1), ('see', 1), ('seller', 1), ('short', 1), ('street', 1), ('ultra', 1), ('wall', 1)], [('reuter', 1), ('bet', 1), ...]
```

The preceding output is the BoW for our data. Now, let's generate TF-IDF from the BoW.

## TF-IDF

To do TF-IDF, we will use the `TfidfModel` class:

```
from gensim.models import TfidfModel, LsiModel
tfidf = TfidfModel(bow_corpus)
tfidf_corpus = tfidf[bow_corpus]
```

To save space, we do not need to print out some records in `tfidf_corpus` here. We will save the BoW corpus, the TF-IDF corpus, and the dictionary for production use.

Save the BoW corpus as follows:

```
import pickle
file = open(path + "/BoW_AGnews_corpus.pkl", 'wb')
pickle.dump(bow_corpus, file)
file.close()
```

Then, save the TF-IDF corpus:

```
file = open(path + "/tfidf_AGnews_corpus.pkl", 'wb')
pickle.dump(tfidf_corpus, file)
file.close()
```

Finally, save the dictionary:

```
from gensim.test.utils import datapath
dict_file = datapath(path + "/gensim_dictionary_AGnews")
gensim_dictionary.save(dict_file)
```

Now, we will learn to model LSI with Gensim.

## Modeling with Gensim

The Gensim's LSI module implements fast, truncated SVD. The module is actually engineered to process very large corpora through distributed computing. It can also take a stream of corpora as input. Let's use the `LsiModel()` function to build our first model. We will build it with the corpus in BoW and TF-IDF.

## BoW

Let's start with the model with the corpus in BoW:

```
import gensim
lsi_model = gensim.models.lsimodel.LsiModel(
    corpus=bow_corpus,
    id2word=gensim_dictionary,
    num_topics=20)
```

Let's explain the previous parameters:

- `corpus`: This is the corpus of our data. It is a required input parameter. Here we use `bow_corpus`.
- `id2word`: This is a list of the IDs of the words. We assign our dictionary as `gensim_dictionary`. If we do not assign a dictionary, Gensim will show a warning message but still use its default dictionary.
- `num_topics`: This is the number of latent dimensions in SVD. The latent dimensions are the topics in LSI. We restrict the number of topics to 20 for this first model, just to observe the results. Later, we will learn how to use the coherence score to determine the optimal number of topics.

The model is built and stored in `lsi_model`. Let's print out the topics. We can use `pprint`, which is designed to pretty-print the results:

```
import pprint as pp
pp pprint(lsi_model.print_topics())
```

The output of the six topics is as follows:

```
[ (0,
  '0.428*"reuter" + 0.347*"new" + 0.284*"said" + 0.265*"fullquot"
+ '0.198*"stock" + 0.161*"investor" + 0.155*"york" + 0.155*"com" +
'0.151*"http" + 0.148*"www"' ),
 (1,
  '-0.336*"fullquot" + 0.299*"said" + 0.287*"quot" + -0.234*"reuter"
+ '0.230*"new" + -0.185*"stock" + -0.176*"http" + 0.174*"year" +
-0.174*"href" +
  '+ -0.174*"www"' ),
 (2,
  '0.754*"new" + -0.436*"said" + -0.263*"quot" + 0.253*"york" +
'-0.106*"reuter" + -0.069*"fullquot" + -0.067*"offici" + -0.063*"kill"
+ '-0.051*"iraq" + -0.046*"thursday"' ),
 (3,
  '0.899*"quot" + -0.232*"said" + -0.121*"year" + 0.110*"new" +
0.085*"fullquot" + -0.074*"reuter" + -0.058*"offici" + -0.057*"kill"
+ '-0.050*"wednesday" + -0.050*"oil"' ),
```

```
(4,
'0.635*"year" + -0.473*"said" + -0.275*"new" + 0.165*"game" +
0.132*"world" + 0.098*"time" + 0.091*"season" + 0.090*"win" +
-0.082*"york" +
'0.079*"team"' ),
(5,
'0.500*"compani" + -0.221*"game" + -0.192*"reuter" + 0.172*"corp" +
'0.150*"softwar" + 0.146*"servic" + -0.134*"state" + -0.129*"sundai" +
'0.124*"million" + -0.118*"win"' ),
```

The rule to interpret the results is this – a word with a positive weight means the topic is about that word, and a word with a negative weight means the topic is not about that word. Let’s use this rule on topic “5” to understand how the model classifies our news article. It has positive words such as “compani,” “softwar,” “servic,” and “million.” These words are more likely to be associated with technology companies and services. Topic “5” has negative words such as “game,” “reuter,” state,” “sundai,” and “win,” which are more likely to occur in sports-related news. Recall that our AG news data has four classes – “world,” “sports,” “business,” and “sci/tech.” Topic “5” is more likely to relate to “business” news and less likely to be about “sports” or “world.” In other words, any news that is about the services of software companies is likely to be included under topic “5”.

Let’s use the same rule to understand topic “0.” It has positive words such as “stock,” “investor,” “New,” “York,” and “com.” So, we can guess that this topic is about business news. News about the stock market or investors can be classified under topic “0”. However, topic “0” also contains general words, such as “reuter,” “said,” and “fullquot,” that appear in other topics. Their presence makes interpretation difficult. This is because these words appear frequently in the AG news data, and we feed the model with the BoW corpus. Those frequent words are considered important in BoW.

What can we do to improve the model results? We know that the BoW method simply measures the word count but not the significance of a word. The alternative method, TF-IDF, is designed to reflect the importance of a word in a document of a corpus.

## TF-IDF

There are frequent words in the AG news data that do not carry any special meaning and can be down-weighted, such as “reuter” or “news.” TF-IDF can adjust the appropriate weights. Let’s build our second model with the TF-IDF corpus:

```
import gensim
lsi_model = gensim.models.lsimodel.LsiModel(
    corpus=tfidf_corpus,
    id2word=gensim_dictionary,
    num_topics=20)
```

The preceding code is the same as the code for BoW, except we assign the TF-IDF corpus, `tfidf_corpus`. Let's print out the topics:

```
import pprint as pp
pp pprint(lsi_model.print_topics())
```

The output of the six topics is as follows. It is important to note that these six topics do not relate to the previous six topics:

```
[(0,
  '0.200*"reuter" + 0.184*"new" + 0.178*"said" + 0.132*"quot" +
  0.130*"year" + '0.126*"compani" + 0.125*"stock" + 0.122*"fullquot" +
  0.118*"oil" + '0.115*"york"' ),
 (1,
  '-0.417*"fullquot" + -0.251*"stock" + -0.216*"investor" +
  -0.213*"reuter" + -0.211*"http" + -0.210*"href" + -0.209*"www" +
  -0.209*"quickinfo" + -0.209*"aspx" + -0.205*"ticker"' ),
 (2,
  '-0.232*"kill" + -0.184*"fullquot" + -0.177*"iraq" + -0.160*"baghdad" +
  + -0.147*"iraqi" + 0.147*"compani" + -0.142*"reuter" + -0.128*"bomb" +
  -0.128*"peopl" + -0.126*"palestinian"' ),
 (3,
  '-0.424*"oil" + 0.364*"quot" + -0.328*"price" + -0.164*"crude" +
  + 0.160*"fullquot" + -0.141*"barrel" + 0.133*"microsoft" +
  0.119*"softwar" + -0.105*"record" + -0.102*"suppli"' ),
 (4,
  '0.268*"game" + 0.181*"win" + 0.165*"season" + 0.153*"leagu" +
  0.150*"night" + 0.144*"victori" + 0.141*"team" + -0.136*"quot" +
  0.126*"seri" + ' '-0.123*"said"' ),
 (5,
  '-0.845*"quot" + -0.135*"oil" + 0.114*"softwar" + -0.112*"profil" +
  0.106*"compani" + 0.106*"servic" + -0.100*"price" + 0.096*"microsoft" +
  + -0.092*"research" + 0.078*"internet"' ),
```

Let's take topic "5" as an example. The positive words are "softwar," "compani," "service," "microsoft," and "internet." The negative words are "oil," "profil," and "price." This topic is about business news and is less likely to be world news or oil prices. Let's examine Topic "4." Its positive words are "game," "win," "season," "leagu," and "victori." We can confidently say this topic is about sports news. The use of TF-IDF corpus seems to generate more distinct results than the BoW corpus.

However, we still see the frequently used words "fullquot," "said," "quot," and "reuter" in the TF-IDF results, although they are not as prevailing as in the BoW results. Their presence challenges our interpretation of the topics. One solution is to remove known frequently used words from the type of documents you model with. You may recall that we said that a large amount of time in NLP tasks is spent pre-processing. The quality of an NLP model depends on the refinement of words in the pre-processing step. However, in many applications, a gentle treatment to remove words is enough to yield reasonable results.

In summary, we have a few ways to get a better model outcome. One is to build a model with the TF-IDF corpus and a model with the BoW corpus. The model that has more distinct topics determines the choice. It is hard to comment on whether TF-IDF will definitely deliver a better result than BoW. Another way is to remove more domain-specific common words to fine-tune the pre-processing step. For example, there are common words for the corpora, such as “reuter” or “http,” that can be removed. You will need to experiment with several alternatives to pursue finer results.

In the following section, let’s return to a bigger question – how to determine the optimal number of topics. The answer is the **coherence score**.

## Using the coherence score to find the optimal number of topics

The previous section arbitrarily set the number of topics as 20. Is this the optimal number of topics? To investigate this, we need to understand the “scope” of a topic. A topic can have a set of words that are loosely connected, or closely connected. The latter is a distinctive topic, but the former is not distinctive enough. In other words, the “closeness” of words in a topic is an important measure. If a topic has words that are very loosely connected, the topic may be better separated into more than one.

In order to measure the “closeness” of a topic, Röder, Both, and Hinneburg (2015) [5] proposed a metric called the coherence score. The score is defined as the average or median of pairwise word similarities, formed by the top words of a given topic. The value of a coherence score itself doesn’t have a universal meaning because it varies, based on the scoring method and the corpus being analyzed. We will compute and compare the coherence scores of topics to understand the quality of a topic.

We have produced the `lsi_model` model with the TF-IDF corpus. Let’s calculate the coherence score:

```
from gensim.models.coherencemodel import CoherenceModel
coherence = CoherenceModel(
    model=lsi_model,
    corpus=tfidf_corpus,
    dictionary=gensim_dictionary,
    coherence='u_mass')
```

The parameters for the `CoherenceModel` function seem self-explanatory. It requires the model, the corpus, and the dictionary. Here, we encounter something new, the `coherence` parameter, which we will explain in more detail.

There are three ways to compute the coherence score. The most popular one is UMass, the second one is CV (which is also the default metric of Gensim), and the third one is CP. The preceding code uses UMass as the coherence score. Let's explore the three coherence score methods in more detail:

- **Umass coherence:** Umass compares the co-occurrence statistics of words in the corpus to the co-occurrence statistics within each topic. It uses **pointwise mutual information (PMI)** and considers the logarithm of the ratio of these co-occurrence probabilities. A low `U_mass` score suggests more coherent topics.
- **CV coherence:** This method measures the coherence of topics by calculating the pairwise similarity between the top words (usually nouns or adjectives) in each topic. A high CV score indicates more coherent topics.
- **CP coherence:** CP coherence calculates the co-occurrence probabilities between word pairs and compares these probabilities to those in a background corpus. High CP coherence scores indicate more coherent topics.

What we will do is build LSI models for a range of topics and return their coherence scores. Because we use the Umass score, the model that yields the smallest coherence score will be the better model. Let's put the modeling function, `LsiModel()`, and the `CoherenceModel()` function in our user-defined function, called `coherenceScore()`:

```
def coherenceScore(k):  
    lsi_model = LsiModel(  
        corpus=tfidf_corpus,  
        id2word=gensim_dictionary,  
        num_topic=k)  
    coherence = CoherenceModel(  
        model=lsi_model,  
        corpus=tfidf_corpus,  
        dictionary=gensim_dictionary,  
        coherence='u_mass')  
    return coherence.get_coherence()
```

The preceding user-defined function allows us to build a range of models with a different number of topics. It also will return the coherence scores for the models. We will run models of 100, 150, 200, and 250 topics:

```
numTopics = [100, 150, 200, 250]
```

Let's create an empty list, `coherenceScores`, to store the coherence scores.

```
coherenceScores = []
Then let's iterate to build many models and store the coherence scores
in the list coherenceScores:
for k in numTopics:
    c_UMass = coherenceScore(k)
    coherenceScores.append(c_UMass)
```

The only difference between these models is their number of topics. Our goal is to find the model with the number of topics that yields the lowest coherence score. We can plot the scores by the number of topics to find out which one corresponds to the lowest coherence score:

```
from matplotlib import pyplot as plt
plt.plot(coherenceScores)
plt.show()
```

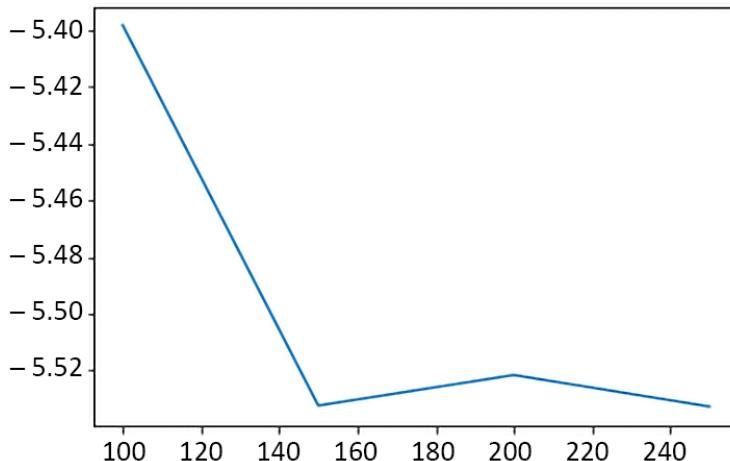


Figure 6.1 – The coherence scores for a range of topics

Figure 6.1 tells us the optimal number of topics is about 150 topics. Hence, we will build the final model with 150 topics with the following code:

```
import gensim
lsi_model = gensim.models.lsimodel.LsiModel(
    corpus=tfidf_corpus,
    id2word=gensim_dictionary,
    num_topics=150)
```

## Saving the model for production

There is one more thing we need to do before actually using the model. We need to save the following objects for future use:

- The dictionary list
- The model
- The BoW object
- The TF-IDF object

Let's start by saving the dictionary list.

Gensim has a utility function called `datapath`. This is where we can specify the physical location of the file. Then, we can save the dictionary using the `.save()` function. Here is the code for it:

```
from gensim.test.utils import datapath
dict_file = datapath(path + "/gensim_dictionary_AGnews")
gensim_dictionary.save(dict_file)
```

Save the model as follows:

```
lsi_model.save(path + "/ag_news_lsi_model")
```

Save the BoW object using pickle:

```
import pickle
file = open(path + "/BoW_AGnews_corpus.pkl", 'wb')
pickle.dump(bow_corpus, file)
file.close()
```

Save the TF-IDF object using pickle:

```
import pickle
file = open(path + "/tfidf_AGnews_corpus.pkl", 'wb')
pickle.dump(bow_corpus, file)
file.close()
```

Let's understand our final model in more detail. Then, we will put the model into production.

The model object is a list of 150 lists because there are 150 topics in our case. The first two topics look like the following:

```
lsi_model.get_topics()[0:2]
```

Here is the output:

```
[ 3.05711460e-03, 2.64117477e-04, 9.95368227e-04, ..., 1.13412063e-04,
2.80756159e-05, 3.35248270e-05],
[ 1.63416911e-03, 1.74680591e-04, -5.25465611e-05, ..., 1.82614476e-
05, 1.34159268e-05, 2.71150614e-05]]
```

Each list in the list of lists is a topic. It contains the weight/coefficient for each word. There are 40,360 unique words in our dictionary, so the length of a list should be 40,360:

```
len(lsi_model.get_topics()[0])
```

We see the following output:

```
40,360
```

We can show each topic in terms of words. Let's show three topics:

```
lsi_model.show_topics()[0:3]
```

The output looks like this:

```
[(0, '0.199*"reuter" + 0.185*"new" + 0.178*"said" + 0.132*"quot" +
0.131*"year" + 0.126*"compani" + 0.125*"stock" + 0.122*"fullquot" +
0.119*"oil" + 0.116*"york"' ),
(1, '-0.415*"fullquot" + -0.250*"stock" + -0.215*"investor" +
-0.210*"reuter" + -0.210*"http" + -0.209*"href" + -0.208*"www" +
-0.207*"quickinfo" + -0.207*"aspx" + -0.203*"ticker"' ),
(2, '-0.236*"kill" + -0.187*"fullquot" + -0.174*"iraq" +
-0.161*"baghdad" + 0.150*"compani" + -0.148*"iraqi" + -0.146*"reuter" +
-0.131*"palestinian" + -0.128*"bomb" + -0.125*"peopl"' ),
```

Let's examine a document to see which topic it belongs to. I will arbitrarily select the 45<sup>th</sup> document:

```
i = 45
train['Description'][i]
```

The output is as follows:

```
A lawsuit against Gary Winnick, the former chief of Global Crossing,
refocuses attention on what Mr. Winnick knew about his company's
finances as it imploded.
```

The BoW for this document is as follows:

```
bow_corpus[i]
```

The output is as follows:

```
[(93, 1), (206, 1), (513, 1), (514, 1), (515, 1), (516, 1), (517, 1),
(518, 1), (519, 1), (520, 1), (521, 1), (522, 2)]
```

This ID and frequency presentation is hard to understand, as mentioned in the previous section. It is easier to show the actual words with their frequency. Let's repeat the same code to show the words and their frequency:

```
id_words = [[(gensim_dictionary[id], count) for id,
             count in line] for line in bow_corpus[i:(i+1)]]
pp.pprint(id_words)
```

The output is as follows:

```
[[(‘compani’, 1),
(‘financ’, 1),
(‘attent’, 1),
(‘chief’, 1),
(‘cross’, 1),
(‘gari’, 1),
(‘global’, 1),
(‘implod’, 1),
(‘knew’, 1),
(‘lawsuit’, 1),
(‘refocus’, 1),
(‘winnick’, 2)]]
```

This is just the preprocessed result of the 45<sup>th</sup> document. Because our model is built on the TF-IDF text representation, let's see the TF-IDF values for this document:

```
a = tfidf_corpus[i]
pp.pprint(a)
```

The output is as follows:

```
[(93, 0.09256640306973117),
(206, 0.1824096655332926),
(513, 0.22115919307277485),
(514, 0.14015129409880323),
(515, 0.20612060750761854),
(516, 0.22262657288143822),
(517, 0.15833380233728434),
(518, 0.35453549783337673),
(519, 0.23179498753912278),
(520, 0.1802823632220798),
```

```
(521, 0.35453549783337673),  
(522, 0.6594429672721119)]
```

Note that there are 12 words, so there are 12 TF-IDF values. We will feed this list to the model. This will map this document to the 150 topics:

```
a_doc = lsi_model[a]  
print(len(a_doc))
```

Here is the output:

```
150
```

The `a_doc` list contains the two tuples, sorted by topic ID. Let's print it out:

```
pp.pprint(a_doc)
```

The output is as follows:

```
[ (0, 0.032195821813022996),  
(1, 0.0070061879068521035),  
(2, 0.01830094969270196),  
(3, 0.01264810775891833),  
...  
(146, 0.001394712813009084),  
(147, 0.016867685305748538),  
(148, -0.00013671768922682433),  
(149, 0.0028921653309908237)]
```

The value in a 2-tuple is the weighted sum of words for a topic. For example, the value for topic “0” is 0.03219582. It is computed by the formula of topic “0” as follows:

```
(0, '0.199*"reuter" + 0.185*"new" + 0.178*"said" + 0.132*"quot" +  
0.131*"year" + 0.126*"compani" + 0.125*"stock" + 0.122*"fullquot" +  
0.119*"oil" + 0.116*"york"'+...).
```

All the formulas should have 40,360 words. If a word is not in the word list of topic “0,” it is zero. Document “45” only has 12 words, so most of the words are assigned zeros. Let's sort the topics by the values and show the top five most relevant topics:

```
import pprint as pp  
a_doc.sort(key=lambda a: a[1], reverse=True)  
pp.pprint(a_doc[0:5])
```

The output is as follows:

```
[ (28, 0.05444252523051011),  
 (53, 0.03593073348251731),  
 (0, 0.032195821813022996),  
 (24, 0.026456723043667373),  
 (41, 0.02445141282153991) ]
```

The first topic for document “45” is topic “28.” Let’s show topic “28”:

```
lsi_model.show_topics()[28]
```

The output is as follows:

```
(28, '0.284*"compani" + -0.183*"servic" + -0.178*"million" +  
 0.168*"execut" + -0.166*"year" + 0.158*"game" + -0.150*"billion" +  
 0.147*"chief" + -0.133*"nuclear" + 0.129*"market"')
```

Recall that document “45” is a business news story about a lawsuit against the chief of a company on the financial matter: “A lawsuit against Gary Winnick, the former chief of Global Crossing, refocuses attention on what Mr. Winnick knew about his company’s finances as it imploded.” The model finds some level of relevance to topic “28.”

Now that we understand the outcome of a model, we will move it to production. Let’s see whether the model is ready for information retrieval.

## Using the model as an information retrieval tool

Establishing a search engine involves many significant engineering tasks, so in this section, we’ll go through the essential steps to deploy an LSI model to be part of a search engine. These steps are as follows:

1. Load the saved objects.
2. Preprocess the new document.
3. Score the new document to get the latent topic scores.
4. Calculate the similarity scores with the new document.
5. Find documents with high similarity scores.

First, we will load the four saved objects. The objects include the dictionary list, the model, the BoW object, and the TF-IDF object.

## Loading the dictionary list

Gensim has a utility function called datapath. It points to the physical location of the file. Here is the code for it:

```
from gensim.corpora import Dictionary
from gensim.test.utils import datapath
dict_file = datapath(path + "/gensim_dictionary_AGnews")
gensim_dictionary = Dictionary.load(dict_file)
```

## *Loading the model*

Now, let's load the model with the following:

```
import gensim
lsi_model = gensim.models.lsimodel.LsiModel.load(path + "/ag_news_lsi_
model")
```

## *Loading the BoW object*

You can use pickle to load the BoW object:

```
import pickle
# open a file, where you stored the pickled data
file = open(path + "/BoW_AGnews_corpus.pkl", 'rb')
bow_corpus = pickle.load(file)
# close the file
file.close()
```

## *Loading the TF-IDF object*

You can use pickle to load the TF-IDF object:

```
import pickle
# open a file, where you stored the pickled data
file = open(path + "/tfidf_AGnews_corpus.pkl", 'rb')
bow_corpus = pickle.load(file)
# close the file
file.close()
```

Once the objects are loaded, we will conduct the keyword search.

## Preprocessing the new document

Let's suppose we are interested in any economic news relating to the economic outlook and general prices, and our keywords are "crude prices," "inflation," "economy," and "outlook earnings." Let's type all these words in a new document as "Crude prices inflation the economy outlook earnings." Note that we can use multiple keywords.

```
doc = "Crude prices inflation the economy outlook earnings"
```

Because our new document is a raw sentence, it should be preprocessed, just as it was with the training data. We have learned that the preprocessing includes several key steps: (a) tokenization, (b) lowercase conversion, (c) stop word removal, (d) punctuation removal, and (e) stemming. We used the `preprocess_string()` function for pre-processing, so we will do the same for the new document. Here is the code for it:

```
doc_tokenized = preprocess_string(doc)
doc_tokenized
```

The raw document is now tokenized and lemmatized, as shown here:

```
['crude', 'price', 'inflat', 'economi', 'outlook', 'earn']
```

The `gensim_dictionary` dictionary was built for the training data and used to get the bag of words. We shall apply the same dictionary to the new document to get the BoW. The following is the code for it:

```
vec_bow = gensim_dictionary.doc2bow(doc_tokenized)
```

The BoW looks like this:

```
pp pprint(vec_bow)
```

The output is as follows:

```
[(28, 1),
(31, 1),
(32, 1),
(35, 1),
(37, 1),
(2099, 1)]
```

The TF-IDF of these words looks like this:

```
from gensim.models import TfidfModel
tfidf = TfidfModel(bow_corpus)
vec_tfidf = tfidf[vec_bow]
pp pprint(vec_tfidf)
```

The output is as follows:

```
[(28, 0.413722234207753),  
(31, 0.36285954267555626),  
(32, 0.3916480509570918),  
(35, 0.4699070384454736),  
(37, 0.2806698374889378),  
(2099, 0.49415633855722196)]
```

With the TF-IDF values, we can score the document.

## Scoring the document to get the latent topic scores

The `lsi_model` model was built on the TF-IDF values. We will enter the TF-IDF values of the document to get the score, `vec_lsi`:

```
vec_lsi = lsi_model[vec_tfidf]
```

There are 150 scores for the 150 topics. Let's print out a few records:

```
print(vec_lsi)
```

The output is as follows:

```
[(0, 0.267271677771786),  
(1, -0.1621443525198505),  
(2, 0.20268363003855583),  
(3, -0.6339930030487946),  
(4, -0.1314557024167698),  
(5, -0.1751935709279391),  
(6, 0.13491543897394606),  
(7, 0.08094112501762366),  
(8, 0.039317913231694104),  
(9, 0.02756727945165236),  
(10, 0.15484079886854193), ...]
```

You can double-check that the length is 150:

```
len(vec_lsi)  
150
```

## Calculating the similarity scores with the new document

Now, we will find documents that are similar to the new document. In our case, there are 120,000 documents. Each of the documents is represented by a vector of 150 topic scores. The new document is also a vector of 150 latent scores. We will find the document vectors that are similar to the vector of the new document. How do we measure the similarity of two vectors? Do you remember the cosine similarity from the previous chapter? It is a popular method in data science that measures the similarity of two vectors by the angle between them. A high value means a high level of similarity. We will use the similarity score here. We will use Gensim's `MatrixSimilarity()` function:

```
from gensim import similarities
index = similarities.MatrixSimilarity(
    lsi_model[tfidf_corpus])
```

Once the index is built, we can use it to compare the similarity of the new document with each of the 120,000 documents. The vector of our new document is `vec_lsi`. We will store the similarity scores in a list called `sims`:

```
sims = index[vec_lsi]
```

The length of the list, `sims`, should be the total number of documents, 120,000, as shown here:

```
len(sims)
120,000
```

If we print out the list, `sims`, we shall see the similarity scores:

```
pp.pprint(sims)
```

The output is as follows:

```
[ 0.15549277, -0.08580323, 0.6605106 , ..., 0.01350161, -0.02669507,
-0.06476542]
```

Note that similarity measures and coherence scores are two distinct concepts. The similarity score measures the similarity between documents in the training data. The coherence score quantifies how similar words in a topic are to each other.

## Finding documents with high similarity scores

The `sims` list refers to each document, but it does not have a label for a document. In order to assign a sequential number, we can use Python's `enumerate()` function to assign the document number. The function will produce a *t*-tuple in the form of (a document number and similarity score):

```
import pprint as pp
my_outcome = list(enumerate(sims))
```

We then sort the two tuples in the list by the similar measure (the second element) in descending order (`reverse=True`), and then we print out the top 10 news articles in the training data that are close to our query:

```
my_outcome.sort(key=lambda a: a[1], reverse=True)
```

The top 10 most relevant documents are as follows:

```
pp pprint(my_outcome[0:10])
```

The output is as follows:

```
[(93167, 0.78405493),  
(24164, 0.78052753),  
(20267, 0.7723593),  
(68823, 0.76993084),  
(5313, 0.76508605),  
(7198, 0.7607913),  
(44747, 0.758785),  
(66982, 0.74543446),  
(61143, 0.74483055),  
(24636, 0.73511076)]
```

Let's print out document '93167' and its tokens to see what it's about. This article is about price forecasts for the transportation industry in the near future:

```
train['Description'][93167]
```

Here is the output:

```
A huge jump in wholesale prices sent stocks falling yesterday as  
investors worried that rising oil prices were taking a toll on the  
overall economy.
```

```
train['Description'][24164]
```

Here is the output:

```
A sharp decline in oil prices and a surprise drop in wholesale prices  
pushed stocks higher Friday as investors; concerns about third quarter  
earnings were mitigated.
```

```
train['Description'][20267]
```

Here is the output:

```
In a bid to relieve Western economies of the unprecedented rise in  
oil prices, Saudi Arabia has slashed prices for its crude sales for  
October.
```

```
train['Description'][68823]
```

Here is the output:

US Treasury debt prices eased on Friday as a dearth of economic indicators led investors to ponder the impact of high oil prices on the economy and interest rates.

These documents all relate to our new query about the economy and its keywords, “crude prices,” “inflation,” “economy”, and “outlook earnings.”

## Summary

I hope this chapter and the previous chapter on LSI helped you gain a comprehensive understanding of LSI. In this chapter, we learned how to build an LSI model in Gensim and deploy it as a search engine. We learned how to use the coherence score to find the optimal number of topics. Then, investigated the results of the final model to ensure a sound outcome. Finally, we applied the model to score new documents.

In the following chapter, we will learn about **Latent Dirichlet Allocation (LDA)**. The techniques that we learned in this chapter will help you to learn LDA effectively.

## Questions

1. Describe the coherence score in LSI.
2. Give a short description of each of the three common ways to compute the coherence score.
3. Describe the steps to set up a model as a search engine.

## References

1. Gerry J. Elman (October 2007). “Automated Patent Examination Support - A proposal”. *Biotechnology Law Report*. 26 (5): 435–436. <https://www.liebertpub.com/doi/10.1089/blr.2007.9896>.
2. Helmers L, Horn F, Biegler F, Oppermann T, Müller KR. Automating the search for a patent’s prior art with a full text similarity search. *PLoS One*. 2019 Mar 4;14(3):e0212103. doi: 10.1371/journal.pone.0212103. PMID: 30830911; PMCID: PMC6398827.
3. Xiang Zhang, Junbo Zhao, Yann LeCun. *Character-level Convolutional Networks for Text Classification*. *Advances in Neural Information Processing Systems 28* (NIPS 2015). <https://arxiv.org/abs/1509.01626>.
4. G. M. Del Corso, A. Gulli, and F. Romani. *Ranking a stream of news*. In Proceedings of 14th International World Wide Web Conference, pages 97–106, Chiba, Japan, 2005. [http://groups.di.unipi.it/~gulli/AG\\_corpus\\_of\\_news\\_articles.html](http://groups.di.unipi.it/~gulli/AG_corpus_of_news_articles.html).
5. Röder, M., Both, A., & Hinneburg, A. (2015). *Exploring the Space of Topic Coherence Measures*. *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*. [https://en.wikipedia.org/wiki/Doi\\_\(identifier\) https://doi.org/10.1089/blr.2007.9896](https://en.wikipedia.org/wiki/Doi_(identifier) https://doi.org/10.1089/blr.2007.9896).

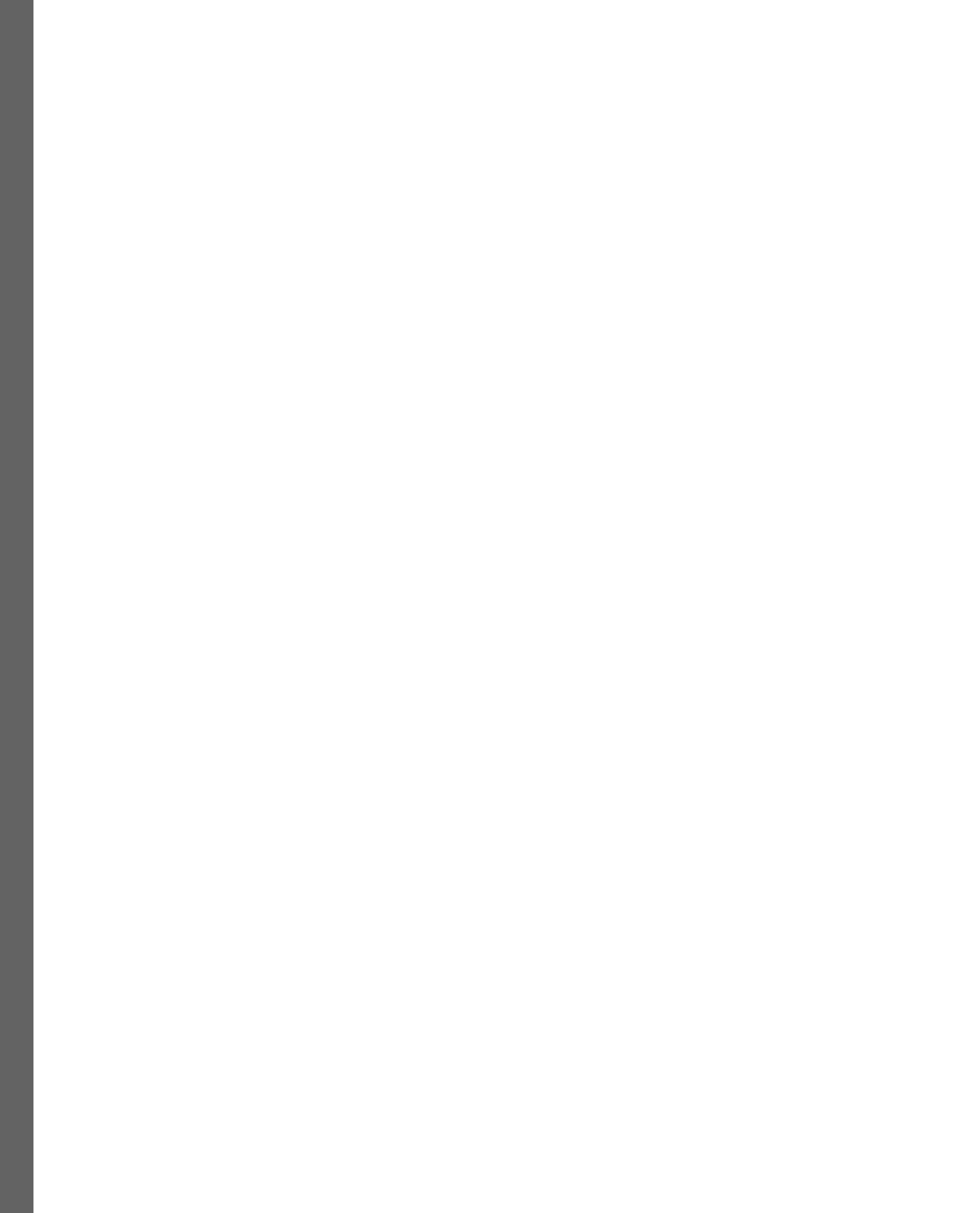
# **Part 3:**

## **Word2Vec and Doc2Vec**

Learning Word2Vec and Doc2Vec is essential for mastering NLP and text analysis. Word2Vec equips you with the ability to represent words as meaningful, dense vectors, capturing semantic relationships crucial for tasks, such as sentiment analysis and machine translation. Doc2Vec extends this capability to entire documents, enabling effective document clustering, information retrieval, and recommendation systems. These techniques not only enhance NLP model performance but also provide foundational knowledge for various applications, job opportunities, and research endeavors, making them indispensable tools in the world of language understanding and artificial intelligence.

This part contains the following chapters:

- *Chapter 7, Using Word2Vec*
- *Chapter 8, Doc2Vec with Gensim*



# 7

## Using Word2Vec

Jane is a content manager for an e-commerce website that sells a wide variety of products. Her role involves categorizing and organizing product descriptions, ensuring accurate search results, and providing personalized recommendations to customers. However, Jane faces a challenge in understanding the underlying relationships and similarities between different products based on their descriptions alone. Jane presents the challenges to the website's data scientist, Emma. Emma understands that the goals are to identify common product-related issues and relationships between different products, and discover potential areas for new products. Emma decides to use Word2Vec, a popular word embedding technique, to convert words to vectors. Word2Vec will enable Emma to measure the similarity between words, allowing search engines to return more relevant results. This is certainly an appealing feature for the e-commerce website. Word2Vec-derived embeddings can enhance document classification for sentiment analysis or even identify spam. It can even help **Named Entity Recognition (NER)** systems to identify people, places, or organizations more accurately. "Certainty there are a few use cases," Jane replied.

Now that we know the use cases of Word2Vec, in this chapter, let's discuss the rationale behind Word2Vec, the modeling structure, and how people use it. Training a Word2Vec model for millions of words in a dictionary is a time-consuming job, so there are pretrained models available for public use. I will show you how to use the pretrained models and then how to build a Word2Vec model on your own data. Further, you will learn how to visualize the outcome using two methods: the first one is by visualizing with t-SNE in Python, and the second one is by using the TensorBoard Embedding Projector.

Although in recent years there have been more powerful transformer-based LLMs for word embeddings (such as BERT word embeddings), not all developers have the resources for training and fine-tuning LLMs to specific domain data that can be prohibitively expensive. Some NLP professionals have commented that fine-tuned models may not necessarily produce more satisfactory results. Word2Vec can be a low-cost alternative that may produce satisfactory results for specific domain data.

The following learning objectives are purposely arranged in a logical order and digestible size:

- Introduction to Word2Vec
- Introduction to **skip-gram (SG)**

- Introduction to **Continuous Bag-of-Words (CBOW)**
- Using a pretrained model for semantic search
- Adding and subtracting words/concepts
- Visualizing Word2Vec with Tensor Board
- Training your own Word2Vec model in CBOW and SG
- Visualizing your Word2Vec model with t-SNE
- Comparing Word2Vec with Doc2Vec, GloVe, and FastText

By the end of this chapter, you will have an understanding of Word2Vec evolution and its algorithm. You will know how to build and visualize your own Word2Vec models.

## Technical requirements

We will use `gensim` to build the Word2Vec models and use `TSNE` from `sklearn` to visualize the outcome:

```
from sklearn.manifold import TSNE
import gensim
from gensim.models import Word2Vec, KeyedVectors
```

You can find the code files for this chapter on GitHub at <https://github.com/PacktPublishing/The-Handbook-of-NLP-with-Gensim/tree/main/Chapter07>.

## Introduction to Word2Vec

In NLP development, an important discovery is the **distributional hypothesis**. It states that words that occur in similar contexts tend to have similar meanings. For example, the words *cat* and *dog*, *temple* and *monk*, or *king* and *queen* are sometimes seen together. In contrast, the words *iron* and *monk*, or *car* and *sky* appear less often in the same contexts. If words are semantically similar, they tend to show up in similar contexts and with similar distributions. The distributional hypothesis received an interesting comment from the linguist J. R. Firth in the 1950s: “You shall know a word by the company it keeps” [1]. This became the theoretical foundation for many computational models of word meaning and word representation.

The distributional hypothesis paves the way for the quantification of word similarities. In 2013, a Google team led by Tomas Mikolov published two milestone papers for Word2Vec and Doc2Vec [2] [3]. A word can be represented by other words in a vector of numerical values. Word2Vec captures similarities and dissimilarities between words. In the previous example, the word vectors of “cat” and “dog” are probably close. In contrast, the word vectors of “iron” and “monk” are relatively far from each other. This is a fantastic observation. *Figure 7.1* shows words that are close to the word “iron” include “gunpowder,” “metals,” and “steel.” Words far from “iron” are “organic,” “sugar,” and “grain.”

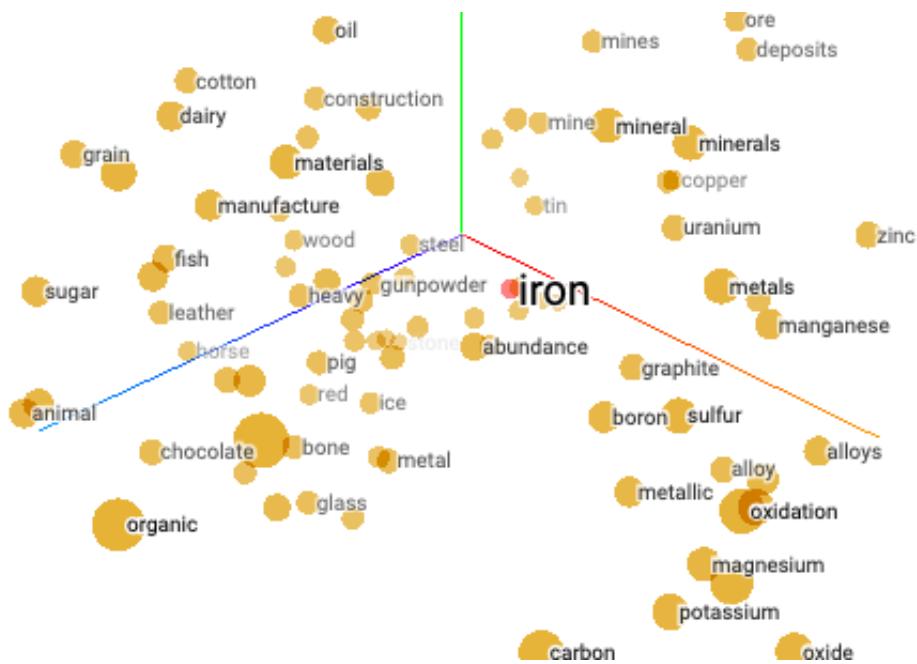


Figure 7.1 – An overview of word similarities for “iron”

Before Word2Vec, the prevailing techniques were BoW and TF-IDF. Since Word2Vec was a milestone in NLP’s history, let’s see its advantages over BoW and TF-IDF.

## Advantages of Word2Vec

Word2Vec captures the **semantic relationships** between words by representing them as vectors in a continuous space. Words with similar meanings or related concepts tend to have vectors that are close together. This enables us to find similar words or identify related terms.

Word2Vec provides **dimensionality reduction**. BoW and TF-IDF tend to create very large and sparse matrices. Word2Vec presents the high-dimensional space of words in low-dimensional word vectors. This dimensionality reduction allows more efficient storage and processing of word representations, making it computationally feasible to work with large vocabularies.

Because vectors can be added or subtracted mathematically and plotted in a multi-dimensional space, Word2Vec enables us to **measure and visualize the similarities or dissimilarities** of words.

Word2Vec training is efficient and fast compared to more complex models, such as transformer-based LLMs. The SG and CBOW architectures that we will learn about in the next section are computationally efficient and can be trained on large text corpora in a reasonable amount of time.

It's important to note that Word2Vec also has limitations. For example, it may struggle with out-of-vocabulary words, it doesn't capture context beyond the local window, and it may not handle rare or infrequent words effectively. However, despite these limitations, Word2Vec has proven to be a valuable tool in many NLP applications, thanks to its ability to capture semantic relationships and provide efficient and interpretable word embeddings.

Other than its applications for word similarities, Word2Vec has been used in more real-world applications.

## Reviewing the real-world applications of Word2Vec

Word2Vec can suggest words that are semantically related. It therefore can serve as an e-commerce recommendation system to recommend products to customers. Word2Vec identifies the relevant items based on the purchase histories of customers. For example, suppose it is the back-to-school season and a student has ordered a curved monitor, a keyboard, a mouse, adapters, and a headset in the past few days. What are the recommendations that they may see? Imagine these items are in a vector space, like *Figure 7.1*. What items are close to these items in the vector space? The recommendation system may suggest a desk lamp.

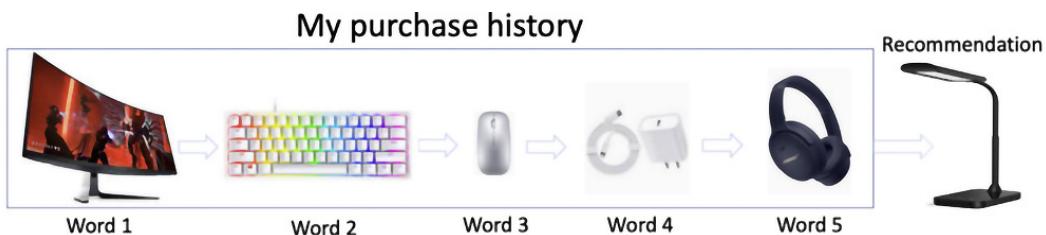


Figure 7.2 – A hypothetical example of purchase history

Can the same idea be applied to movie recommendations? Yes. Here are the movie recommendations for me based on my recent watch history:

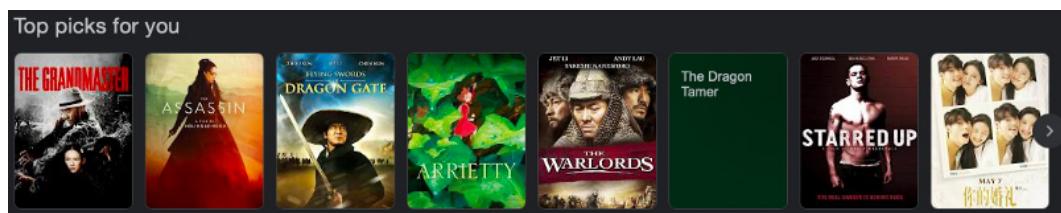


Figure 7.3 – An overview of a hypothetical example of movie watch history

Now that we know the applications of Word2Vec, let's discuss its modeling framework. The Word2Vec model has two techniques: the **Skip-Gram (SG)** model and the **Continuous Bag-of-Words (CBOW)** model. These two techniques are just the reverse neural networks of each other.

## Introduction to Skip-Gram (SG)

In the SG architecture, the objective is to predict the context words given a target word. The training process involves sliding a fixed-size window over the text corpus and generating training examples for each word in the corpus. The target word is selected, and the context words within the window are treated as positive training examples. The SG model aims at maximizing the probability of predicting the context words given the target word.

The SG model is a simple neural network. It has an input layer, a hidden layer, and an output layer. We are not interested in the output layer or the model's architecture, but we are interested in the weights of the hidden layer. The weights become the word embeddings or word vectors. *Figure 7.4* shows an SG neural network.  $w(t)$  in the input layer is the word to be converted to a vector.  $w(t-2)$  and  $w(t-1)$  in the output layer are the two words before  $w(t)$ , and  $w(t+1)$  and  $w(t+2)$  are the two words after  $w(t)$ . Here, we specify a window of just two words. If you specify a window of three words, the output layer will have six neurons for the three words before the input word and the three words after the input word.

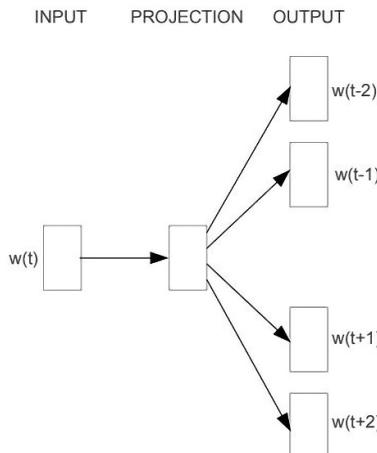


Figure 7.4 – SG model

This neural network model is a classical supervised learning model. We will organize the data into the input and output format to feed into the model. Many textbooks do not explain it clearly and may become an obstacle to learning. So, let's do it step by step.

## Data preparation

The SG model requires word pairs in (input word, output word) format. The input word is the one that we want to embed into numerical values. The output words are the words before and after the input word. Let's take a sentence:

*"Jupiter overtakes Saturn as the planet with the most known moons."*

This sentence will be organized into (input word, output word) format to be fed to the neural network model. Suppose we are going to use two words before and after the output word, as in *Figure 7.4*; the word pairs shall be organized as shown in *Figure 7.5*. In the first instance, the input word is “overtakes.” There is only one word, “Jupiter,” before “overtakes.” So, the first pair is (overtakes, Jupiter). And the next two word pairs are (overtakes, Saturn) and (overtakes, as).

	Source Text	Word Pair
1	Jupiter overtakes Saturn as the planet with the most known moons.	(overtakes, Jupiter) (overtakes, Saturn) (overtakes, as)
2	Jupiter overtakes Saturn as the planet with the most known moons.	(Saturn, Jupiter) (Saturn, overtakes) (Saturn, as) (Saturn, the)
3	Jupiter overtakes Saturn as the planet with the most known moons.	(as, overtakes) (as, Saturn) (as, the) (as, planet)
4	Jupiter overtakes Saturn as the planet with the most known moons.	(the, Saturn) (the, as) (the, planet) (the, with)
5	Jupiter overtakes Saturn as the planet with the most known moons.	(planet, as) (planet, the) (planet, with) (planet, the)
6	Jupiter overtakes Saturn as the planet with the most known moons.	(with, the) (with, planet) (with, the) (with, most)

Figure 7.5 – An overview of data preparation for SG

Notice that the word pairs are not any two random words but the adjacent words to the central word. This arrangement captures the idea that if words are semantically similar, they tend to show up in similar contexts and with similar distributions, which is what we learned as the distributional hypothesis. When we insert the instances in the neural network, they will look like *Figure 7.6*:

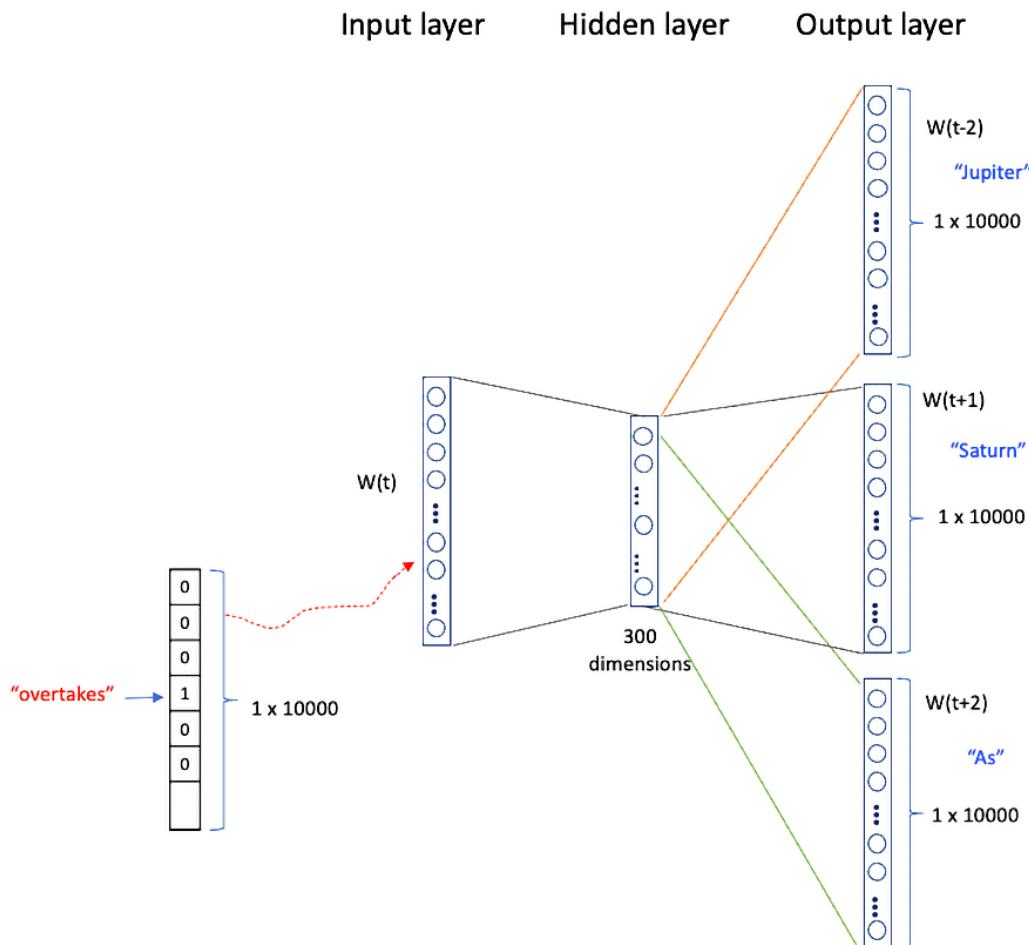


Figure 7.6 – The structure of an SG model

Let me explain the components of the model in *Figure 7.6*.

### The input and output layers

Assume there are 10,000 unique words in the input texts. With one-hot encoding, each word becomes a vector with 10,000 dimensions. In our case, the vector for the word "overtakes" is a  $1 \times 10,000$  vector in which the position of "overtakes" is 1 and all other words are 0. Likewise, the word "Jupiter" is a  $1 \times 10,000$  vector in which the position for "Jupiter" is 1 and all other words are 0. With that, the word pair (overtakes, Jupiter) has the matching dimensions. The input vector is a  $1 \times 10,000$  vector, and the output vector is also a  $1 \times 10,000$  vector.

## The hidden layer

The weights in the hidden layer become the word embeddings. In *Figure 7.4*, there is a hidden layer with 300 neurons. Between the input layer and the hidden layer is a 10,000 x 300 weight matrix. Each node in the hidden layer is a weighted sum of the input nodes. Each word in the input layer will be represented by a vector of 300 dimensions (300 words). Notice that most of the input nodes are zeros; the weights coming from the non-zero input nodes are the ones contributing to the hidden layer. You may ask why there are 300 neurons in the hidden layer. This can be controlled by a hyperparameter. Google's pretrained Word2Vec model sets the number of neurons in the hidden layer at 300 nodes. So, I use 300 nodes for the hidden layer in this example.

## Should I remove stop words for training Word2Vec?

In TF-IDF, we've learned we can improve the prediction accuracy by removing the stop words. Should we remove stop words for Word2Vec? The short answer is "not necessarily". Word2Vec uses adjacent words in the word window. Stop words in the word window can provide some context. Therefore, removing stop words may or may not help the prediction accuracy.

Let's continue with the next Word2Vec technique.

## Model computation

The SG model is designed to train the hidden weights (which become word embeddings) by learning to predict context words from target words (or vice versa) within a given window of text. It uses a **naive** (or **standard**) **softmax** approach. The loss function computes the dissimilarity between the predicted probabilities and the actual words. It aims to minimize this dissimilarity by adjusting the weights during model training. Mathematically, this loss function is expressed as the negative log-likelihood of the observed context words under the model's predicted probabilities.

Let me talk about the naive softmax approach in NLP. It is used in neural networks to compute a probability distribution over a set of classes or categories. In NLP, each word is a class, and the goal is to calculate the probability of each class being the correct one. This is done by exponentiating and normalizing the dot product of the input context vector and the weight vectors for all classes.

The naive softmax computation is effective for small to moderately sized vocabularies. But it will become exponentially more expensive as the vocabulary size grows, making it impractical for a large volume of vocabularies. To address this issue, more efficient techniques such as **negative sampling** have been developed. Instead of trying to predict the correct output (context) for every word in a large vocabulary, negative sampling focuses on a smaller, randomly chosen subset of negative examples (words not in the context) for each positive example (words in the context). During training, the model is presented with pairs of words, with some being positive examples and others being negative examples. The model's objective is to correctly classify positive pairs as real and negative pairs as fake. By sampling a limited number of negative examples per positive example, negative sampling significantly reduces the computational burden while still allowing the model to learn meaningful word embeddings that

capture semantic relationships between words. This technique has become instrumental in training word embeddings for large vocabularies and corpora.

We have learned about the SG model structure, now let's learn another variant – CBOW.

## Introduction to CBOW

The neural network of CBOW is shown in *Figure 7.7*. It looks like the mirror image of SG. The input layer consists of words that are adjacent to the target words. Again, we are interested in the weights in the hidden layer. They will be the word embeddings.

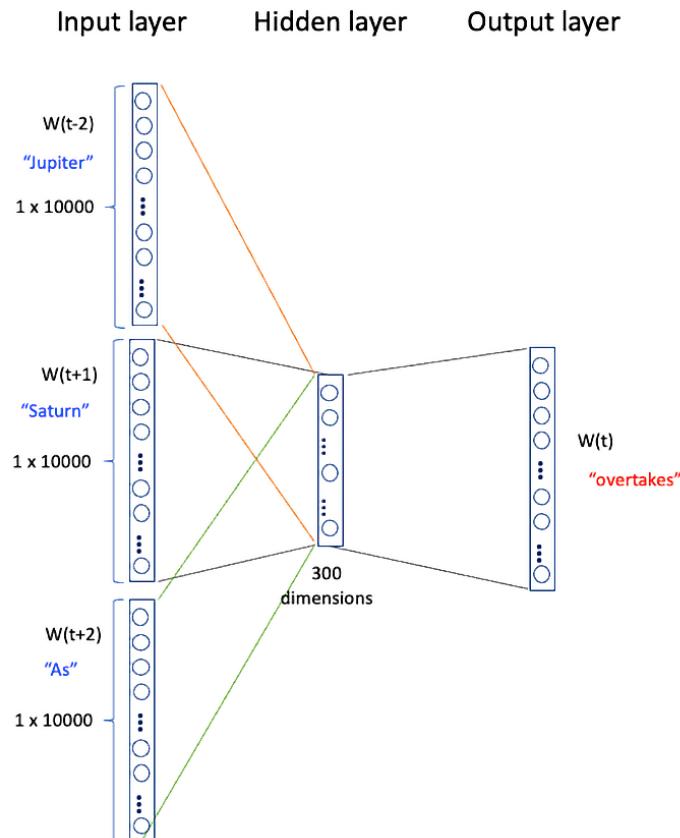


Figure 7.7 – The structure of a CBOW model

The word pairs of the input words and the output words become the pairs as shown in *Figure 7.8*. They are just the reverse of the word pairs in *Figure 7.5*. The structure of the neural network is reversed too. Between the hidden layer and the output layer is a  $300 \times 10,000$  weight matrix. This weight matrix is what we are interested in because it has the vector encodings of all the unique words. If we inspect

carefully, we will see most of the input nodes are zeros; the weights coming from the non-zero input nodes are the ones contributing to the hidden layer. The  $i^{\text{th}}$  row in the weight matrix is the weight for the  $i^{\text{th}}$  word.

	Source Text	Word Pair
1	Jupiter overtakes Saturn as the planet with the most known moons.	(Jupiter, overtakes) (Saturn, overtakes) (as, overtakes)
2	Jupiter overtakes Saturn as the planet with the most known moons.	(Jupiter, Saturn) (overtakes, Saturn) (as, Saturn) (the, Saturn)
3	Jupiter overtakes Saturn as the planet with the most known moons.	(overtakes, as) (Saturn, as) (the, as) (planet, as)
4	Jupiter overtakes Saturn as the planet with the most known moons.	(Saturn, the) (as, the) (planet, the) (with, the)
5	Jupiter overtakes Saturn as the planet with the most known moons.	(as, planet) (the, planet) (with, planet) (the, planet)
6	Jupiter overtakes Saturn as the planet with the most known moons.	(the, with) (planet, with) (the, with) (most, with)

Figure 7.8 – An overview of data preparation for a CBOW model

There are many pretrained Word2Vec models. Now, let's use Gensim to load and use them.

## Using a pretrained model for semantic search

Gensim is probably the most popular library for Word2Vec modeling. A famous pretrained Word2Vec model is the Google Word2Vec model. The model can represent 100 billion words. Each word is represented by a vector of 300 dimensions (300 words). This pretrained model has been released to the public. It can be downloaded from the Google Word2Vec Archive (<https://code.google.com/archive/p/word2vec/>) or Kaggle (<https://www.kaggle.com/datasets/leadbest/googlenewsvectorsnegative300>).

Gensim has developed a standalone module called `KeyedVectors` that can query any word vectors built by different models, such as `Word2Vec` or `FastText`. Let's import the module:

```
import gensim  
from gensim.models import Word2Vec, KeyedVectors
```

I use the `load_word2vec_format()` class to load the Google News `Word2Vec` model. The `limit = 10000` parameter sets the maximum number of word vectors to 10,000. If we do not specify it, the default will load all word vectors:

```
folder = '/content/gdrive/My Drive/data/'  
data_file = 'GoogleNews-vectors-negative300.bin'  
model = KeyedVectors.load_word2vec_format(folder +  
    data_file, binary=True, limit=100000)
```

Let's test the word vector for `car`. It will print out a vector of 300 dimensions:

```
vec = model['car']  
print(vec)
```

The output is as follows:

```
[ 0.13085938  0.00842285 ... -0.15039062]
```

We have mentioned `Word2Vec` can show the similarities of words. Let's show them. You do not need to remember too many functions of `Word2Vec`. The most frequently used function is `most_similar()`, which can find the top N most similar words. Let's find similar words to `car`:

```
model.most_similar('car')
```

The outputs are as follows:

```
[('vehicle',  0.7821096181869507),  
 ('cars',  0.7423830032348633),  
 ('SUV',  0.7160962820053101),  
 ('minivan',  0.6907036304473877),  
 ('truck',  0.6735789775848389),  
 ('Car',  0.6677608489990234),  
 ('Ford_Focus',  0.667320191860199),  
 ('Honda_Civic',  0.662684977054596),  
 ('Jeep',  0.6511331796646118),  
 ('pickup_truck',  0.64414381980896)]
```

The previous outputs appear satisfactory. Notice the words `car`, `Car`, and `cars` are three distinct words. The Google News Word2Vec model did not preprocess the texts to their stem form, as we discussed in *Chapter 3, Text Wrangling and Preprocessing*. What will happen if we apply lemmatization first, then do Word2Vec modeling? In that case, the three words `car`, `Car`, and `cars` will become `car`.

Let's examine another word, `Ford`:

```
model.most_similar('Ford')
```

The outputs are as follows:

```
[('Ford_Motor_Co.', 0.6826023459434509),  
 ('Chrysler', 0.6106929183006287),  
 ('Ford_Motor', 0.6086850762367249),  
 ('Toyota', 0.5959083437919617),  
 ('Nissan', 0.5914361476898193),  
 ('Fords', 0.5778011083602905),  
 ('automaker', 0.5739781260490417),  
 ('General_Motors', 0.5679431557655334),  
 ('Ford_Motor_Co', 0.5612248182296753),  
 ('GM', 0.5596591234207153)]
```

How about the car brand `Tesla`?

```
model.most_similar('Tesla')
```

The outputs are other modern or **electronic vehicles (EVs)**:

```
[('Tesla_Motors', 0.720951497554779),  
 ('Volt', 0.5760279893875122),  
 ('Fisker', 0.5558066368103027),  
 ('Chevy_Volt', 0.5387632846832275),  
 ('Roadster', 0.5383760929107666),  
 ('Chevrolet_Volt', 0.5310286283493042),  
 ('Prius', 0.5102267861366272),  
 ('Musk', 0.5088856220245361),  
 ('EVs', 0.49960029125213623),  
 ('Nissan_Leaf', 0.49743831157684326)]
```



Figure 7.9 – Car model images

How about the Korean car brand Kia? It returns the Korean brand Hyundai and the Japanese brands Nissan and Mazda:

```
model.most_similar('Kia')
The outputs are:
[('Hyundai', 0.7287631034851074),
 ('Kia_Motors', 0.6962010860443115),
 ('Nissan', 0.6836174726486206),
 ('Mazda', 0.6306065320968628),
 ('Cruze', 0.6188750863075256),
 ('Passat', 0.6150655746459961),
 ('Lexus', 0.6079877614974976),
 ('GM_Daewoo', 0.6036292314529419),
 ('Toyota', 0.594830334186554),
 ('Camry', 0.594078779220581)]
```

We have mentioned the words can be added or subtracted. Let's see how it works.

## Adding and subtracting words/concepts

Our brain understands the semantic relationships between words. For example, if you heard the riddle “What is the female version of a king?”, you know the answer is “Queen.” We are thinking “as male is to a king, so a female is to a queen.” We can even write a simple antonymic expression “king–male + female = queen.”

A fantastic use case of word vectors is that words can be added or subtracted arithmetically. The addition of two words means the combination of two concepts. It will return a new vector that represents the combination. If you subtract a word/concept from another word/concept, it means you want more of the previous concept and less of the second concept.

## Example 1

Let's apply this concept to our car world. A minivan can hold more people than a small sedan. What types of vehicles are similar to a minivan but not a sedan? Maybe a bus or a truck, which can hold more people than a sedan. Next, we do "minivan—sedan". We get tractor, bus, and dump truck—not too bad:

```
vec = model['minivan'] - model['sedan']
model.most_similar([vec])
```

The outputs are as follows:

```
[('tractor_trailer', 0.29951590299606323),
 ('bus', 0.2529987692832947),
 ('dump_truck', 0.24461477994918823),
 ('van', 0.2445259392261505),
 ('truck', 0.24246448278427124),
 ('minivan', 0.23668046295642853),
 ('MINNEAPOLIS', 0.2338685542345047),
 ('child_endangerment', 0.23220384120941162),
 ('daycare', 0.2305922508239746),
 ('Greyhound_bus', 0.2305399775505066)]
```

## Example 2

If we hear the names of high-end cars such as Ferrari or Porsche, what other cars will we think of? We probably will think of BMW or Corvette. In the next code, we add Ferrari and Porsche together. It returns BMW or Corvette, but not any low-price cars. Again, not bad:

```
vec = model['Ferraris'] + model['Porsches']
model.most_similar([vec])
```

The results are other high-end vehicles:

```
[('Porsches', 0.9251331090927124),
 ('Ferraris', 0.9171532988548279),
 ('BMW', 0.7080749273300171),
 ('Corvettes', 0.6783405542373657),
 ('Ferrari', 0.6690179705619812),
 ('cars', 0.6335420608520508),
 ('Porsche', 0.6244776248931885),
 ('Aston_Martin', 0.6212916374206543),
 ('Maserati', 0.6196482181549072),
 ('Hondas', 0.6056317090988159)]
```



Figure 7.10 – Car model images

You may ask whether there are more pretrained models other than the Google News Word2Vec model. For those of you who want to find more pretrained models, Gensim has more pretrained models at <https://github.com/RaRe-Technologies/gensim-data>.

The previous examples show words can be expressed as vectors. It will be even better if we can demonstrate the vectors in a 3D space, as shown in *Figure 7.1*. I will show you how to use the TensorBoard to do that.

## Visualizing Word2Vec with TensorBoard

Let me introduce a great app—the Tensor Board Embedding Projector (<https://projector.tensorflow.org/>). It can graphically represent high-dimensional word embeddings. It lets you choose UMAP, T-SNE, PCA, and other dimensional-reduction techniques. In the right panel, you can enter a word. It can graphically represent high-dimensional word embeddings. It lets you choose UMAP, T-SNE, PCA, and other dimensional-reduction techniques. In the right panel, you can enter a word.



Figure 7.11 – An overview of TensorBoard

I entered the word `building`. It returns related words, such as `construction`, `houses`, `constructed`, `built`, and so on:



Figure 7.12 – An overview of the right panel of TensorBoard

The values in *Figure 7.12* are the cosine similarities between the vector of `building` and the vectors of the other words. Remember, cosine similarity measures the similarity between two vectors. It is a value between -1 and 1. A value of 1 indicates that the vectors are perfectly similar and point in the same direction, while a value of -1 indicates perfect dissimilarity, with the vectors pointing in exactly opposite directions. A value of 0 indicates orthogonality, meaning the vectors are unrelated or have no similarity.

This pretrained Word2Vec model shows its great capability. On many occasions, you have proprietary data and need to build your own model. Let's learn how to build a Word2Vec model.

## Training your own Word2Vec model in CBOW and Skip-Gram

I will show you how to build the model step by step. The steps are as follows:

1. Load the data.
2. Text preprocessing.
3. Modeling.
4. Save and load the model.
5. Use the model.

The text preprocessing only consists of tokenization. In *Figure 7.5* and *7.8*, we have explained how text data is prepared to be the inputs and outputs. Gensim has taken care of this data preparation. All we need to do is text tokenization.

### Load the data

Let's continue to use the same AG's news articles in order to save your learning time. This dataset has 120,000 news articles in the training dataset. Let's load it:

```
import pandas as pd
import numpy as np
pd.set_option('display.max_colwidth', -1)
path = "/content/gdrive/My Drive/data/gensim"
train = pd.read_csv(path + "/ag_news_train.csv")
```

### Text preprocessing

Let's apply the same preprocessing procedure. The tokenized outcome, which is a list of lists, is what we will be using to build the model:

```
from gensim.parsing.preprocessing import preprocess_string
text_tokenized = []
for doc in train['Description']:
    k = preprocess_string(doc)
    text_tokenized.append(k)
```

The list of lists looks like this:

```
[['reuter', 'short', 'seller', 'wall', 'street', 'dwindl', 'band',
'ultra', 'cynic', 'see', 'green'],
['reuter', 'privat', 'invest', 'firm', 'carlyl', 'group', 'reput',
'make', 'time', 'occasion', 'controversi', 'plai', 'defens',
```

```
'industri', 'quietli', 'place', 'bet', 'market'],
['reuter', 'soar', 'crude', 'price', 'plu', 'worri', 'eonomi',
'outlook', 'earn', 'expect', 'hang', 'stock', 'market', 'week',
'depth', 'summer', 'doldrum']]
```

## Training your own Word2Vec model in CBOW

Gensim made it very simple to build a Word2Vec model with only one line of code. Let's build a CBOW Word2Vec model:

```
import pandas as pd
import gensim
from gensim.models import Word2Vec, KeyedVectors
cbow_model = Word2Vec(sentences=text_tokenized, sg=0,
min_count=10, window =3)
```

Let's review the parameters:

- **sentences**: This is a list of lists of tokens.
- **sg**: This hyperparameter is an important one. 1 is for SG and 0 is for CBOW. The default value is 0.
- **min\_count**: This is a threshold beyond which a word will be ignored if its count is less than this minimum count.
- **window**: This is the maximum distance between the current and predicted word within a sentence. The default value is 5.

### **Saving and loading the CBOW model**

Now the model is complete. We will save it in a local directory for future use:

```
cbow_model.save(path + "/cbow.model")
```

Sometime later when you need to use the model, you just need to load it:

```
Loaded_cbow = Word2Vec.load(path + "/cbow.model")
```

### **Using the CBOW model**

Now that the model is complete, we can use it to find word similarities. Our dataset is a news headline dataset that has business terms such as "stock market." Let's test the business word `stock` for the stock market:

```
Loaded_cbow.wv.most_similar("stock")
```

The outputs are as follows:

```
[('share', 0.7349613904953003),  
 ('currenc', 0.6588786244392395),  
 ('dollar', 0.6058856248855591),  
 ('yen', 0.6014151573181152),  
 ('slightli', 0.6013827323913574),  
 ('broader', 0.5927409529685974),  
 ('index', 0.5899143218994141),  
 ('mercantil', 0.5821986794471741),  
 ('seng', 0.577268123626709),  
 ('ipo', 0.5656859874725342)]
```

Let's test another word, song. Interestingly, it returns download, iTunes (itun), and so on. These are words we often read in news headlines:

```
Loaded_cbowodel.wv.most_similar("song")
```

The outputs are as follows:

```
[('download', 0.8578492403030396),  
 ('copi', 0.801600456237793),  
 ('tune', 0.7841229438781738),  
 ('music', 0.777767539024353),  
 ('itun', 0.7776855230331421),  
 ('movi', 0.7719132304191589),  
 ('jukebox', 0.7421693801879883),  
 ('gift', 0.736247181892395),  
 ('realmetwork', 0.7360978126525879),  
 ('album', 0.7328618168830872)]
```

This process has built a CBOW model. Similarly, let's build an SG model.

## Training your own Word2Vec model in Skip-Gram

It is straightforward to build an SG model. You just need to assign `sg=1`. Let's see this in the following code:

```
sg_model = Word2Vec(sentences=text_tokenized, sg=1,  
 min_count=10, window =3)
```

That's it. The SG model is completed.

### Using the Skip-Gram model

Let's print out the most similar words to song. It returns download, millionth, album, and music:

```
sg_model.wv.most_similar("song")
```

The outputs are as follows:

```
[('download', 0.7793432474136353),  
 ('millionth', 0.7647113800048828),  
 ('album', 0.7639385461807251),  
 ('music', 0.7579527497291565),  
 ('itun', 0.7550144195556641),  
 ('realmetwork', 0.7388608455657959),  
 ('jukebox', 0.7385604977607727),  
 ('xeni', 0.7265862822532654),  
 ('bootleg', 0.7262958884239197),  
 ('megastor', 0.7138321399688721)]
```

You have seen that building a Word2Vec model is straightforward, it is important to be able to display the results. I will introduce the t-SNE technique and the TensorBoard to help you do data visualization.

## Visualizing your Word2Vec model with t-SNE

When we attempt to visualize a high-dimensional vector to a 2D plot, we have to reduce the dimensions first. The most popular dimension reduction technique probably is **Principal Component Analysis (PCA)**. However, PCA has limitations, as I have outlined in the article *Dimension reduction with Python* (<https://towardsdatascience.com/dimension-reduction-techniques-with-python-f36ca7009e5c>) [3]. In this section, I will give a brief introduction to t-SNE and use it to visualize our model.

t-SNE is the abbreviation for **t-distributed Stochastic Neighbor Embedding**. It was developed by Laurens van der Maaten and Geoffrey Hinton in their paper [4]. It is a dimensionality reduction technique used for visualizing high-dimensional data in a low-dimensional space. It preserves the local structure of the data while revealing the underlying global patterns. Let's see the graph about the Swiss roll in *Figure 7.13*. If we just simply collapse the roll, it will mess up the roll. It will be better if we can "unroll" the Swiss roll into a flat cake. This means to preserve the local structure to unroll it. In mathematics, it means the relationships between adjacent points will be preserved.

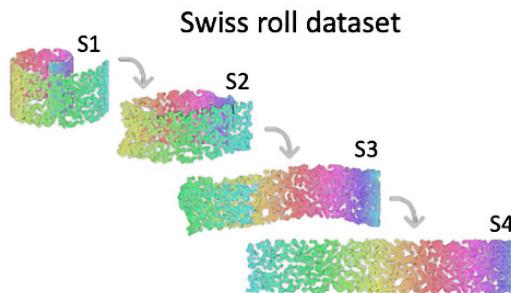


Figure 7.13 – t-SNE for a Swiss roll

Let's explain t-SNE with another example. *Figure 7.14* shows a 3-dimensional tetrahedron. A simple way is to collapse the 3-dimensional graph to a 2-dimensional graph like Panel (A). It may not work well because Cluster A may collapse onto Cluster B and mess up the presentation. t-SNE will convert it into Panel (B), which is probably a better presentation. It preserves the far distances between Cluster (A)-(E) while keeping the local distances of points in each cluster.

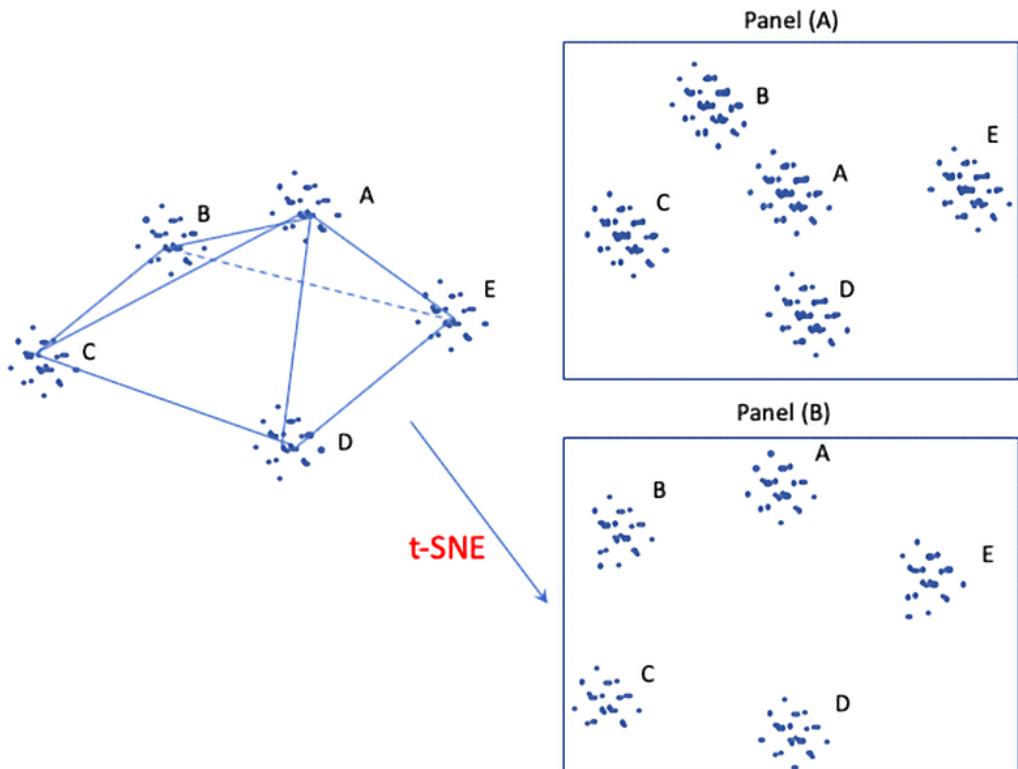


Figure 7.14 – t-SNE for a tetrahedron

Now let's load the t-SNE library in Python:

```
from sklearn.manifold import TSNE
import re
import matplotlib.pyplot as plt
```

We want to annotate our graph with words. Let's extract the words in a list called `vocab`:

```
vocab = list(cbow_model.wv.vocab)
vocab
```

Let's get the list of vectors of the words and call it `X`:

```
X = cbow_model[cbow_model.wv.vocab]
print(X)
```

The outputs are as follows:

```
[[ 1.1559312 , 0.5244283 , 0.41477892, ...,-0.07753743, 1.4355767 ,
-0.21487173], [ 0.33487827, -0.4401077 , -0.12471292, ...,-0.9153409
, -1.0702105 , 0.23701574], [ 0.1377047 , -0.26083317, 0.2421679 , ...,
-0.09389343, -0.17376275, 0.31428534], ...,
```

The following code declares the t-SNE model, then fits the model to our data `X`. The default of `n_components` is 2 because it will make a 2-dimensional result:

```
tsne = TSNE(n_components=2)
X_tsne = tsne.fit_transform(X)
```

Now we are ready to plot the result, `X_tsne`. I plot the first 100 words with `len = 100`. I first convert `X_tsne` to a pandas DataFrame, `df`. The code `ax.scatter()` plots words in dots. Then we annotate each word:

```
len = 100
df = pd.DataFrame(X_tsne[0:len], index=vocab[0:len],
columns=['x', 'y'])
fig = plt.figure(figsize=(8,5))
ax = fig.add_subplot(1, 1, 1)
ax.scatter(df['x'], df['y'])
for word, pos in df.iterrows():
    ax.annotate(word, pos)
```

The output looks like this:

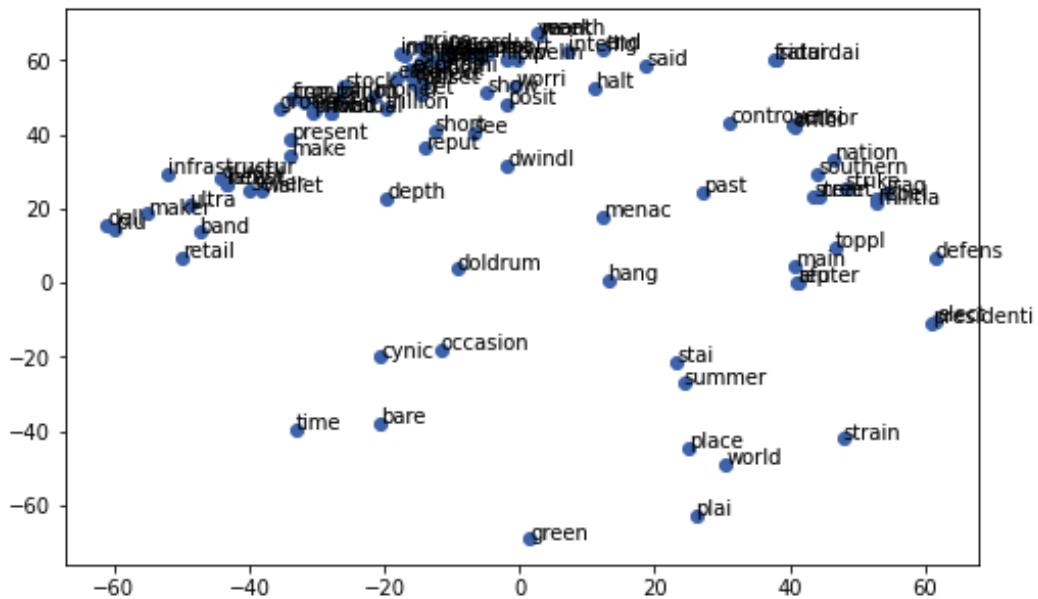


Figure 7.15 – An overview of t-SNE visualization

Knowing the tools to visualize Word2Vec really helps with its explainability. Let's continue to review a few related techniques.

## Comparing Word2Vec with Doc2Vec, GloVe, and fastText

There are a few similar techniques to Word2Vec, including Doc2Vec, GloVec, and fastText. As we are close to the end of the chapter, let's spend some time discussing them and comparing the differences.

### Word2Vec versus Doc2Vec

Both Word2Vec and Doc2Vec are based on the distributional hypothesis. While Word2Vec focuses on learning vector representations for individual words, Doc2Vec extends Word2Vec to learn vector representations for entire documents or paragraphs. In terms of the modeling approach (<https://arxiv.org/abs/1310.4546>), Word2Vec takes a sequence of words as input and learns word embeddings. Doc2Vec takes a sequence of words along with an additional document ID (or label) as input. It learns document embeddings by predicting words in the context of the document ID. The document ID acts as an additional input signal, helping the model to differentiate between different documents. We will learn about Doc2Vec in the next chapter.

## Word2Vec versus GloVe

A popular word-embedding method is “GloVe” (Global Vectors for Word Representation). It was developed in 2014 by the NLP team at Stanford University. The GloVe model also represents a word by other words that are semantically similar in a vector. The model’s name, **GloVe** stands for **Global Vectors**. It was trained on a large Twitter dataset with 2 billion tweets and 1.2 million vocabularies. Because Word2Vec and GloVe are similar word embedding techniques, Gensim doesn’t provide the GloVe algorithm.

GloVe focuses on capturing the global statistical properties of word co-occurrence patterns in a corpus. It constructs a co-occurrence matrix based on word co-occurrence statistics, and then factorizes this matrix to learn the word embeddings. The modeling technique of GloVe is different from Word2Vec. GloVe is an unsupervised learning approach, while Word2Vec is supervised learning, as mentioned earlier.

## Word2Vec versus FastText

FastText was open sourced by **Facebook’s AI Research (FAIR)** lab. It is a library that learns text representations and text classifiers. When it was released, it emphasized its fast-training speed, as its name suggests. It can model “on more than 1 billion words in less than 10 minutes using a standard multicore CPU” [8]. It uses a classification system to place words into categories and sub-categories but doesn’t try to understand words like Word2Vec. This Gensim notebook ([https://github.com/RaRe-Technologies/gensim/blob/develop/docs/notebooks/Word2Vec\\_FastText\\_Comparison.ipynb](https://github.com/RaRe-Technologies/gensim/blob/develop/docs/notebooks/Word2Vec_FastText_Comparison.ipynb)) performs a comparison between Word2Vec and FastText.

Let’s understand the differences between them. Word2Vec operates at the word level. It learns embeddings for complete words but does not handle **out-of-vocabulary (OOV)** words not present in the training data. FastText extends Word2Vec by considering subword information. It breaks words down into character n-grams (subword units), such as prefixes and suffixes, and learns embeddings for these subword units in addition to complete words. This enables FastText to handle OOV words and capture information from morphologically related words.

## Summary

I hope this chapter provided systematic guidance for you to understand Word2Vec. In this chapter, we started with the distributional hypothesis, which says if words are semantically similar, they tend to show up in similar contexts and with similar distributions. Word2Vec is almost the quantification of the distributional hypothesis. Word2Vec captures the similarities of words/concepts in vector form. Because vectors imply a measure of distance, Word2Vec enables us to measure the similarities of words or concepts.

---

We also learned the advantages of Word2Vec over **Bag of Words (BOW)** and **Term Frequency-Inverse Document Frequency (TF-IDF)**. Word2Vec can also capture the compositional relationships between words. Word2Vec can reduce dimensionality by presenting the high-dimensional space of words in lower-dimensional word vectors. We are also informed that Word2Vec has been applied in many real-world recommendation systems.

We learned how to use a pretrained Word2Vec model and visualize it with TensorBoard. Then we moved on to modeling with our own data. We learned that the Word2Vec model has two techniques: the **Continuous Bag-of-Words (CBOW)** model and the **Skip-Gram (SG)** model. The two techniques are just the reverse neural networks of each other. We learned how to build Word2Vec models and visualize word vectors with t-SNE. Finally, we compared the differences between Word2Vec, Doc2Vec, GloVe, and FastText.

I trust this chapter prepared you for success in Word2Vec modeling. In the next chapter, we will learn about a related modeling technique – Doc2Vec.

## Questions

1. What is the distributional hypothesis?
2. What are the advantages of Word2Vec?
3. Describe some real-world applications of Word2Vec.
4. Describe the two modeling techniques of Word2Vec.
5. Describe the architectures of the Skip-Gram and the Continuous Bag-of-Words models.
6. Describe the t-SNE technique.

## References

1. John Rupert Firth—Wikipedia ([https://en.wikipedia.org/wiki/John\\_Rupert\\_Firth](https://en.wikipedia.org/wiki/John_Rupert_Firth))
2. Mikolov, T., Chen, K., Corrado, G.S., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *International Conference on Learning Representations*. <https://arxiv.org/abs/1310.4546>
3. Kuo, Chris, “Dimension Reduction with Python”, <https://towardsdatascience.com/dimension-reduction-techniques-with-python-f36ca7009e5c>
4. Maaten, L.V., & Hinton, G.E. (2008). Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9, 2579–2605.
5. Le, Q.V., & Mikolov, T. (2014). Distributed Representations of Sentences and Documents. *International Conference on Machine Learning*. <https://arxiv.org/abs/1405.4053>

6. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., & Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. *ArXiv*, *abs/1310.4546*.
7. Pennington, J., Socher, R., & Manning, C.D. (2014). GloVe: Global Vectors for Word Representation. *Conference on Empirical Methods in Natural Language Processing*.
8. Low, Cherlynn. "Facebook is open-sourcing its AI bot-building research". Engadget. Retrieved 12 January 2018.

# 8

## Doc2Vec with Gensim

Doc2Vec is a technique for document embedding and text analysis. Doc2Vec can be used to build document retrieval systems, helping users find relevant documents. It can be used in recommendation systems to suggest articles, news, or products to users based on their historical preferences or behavior. It has enabled many real-world applications in a variety of domains. In healthcare, it is used with **electronic health records (EHRs)** to classify medical documents. In the legal field, it helps to organize and classify legal documents, particularly in e-discovery and case law research. In social media texts, it is used to identify sentiment and user behavior to help companies understand public opinion or identify emerging topics. In academic research, it helps to identify related research papers and explore connections between different studies. Similar applications exist for e-commerce, travel websites, and job-recruiting websites.

With the rise of LLMs that are capable of various tasks including search engine functionality and information retrieval, is it worthwhile to learn Doc2Vec? Whether LLMs are necessarily better for search engines than Doc2Vec depends on the specific use case and the requirements of the search engine. Both types of models have their own strengths and weaknesses, and the choice between them should be made based on the specific needs of the application. Let's compare them in terms of contextual understanding and resource constraints. In terms of contextual understanding, LLMs can generate contextually relevant text and might be better suited for understanding the context and meaning of documents. In contrast, Doc2Vec is particularly useful when you want to compare documents or find similar documents based on their content. In terms of resource requirement, LLMs usually require powerful hardware and may not be suitable for resource-constrained environments. In contrast, Doc2Vec models are typically more lightweight and easier to implement, making them more practical for certain applications and platforms.

Therefore, if you consider factors such as data availability, resource constraints, and the nature of your search queries, you may consider Dev2Vec a viable alternative. For these reasons, it is still worthwhile devoting your time to learning Doc2Vec.

In this chapter, I will describe the evolution from Word2Vec to Doc2Vec and the modeling techniques. Then I will show you how to build Doc2Vec models and put the models to use scoring new documents. Specifically, you will learn about the following topics:

- From Word2Vec to Doc2Vec
- PV-DBOW
- PV-DM
- The real-world applications of Doc2Vec
- Doc2Vec modeling with Gensim
- Putting the model into production
- Tips for building a good Doc2Vec model

By the end of this chapter, you will understand the two neural network architectures of Doc2Vec (PV-DBOW and PV-DM). You will be able to articulate the evolution of Word2Vec to Doc2Vec. You will gain hands-on knowledge of developing Doc2Vec models.

## Technical requirements

We need to install Gensim 3.8.3:

```
pip install gensim==3.8.3
```

Other versions of Gensim may incur an error message for Doc2Vec:

```
AttributeError: 'Doc2Vec' object has no attribute 'dv'
```

After you installed Gensim 3.8.3, import the following:

```
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
```

You can find code files for this chapter on GitHub at <https://github.com/PacktPublishing/The-Handbook-of-NLP-with-Gensim/tree/main/Chapter08>.

## From Word2Vec to Doc2Vec

Word2Vec, pioneered by Mikolov et al. in 2013 [1], generates vector representations for individual words in a text corpus. It represents words as continuous vectors that capture the semantic meaning of words in a high-dimensional space. Doc2Vec, led by Le and Mikolov [2], extends the idea of Word2Vec to generate vector representations for entire documents or paragraphs. It represents documents as continuous vectors in a similar vector space where documents with similar content or meaning are closer together. Doc2Vec has a wide range of applications for tasks involving document-level analysis such as document similarity, content recommendation, document clustering, and text summarization. The *document* in Doc2Vec can be a sentence, a paragraph, or an entire article. Le and Mikolov [2] refer to Doc2Vec as **Paragraph Vector (PV)** to emphasize the fact that it transforms a paragraph into a vector. In Word2Vec, each word has a unique ID. In Doc2Vec, each paragraph is assigned a paragraph ID. By using IDs for paragraphs, Doc2Vec employs a fantastic idea: why don't we consider a paragraph ID as a word? *When a paragraph ID is represented as a word, it can be embedded as a vector.*

Since Doc2Vec was inspired by Word2Vec, their model architectures resemble each other. Word2Vec model is a shallow neural network with an input layer, a hidden layer, and an output layer. Likewise, Doc2Vec is a simple neural network with input, hidden, and output layers. Word2Vec has two variations of model architectures: the Skip-Gram model and the CBOW model. Doc2Vec has two model architectures: the PV-DBOW model and the PV-DM model. The abbreviation **PV-DBOW** stands for **Paragraph Vector with Distributed Bag-of Words** and **PV-DM** stands for **Paragraph Vectors—Distributed Memory**. Their similarities are shown in *Figure 8.1*:

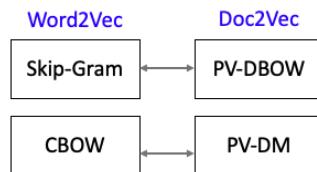


Figure 8.1 – From Word2Vec to Doc2Vec

Let's learn about PV-DBOW first.

## PV-DBOW

*Figure 8.2* is the neural network for PV-DBOW that has an input layer, a hidden layer, and an output layer. The input layer is a vector of the paragraph IDs. Assume a corpus has 500 paragraphs. Each of the paragraph IDs is one-hot encoded and the length of each paragraph vector is 500. For example, **Paragraph “1”** is a  $1 \times 500$  vector where only the position of “1” is 1 and the rest are zeros.

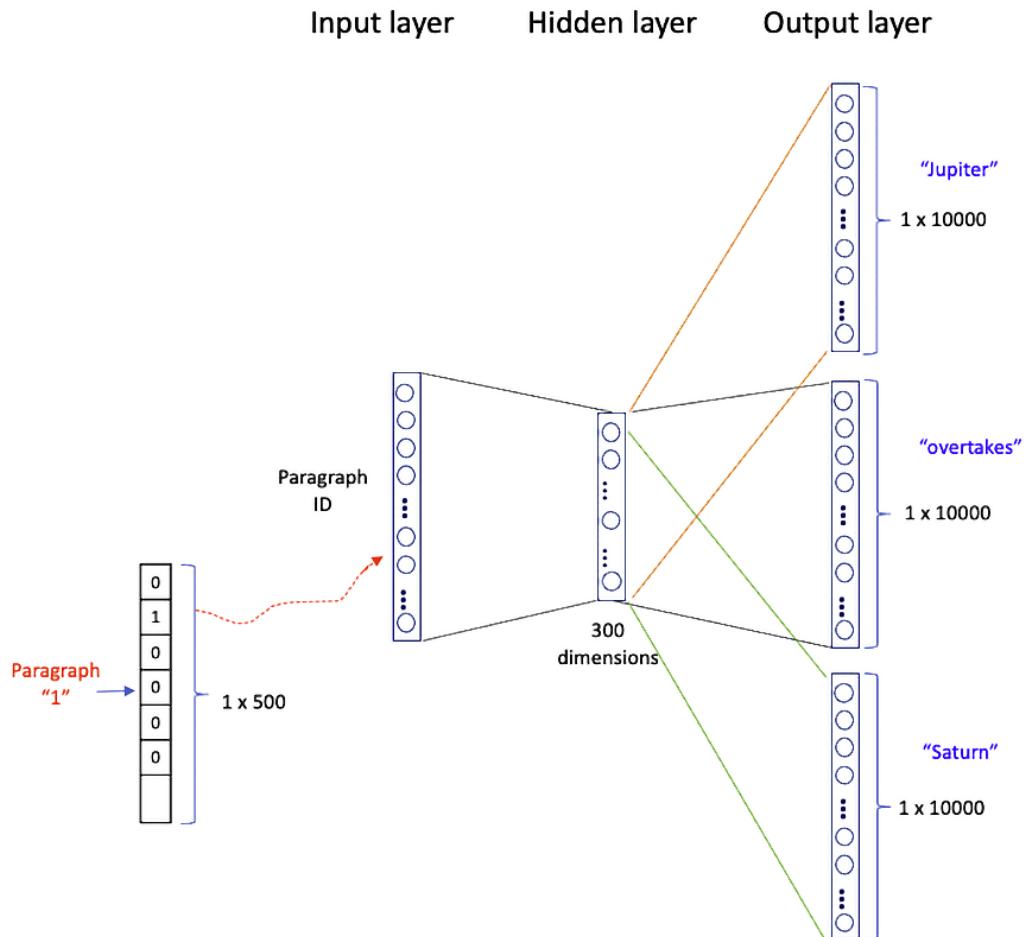


Figure 8.2 – PV-DBOW

Let's see how a paragraph is prepared to feed into the neural network model.

The neural network requires data to follow the (input, output) format. Let's first see how to do this. In Word2Vec, we organize texts into word pairs to feed into its neural network model. Its format is (word, adjacent word) for the input and output layer. In Doc2Vec, the format for the word pairs is (paragraph ID, word) for the input and output layer. Assume we have two paragraphs:

- Paragraph 1: *"Jupiter overtakes Saturn as the planet with the most known moons."*
- Paragraph 2: *"Honeybees waggle to communicate."*

The word pairs of (paragraph ID, word) for the sentences are shown in *Figure 8.3*.

	Source Text	Word Pair
1	Jupiter overtakes Saturn as the planet with the most known moons.	(“1”, Jupiter), (“1”, overtakes), (“1”, Saturn)
1	Jupiter overtakes Saturn as the planet with the most known moons.	(“1”, overtakes), (“1”, Saturn), (“1”, as)
1	Jupiter overtakes Saturn as the planet with the most known moons.	(“1”, Saturn), (“1”, as), (“1”, the)
1	Jupiter overtakes Saturn as the planet with the most known moons.	(“1”, as), (“1”, the), (“1”, planet)
1	Jupiter overtakes Saturn as the planet with the most known moons.	(“1”, the), (“1”, planet), (“1”, with)
1	Jupiter overtakes Saturn as the planet with the most known moons.	(“1”, planet), (“1”, with), (“1”, the)
2	Honeybees waggle to communicate.	(“2”, honeybees), (“2”, waggle), (“2”, to)
2	Honeybees waggle to communicate.	(“2”, waggle), (“2”, to), (“2”, communicate)

Figure 8.3 – An overview of data preparation for PV-DBOW

In this example, the window size for words is 3 words. A larger window size can be tested in modeling. Also, the word window in PV-DBOW does not require the words following the word ordering. It can sample words randomly from the window of words. The word ordering in *Figure 8.3* can be considered a special case of random sampling in PV-DBOW.

Now let's understand each layer.

## The input layer

In *Figure 8.3* the paragraph ID in each word pair is the input for the input layer. Each ID is one-hot encoded as a  $1 \times 500$  vector. For example, Paragraph “123” shall become a  $1 \times 500$  vector where the 123<sup>rd</sup> element in the array is 1 and the rest are zeros.

## The hidden layer

The hidden layer is a crucial component of the neural network architecture used to learn paragraph vectors. The hidden layer captures the semantic information of documents in continuous-valued vectors. The hidden layer has  $N$  nodes. Each node is a weighted sum of the input nodes. Between the input layer and the hidden layer is the  $500 \times N$  weight matrix, which we are interested in. The  $i^{\text{th}}$  row in the matrix is the weight for the  $i^{\text{th}}$  paragraph. Because most of the paragraph nodes are zeros, the updates during training are typically sparse and efficient.

## The output layer

In *Figure 8.3* the word in a word pair is for the output layer. Each word is one-hot encoded to be a vector. Suppose there are 10,000 unique words in the texts, then the length of a vector is 10,000. For example, the vector for the word “Jupiter” is a  $1 \times 10,000$  vector in which the position for “Jupiter” is 1 and all others are 0. In other words, the paragraph ID “1” in the word pair (“1”, Jupiter) will become a  $1 \times 500$  vector to the input layer, and the word “Jupiter” will become a  $1 \times 10,000$  vector to the output layer.

## Model optimization

The overall objective is to adjust the document vectors in such a way that they are good at predicting the actual context words for the given document while being poor at predicting randomly sampled negative context words. The standard softmax is supposed to be used in neural networks to compute a probability distribution over a set of classes. The classes are words in NLP.

However, the standard softmax is not really practical for large vocabularies. When the size of vocabularies is large, the computation for naive softmax computation will become exponentially expensive and impractical. To address this issue, more efficient techniques such as **negative sampling** have been developed. Negative sampling focuses on a smaller, randomly chosen subset of negative examples (words not in the context) for each positive example (words in the context). Thus, for each training example (paragraph ID, word), PV-DBOW randomly samples a few “negative” context words from the vocabulary. These negative samples are words that are not the actual context words.

The loss for each training instance is calculated using binary logistic regression. For the positive example, the model aims to predict a probability close to 1 (indicating that the actual context word is indeed a context word for the given document). For the negative examples, the model aims to predict probabilities close to 0 (indicating that these randomly chosen words are not context words for the document). Finally, the logistic sigmoid function is typically used to transform the model’s predictions into probabilities between 0 and 1. The sigmoid function takes the dot product of the document vector and the word vector for each context word candidate and applies the sigmoid transformation.

With this understanding of how PV-DBOW works under our belts, let’s learn the second architecture, PV-DM.

## PV-DM

The word pairs in PV-DM are arranged as shown in *Figure 8.4*. It adds the paragraph IDs in the texts then uses a sliding window to form word pairs. For example, Paragraph “1” has “Jupiter overtakes Saturn...”, so the word pairs are (“1”, Saturn), (Jupiter, Saturn), and (overtakes, Saturn).

	Source Text	Word Pair
1	Jupiter overtakes Saturn as the planet with the most known moons.	(1, Saturn), (Jupiter, Saturn), (overtakes, Saturn)
1	Jupiter overtakes Saturn as the planet with the most known moons.	(1, as), (overtakes, as), (Saturn, as)
1	Jupiter overtakes Saturn as the planet with the most known moons.	(1, the), (Saturn, the), (as, the)
1	Jupiter overtakes Saturn as the planet with the most known moons.	(1, planet), (as, planet), (the, planet)
1	Jupiter overtakes Saturn as the planet with the most known moons.	(1, with), (the, with), (planet, with)
1	Jupiter overtakes Saturn as the planet with the most known moons.	(1, the), (planet, the), (with, the)
2	Honeybees waggle to communicate.	(2, to), (honeybees, to), (waggle, to)
2	Honeybees waggle to communicate.	(2, communicate), (waggle, communicate), (to, communicate)

Figure 8.4 – An overview of Data preparation for PV-DM

Figure 8.5 shows the neural network for PV-DM.

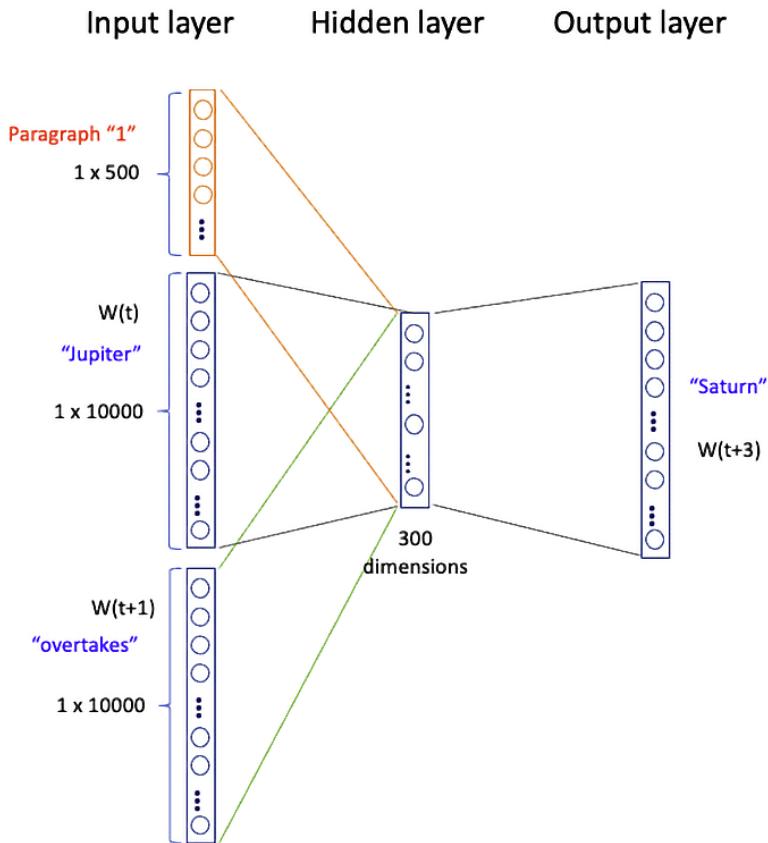


Figure 8.5 – PV-DM

The word pairs in Figure 8.4 are the inputs for the input and output layers. Different from the word pairs of PV-DBOW in Figure 8.3, the word pairs in Figure 8.4 only have one instance for the paragraph ID. Each paragraph ID is one-hot encoded as a  $1 \times 500$  vector. Again, for example, paragraph "123" shall become a  $1 \times 500$  vector where the 123<sup>rd</sup> element in the array is 1 and the rest are zeros. All the words are one-hot encoded to be  $1 \times 10,000$  vectors. Hence, the pair (1, Saturn) is a  $1 \times 500$  paragraph vector for the input layer and a  $1 \times 10,000$  vector for the output layer.

Between the paragraph vector and the hidden layer is the  $500 \times 300$  weight matrix. This is the weight matrix that we are interested in. In this setting, a paragraph ID will be embedded as a vector in the hidden layer.

Do you notice the word order is maintained in PV-DM? Thus PV-DM performs consistently better than PV-DBOW, as documented by the authors [2]. Further, the paragraph ID acts as a memory that remembers what is missing from the context. It is the reason that the authors [2] call the model the **Distributed “Memory” Model of Paragraph Vectors (PV-DM)**.

## The real-world applications of Doc2Vec

The Doc2Vec technique has been used in many fields in which text data is the most important asset. Let me give some examples of the applications.

Job boards and professional social networks use recommender systems to recommend similar job postings. When you look for a job on LinkedIn or Indeed.com, you may see similar job postings presented next to your target job posting. It is done by Doc2Vec. Doc2Vec also is used by companies such as Airbnb and Alibaba to build their product recommendation systems [3] [4].

Legal professionals need a legal document recommendation system that can automatically pull similar judgments to prepare their arguments in the court. Legal textual information is domain specific. Dhanani, Mehta, and Rana presented that Doc2Vec can perform very rich embedding results [5] [6]. Doc2Vec was even leveraged to uncover the relationships between diseases and disease-genes associations. Gligorijevic et al. [7] found the embedding of both diseases and genes can predict the associations between diseases and genes with very high precision.

Now we are ready to build our Doc2Vec model with the AG news data. Doc2Vec requires text preprocessing. Hence, let's load the data and perform text preprocessing.

## Doc2Vec modeling with Gensim

Just to recap what I have described in *Preface*, this book uses the sampled AG's corpus of news articles. This dataset is a smaller collection that sampled news articles on “world”, “sports”, “business”, and “Science”. It has been used extensively in many NLP modeling projects and is available in Kaggle, PyTorch, Huggingface, and TensorFlow. The data has four classes: Class “1” is news about “World”, Class “2” is news about “Sports”, “3” is “Business”, and “4” is “Sci/Tech”. I will walk you through the following tasks:

- Text preprocessing for Doc2Vec
- Modeling
- Saving the model
- Saving the training data

So let's start with the first step.

## Text preprocessing for Doc2Vec

Let's apply the preprocessing procedure to the dataset:

```
import pandas as pd
import numpy as np
from gensim.parsing.preprocessing import preprocess_string
pd.set_option('display.max_colwidth', -1)
path = '/content/gdrive/My Drive/data/gensim'
data = pd.read_csv(path + "/ag_news_train.csv")
train = list(data['Description'])
```

The quality of a Doc2Vec model outcome depends on good text preprocessing. The preprocessing includes several key steps:

1. Tokenization
2. Lowercase conversion
3. Stop words removal
4. Punctuation removal
5. Stemming

We will use Gensim to do text preprocessing. The following code first declares an empty list called `text_tokenized` to collect all the preprocessed documents. The code iterates each document in the raw text and performs preprocessing by using the `preprocess_string()` function:

```
train_tokenized = []
for doc in train:
    k = preprocess_string(doc)
    train_tokenized.append(k)

train_tokenized[0:3]
```

The list of lists looks like this:

```
[['reuter', 'short', 'seller', 'wall', 'street', 'dwindl', 'band',
'ultra', 'cynic', 'see', 'green'],
['reuter', 'privat', 'invest', 'firm', 'carlyl', 'group', 'reput',
'make', 'time', 'occasion', 'controversi', 'plai', 'defens',
'industri', 'quietli', 'place', 'bet', 'market'],
['reuter', 'soar', 'crude', 'price', 'plu', 'worri', 'economi',
'outlook', 'earn', 'expect', 'hang', 'stock', 'market', 'week',
'depth', 'summer', 'doldrum']]
```

A critical step in Doc2Vec modeling is to tag a paragraph with a paragraph ID. We will use `TaggedDocument` in Gensim to do so:

```
# Gensim libraries
import gensim
import pprint as pp
import random
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
```

The tagging is just to assign a number for each paragraph. We use `tags = [str(i)]` for the  $i^{\text{th}}$  paragraph.

```
train_tagged_data = [TaggedDocument(words=w, tags=[str(i)])
                     for i,w in enumerate(train_tokenized[0:3])]
```

The first three paragraphs look like the following block:

```
pp pprint(train_tagged_data)
[TaggedDocument(words=['reuter', 'short', 'seller', 'wall', 'street',
'dwindl', 'band', 'ultra', 'cynic', 'see', 'green'], tags=['0']),
TaggedDocument(words=['reuter', 'privat', 'invest', 'firm', 'carlyl',
'group', 'reput', 'make', 'time', 'occasion', 'controversi',
'plai', 'defens', 'industri', 'quietli', 'place', 'bet', 'market'],
tags=['1']),
TaggedDocument(words=['reuter', 'soar', 'crude', 'price', 'plu',
'worri', 'eonomi', 'outlook', 'earn', 'expect', 'hang', 'stock',
'market', 'week', 'depth', 'summer', 'doldrum'], tags=['2'])]
```

Let's tag all paragraphs:

```
train_tagged_data = [TaggedDocument(words=w, tags=[str(i)]) for i,w in
enumerate(train_tokenized)]
```

## Modeling

Gensim makes Doc2Vec modeling quite straightforward. Many hyperparameters already have default values based on academic research. The following code declares the model, builds a vocabulary, trains the model, and then saves it as a physical file:

```
model = gensim.models.doc2vec.Doc2Vec(dm=1, window = 5,
min_count=2, epochs=10)
model.build_vocab(train_tagged_data)
model.train(train_tagged_data,
            total_examples=model.corpus_count)
model.save(path + "/doc2vec.model")
```

Let me explain the preceding code step by step:

- `dm`: The parameter `dm` defines the algorithm. Value 1 is PV-DM and 0 is PV-DBOW. The default is 1.
- `window`: The maximum distance between the current and predicted word within a sentence. The default is 5.
- `min_count`: Ignores all words with a frequency lower than this minimum count. The default is 5. If you have a small corpus and there is no word in the vocab with a frequency greater than 5, the vocab will be empty. You will get an error message in the `.build_vocab()` step. Because our dataset is small, we set `min_count` to 2.
- `build_vocab()`: Builds a vocabulary from a sequence of documents.
- `epochs`: Number of iterations over the corpus.

Let's save the data and tokenized data for future use.

## Saving the model

The following code saves the model:

```
import pickle
train_file = path + '/ag_news_train.pkl'
train_tokenized_file = path + '/ag_news_train_tokenized.pkl'
```

## Saving the training data

Also, the training data needs to be saved for future use, which is done using the following snippet:

```
with open(train_file, "wb") as fp:
    pickle.dump(train, fp)
with open(train_tokenized_file, "wb") as fp:
    pickle.dump(train, fp)
```

Now we are ready to put the model into production.

## Putting the model into production

Recall that in LSI Gensim we loaded four objects: the dictionary list, the model, the BoW object, and the TF-IDF object. In Doc2Vec, we only need to load the model object. This is because Doc2Vec does not build on BoW or TF-IDF. I also load the training data just for convenience. It is not required in real-time production.

How do we use Doc2Vec in production? It can be used just like a search engine to retrieve relevant documents based on keyword search. It can also be used to return similar articles to an article of choice. I will demonstrate both use cases. Before that, let's see how to load your model and training data.

### Loading the model

Gensim has a `get_tmpfile` utility function that points to the physical location of the file. We will use it to reference the location of the model to load the model:

```
from gensim.test.utils import get_tmpfile
fname = get_tmpfile(path + "/doc2vec.model")
model = Doc2Vec.load(fname)
```

### Loading the training data

Let's also load the training data using the following code:

```
import pickle
train_file = path + '/ag_news_train.pkl'
train_tokenized_file = path + '/ag_news_train_tokenized.pkl'
with open(train_file, "rb") as fp:
    train = pickle.load(fp)
with open(train_tokenized_file, "rb") as fp:
    train_tokenized = pickle.load(fp)
```

### Use case 1 – find similar articles

I randomly chose document 79 to see if the model can find other similar documents. Document 79 is about a new wireless network development from a consortium of technology companies:

```
take_one = 79
print(train[take_one])
```

The output looks like this:

Reuters—A group of technology companies including Texas Instruments Inc. (TXN.N), STMicroelectronics (STM.PA) and Broadcom Corp. (BRCM.O), on Thursday said they will propose a new wireless networking standard up to 10 times the speed of the current generation.

To find similar documents, Gensim has the `most_similar()` function for Doc2Vec. Remember Doc2Vec represents every document (i.e., paragraph ID) as a vector of words. The `most_similar()` function computes the cosine similarity between the vector of the target document and the other given documents. If you are not familiar with cosine similarity, go back and review *Chapter 5, Cosine Similarity*:

```
similar_doc = model.docvecs.most_similar(take_one)
print('The top similar ones are:')
pp.pprint(similar_doc)
```

The documents with the most similarity to document 97 are the following:

```
[('1153',  0.9218629002571106),
 ('51164', 0.744012176990509),
 ('61822', 0.7407793998718262),
 ('93624', 0.7354568243026733),
 ('42628', 0.714421272277832),
 ('105967', 0.7088165283203125),
 ('72362', 0.7087656259536743),
 ('29430', 0.701684832572937),
 ('2782', 0.698917031288147),
 ('93680', 0.6976069211959839)]
```

Let's print out those documents. The first document 1153 reads as follows:

Id: 1153 the news is: LOS ANGELES (Reuters)—A group of technology companies including Texas Instruments Inc.; on Thursday said they will propose a new wireless networking standard up to 10 times the speed of the current generation.

The second document 51164 contains the following:

Id: 51164 the news is: New version of the Efficeon processor uses only 3 watts of power at 1-GHz speed.

It is about new technology advancement. The third document 61822 looks as follows:

Id: 61822 the news is: Products expected soon that will connect via a new home networking specification backed by some of the world's largest consumer electronics and computer companies.

The preceding one is about new home network specifications released by computer companies. All of these documents do indeed appear to be related to the chosen document 79. Let's see all of them:

```
for i in range(len(similar_doc)):
    id = similar_doc[i][0]
    print('Id:', id, 'the news is:', train[int(id)])
```

The output is as follows:

```
Id: 1153 the news is: LOS ANGELES (Reuters)—A group of technology
companies including Texas Instruments Inc.; on Thursday said they will
propose a new wireless networking standard up to 10 times the speed of
the current generation.

Id: 51164 the news is: New version of the Efficeon processor uses only
3 watts of power at 1-GHz speed.

Id: 61822 the news is: Products expected soon that will connect via
a new home networking specification backed by some of the world's
largest consumer electronics and computer companies.

Id: 93624 the news is: NEW YORK (Reuters)—SBC Communications; on
Wednesday said it reached a 10-year, $400 million deal with Microsoft
Corp.; to provide next-generation television services.

Id: 42628 the news is: September 28, 2004—Michael Powell, chairman of
the Federal Communications Commission, plans to hold a vote this year
on a proposal to require US television broadcasters such as Viacom
Inc.
```

## Use case 2 – find relevant documents based on keywords

While use case 1 involves a search for similar documents based on the target document, we can use Doc2Vec as we would a search engine to search for documents based on keywords. We did this in *Chapter 6, Latent Semantic Indexing with Gensim*, for the LSI model. In that chapter, we were interested in news relating to the economic outlook and general prices. We entered multiple keywords as a new document: "Crude prices inflation the economy outlook earnings". We have preprocessed the new document in *Chapter 6, Latent Semantic Indexing with Gensim*, so we will do the same now:

```
from gensim.parsing.preprocessing import preprocess_string
doc = "Crude prices inflation the economy outlook earnings"
doc_tokenized = preprocess_string(doc)
doc_tokenized
```

The tokenized words are as follows:

```
['crude', 'price', 'inflat', 'economi', 'outlook', 'earn']
```

We will use the `infer_vector(doc_tokenized)` function in Gensim's Doc2Vec. It can infer an unseen document from a document in the training data. The `inferred_vector` output is a vector:

```
inferred_vector = model.infer_vector(doc_tokenized)
```

Then we use the `most_similar()` function to calculate the similarity of the vector to the vectors of other documents in the training data:

```
sims = model.docvecs.most_similar([inferred_vector],  
topn=len(model.docvecs))
```

The `sims` output is a two-tuple list (paragraph ID, similarity score) ranked in descending order. There are 12,000 similarity scores because in our case there are 12,000 documents in the training data vectors based on the similarity measure. Let's only take the top 10:

```
see = sims[0:5]
```

The output is as follows:

```
[('112510', 0.8218212127685547),  
 ('53000', 0.7832474708557129),  
 ('4420', 0.7805075645446777),  
 ('44328', 0.7804965972900391),  
 ('52696', 0.7767146229743958)]
```

Let's print out these documents:

```
for i in range(len(see)):  
    id = see[i][0]  
    print('Id:', id, 'the news is:', train[int(id)])
```

The results are the following:

```
Id: 112510 the news is: NEW YORK (Reuters) - The dollar gained  
broadly on Friday for the third straight session, as strong U.S.  
economic data gave traders additional reasons to take profits in other  
currencies before the year's end.  
Id: 53000 the news is: AP - Tokyo stocks fell Thursday as investors  
took profits after five straight sessions of gains. The dollar was  
down against the Japanese yen.  
Id: 4420 the news is: NEW YORK Despite volatile scepticism among  
investors, Google's stock jumped about 18 percent to $100.01 a share  
when it debuted Thursday on the Nasdaq stock market.  
Id: 44328 the news is: Derek Jeter and Alex Rodriguez rallied the  
Yankees past the Minnesota Twins in the first game of a doubleheader  
at Yankee Stadium.  
Id: 52696 the news is: Tokyo stocks fell Thursday as investors took  
profits after five straight sessions of gains. The dollar was down  
against the Japanese yen.
```

As you can see, these documents are all economic news and have implications on the economic outlook. Before we close this chapter, I would like to share some tips with you to build a good Doc2Vec model.

## Tips on building a good Doc2Vec model

I would like to offer two perspectives:

- It is important to train Doc2Vec with lots of data to achieve stable results. The more words in a paragraph/document, the better. A document with less than 5 words is not easy to differentiate from any other.
- Lemmatization or stemming may not necessarily improve the results. You could try and test a model with lemmatization and a model without. The authors [2] report that “PV-DM is consistently better than PV-DBOW.” Hence the default model in the Gensim class is the PV-DM model.

That completes our discussion in this chapter.

## Summary

In this chapter, we learned about the development of Word2Vec to Doc2Vec. Doc2Vec is based on the idea that any document, be it a sentence or a paragraph or an entire article, can be represented by a vector. This can be described by the term *paragraph vector*. We learned the two variations of the Doc2Vec model architecture: the PV-DBOW model and the PV-DM model. We also built Doc2Vec models and examined their outcomes.

In the next few chapters, we will learn about another milestone in NLP: Latent Dirichlet Allocation for topic modeling. We will learn what the Dirichlet distribution is and then learn about the model itself.

## Questions

1. What do PB-DBOW and PV-DM stand for?
2. Compare the two model variations of Doc2Vec to those of Word2Vec.
3. Explain some real-world applications of Doc2Vec.
4. How does Doc2Vec convert a paragraph into a vector?

## References

1. Mikolov, T., Chen, K., Corrado, G.S., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *International Conference on Learning Representations*. <https://arxiv.org/abs/1310.4546>
2. Le, Q.V., & Mikolov, T. (2014). Distributed Representations of Sentences and Documents. International Conference on Machine Learning.
3. Mihajlo Grbovic and Haibin Cheng. 2018. Real-time Personalization using Embeddings for Search Ranking at Airbnb. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '18). Association for Computing Machinery, New York, NY, USA, 311–320.
4. Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '18). Association for Computing Machinery, New York, NY, USA, 839–848. <https://doi.org/10.1145/3219819.3219869>
5. Dhanani, J., Mehta, R. & Rana, D. Effective and scalable legal judgment recommendation using pre-learned word embedding. *Complex Intell. Syst.* 8, 3199–3213 (2022). <https://doi.org/10.1007/s40747-022-00673-1>
6. Dhanani, J., Mehta, R. and Rana, D.P. (2021), “Legal document recommendation system: a dictionary based approach”, *International Journal of Web Information Systems*, Vol. 17 №3, pp. 187–203. <https://doi.org/10.1108/IJWIS-02-2021-0015>
7. Gligorijevic, D., Stojanovic, J., Djuric, N. et al. Large-Scale Discovery of Disease-Disease and Disease-Gene Associations. *Sci Rep* 6, 32404 (2016). <https://doi.org/10.1038/srep32404>



# Part 4:

## Topic Modeling with Latent Dirichlet Allocation

Topic modeling can produce rich information for each topic. In this part, you will learn about a milestone topic modeling technique – the **Latent Dirichlet Allocation**. There are several fundamental concepts applied in LDA, thus this part introduces the key concepts in different chapters.

First, you will learn about the discrete distribution family from the simple to more generalized forms, including the Dirichlet distribution. With that, this part presents the intuition and architecture of an LDA. With the theoretical understanding, you will build LDA with Gensim.

The next question is how to visualize and communicate the results. Here, you will appreciate the design of the infographic for LDA. Further, you will understand why LDA may not produce reliable results and the advance to Ensemble LDA for model stability.

This part contains the following chapters:

- *Chapter 9, Understanding Discrete Distributions*
- *Chapter 10, Latent Dirichlet Allocation*
- *Chapter 11, LDA Modeling*
- *Chapter 12, LDA Visualization*
- *Chapter 13, The Ensemble LDA for Model Stability*



# 9

## Understanding Discrete Distributions

Latent Dirichlet Allocation (LDA) is named based on its mathematical concepts. Its name contains the word *latent* because it finds a similarity between documents in the latent space. It contains the word *allocation* because it allocates a document to topics. But most of all, it contains the word *Dirichlet* because it is based on the Dirichlet (pronounced as “Deer-e-kh-let”) distribution.

The Dirichlet distribution belongs to the discrete distribution family, which includes the Bernoulli, binomial, multinomial, beta, and Dirichlet distributions. The binomial and multinomial distributions are already frequently used in many machine-learning models that many of you will be familiar with. The beta distribution is a generalization of the binomial distribution, and the Dirichlet distribution is a generalization of the beta distribution. Further, the beta distribution is the conjugate prior probability distribution for the Bernoulli and binomial distributions in Bayesian inference, and the Dirichlet distribution is the conjugate prior probability distribution for the multinomial distributions. For these reasons, a good learning strategy is to cover all of them in a sequence. I will explain the concepts of “conjugate” and “Bayesian” in this chapter.

To keep you engaged in both the mathematics and the applications, I will present each distribution with the following subsections:

- The real-world examples
- The formal definition
- What does it look like?
- Fun facts

Therefore, this chapter covers the family of discrete probability distributions, from simple to complex, in the following sequence:

- The basics of discrete probability distributions
- Bernoulli distributions
- Binomial distributions
- Multinomial distributions
- Beta distributions
- Dirichlet distributions

By the end of this chapter, you will have a clear picture of Dirichlet distributions and how these distributions build on each other. You will learn how to visualize them with Python. Now that you're familiar with this learning path, let's start!

## Technical requirements

We will use `numpy`, `scipy.stats`, and `matplotlib`:

```
import scipy.stats as stats
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

You can find the code files for this chapter on GitHub at <https://github.com/PacktPublishing/The-Handbook-of-NLP-with-Gensim/tree/main/Chapter09>.

## The basics of discrete probability distributions

Let's start with the fundamentals. What is a random variable? A **random variable** is a set of possible outcomes from a random experiment. For example, if we want to know whether tomorrow's weather is sunny or rainy, then "tomorrow's weather" is the random variable, and the possible outcomes are "sunny" and "rainy." A random variable can be discrete or continuous. If a random variable takes only a finite number of distinct values such as "sunny" or "rainy," it is a **discrete random variable**. If a random variable can have a continuum of infinite and uncountable outcomes, it is a **continuous random variable**. Then, a **discrete probability distribution** is a distribution that shows all possible discrete values with respective probabilities for each value. When we say tomorrow is 70% sunny and 30% rainy, we are saying the random  $x$  variable has two outcomes – 70% sunny and 30% rainy. Further, the mathematical function that describes the probability distribution of a continuous random variable is called the **Probability Density Function (PDF)**. If it is for a discrete variable, it is called the **Probability Mass Function (PMF)**.

With that in mind, let's start with the simplest distribution – a Bernoulli distribution.

## Bernoulli distributions

The Bernoulli distribution is a simple discrete distribution that describes the probability of binary outcomes in an experiment. There are many examples of Bernoulli distributions in our daily lives. Let's see some real-world examples:

- What is the chance of a student passing an exam?
- What is the chance of a team winning a championship?
- What is the probability of getting an even number when a fair dice is thrown once?

All these cases have **one event** or **one trial** – either “yes” or “no,” or “pass” or “fail.” Let's see its formal definition.

### The formal definition of a Bernoulli distribution

A discrete distribution has two possible outcomes:

$$P(X = x) = \begin{cases} p & \text{if } x = 1 \\ q = 1 - p & \text{if } x = 0 \end{cases}$$

This relationship is commonly expressed in an exponential form:

$$P(x) = p^x(1 - p)^{1-x} \text{ for } x \in \{0, 1\} \quad \text{Eq. (1)}$$

So, the example of a student passing an exam can be written as follows:

$$P(X = \text{"pass"}) = \begin{cases} p & \text{if "pass"} \\ q = 1 - p & \text{if "fail"} \end{cases}$$

Or in an exponential form as follows:

$$P(x) = p^x(1 - p)^{1-x} \text{ for } x \in \{\text{"pass"}, \text{"fail"}\}$$

### What does it look like?

Although I have not introduced the binomial distribution, let me tell you that the Bernoulli distribution is a special case of the binomial distribution when  $n = 1$ . Understanding this, we can use the **Binomial Distribution Visualization** calculator, as shown in *Figure 9.1*, to visualize the Bernoulli distribution.

*Figure 9.1* shows a Bernoulli distribution with only two values, 0 and 1. Note that **Number of trials (n)** is set to **1**. The **Probability of a Success** marker in the slide bar lets you set the probability to any point between **0** and **1**. Here, the probability is set at **0.76**, so the probability of 1 for the Bernoulli distribution is 0.76.

## Binomial Distribution Visualization

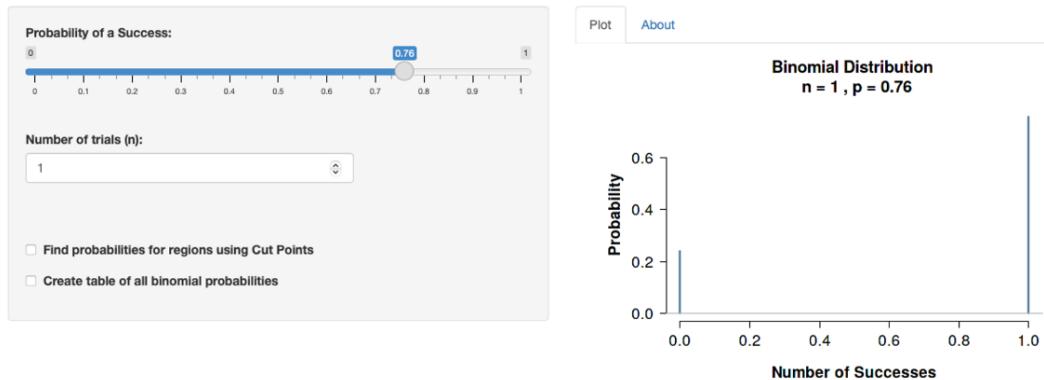


Figure 9.1 – The binomial distribution

### Fun facts

You might be asking, why don't we just give the Bernoulli distribution a simple name such as "a distribution with binary outcomes"? The distribution and the trial are named after the Swiss mathematician Jacob Bernoulli (1654–1705) [1].

The Bernoulli distribution concerns only one trial with binary outcomes. How about repeating trials with binary outcomes? You have asked a great question! That takes us to the binomial distribution.

## Binomial distributions

When there are many independent trials (also called experiments or samples) and each trial has two exact outcomes, a Bernoulli distribution is generalized to the binomial distribution. The binomial distribution measures the number of successes in a sequence of independent experiments, each asking a yes/no question.

### The real-world examples

The following two examples explain the binomial distribution well:

- What is the probability of finding students who passed the final exam when examining 50 students? Suppose the probability of students passing the exam is 0.8. This can be described by a binomial distribution,  $p(50,0.8)$ .
- What is the probability of finding drivers who do not have car insurance when examining 100 drivers? Suppose the probability/proportion of drivers not having car insurance is 0.05. This can be described by a binomial distribution as  $p(100,0.05)$ .

All these examples have the following common elements:

- A fixed number of samples (experiments/trials), such as the number of students or the number of drivers
- Two mutually exclusive outcomes, such as pass/fail or with/without a driver's license
- The probability of success ( $p$ ) – 80% chance to pass an exam, or 5% of drivers without car insurance
- Independent trials – yes

Let's learn how it is formulated.

## The formal definition of a binomial distribution

A binomial distribution is a discrete distribution that describes the number of successes in a fixed number of independent Bernoulli trials or experiments. It is characterized by two parameters –  $n$ , the number of trials or experiments, and  $p$ , the fixed number of times the experiment is repeated. The PMF is given by:

$$P(x; n, p) = \binom{n}{x} p^x (1 - p)^{n-x}$$

There are only two outcomes – 1 and 0, with the probabilities  $p$  and  $1 - p$ , respectively. In the  $n$  experiments, the outcome 1 happens  $x$  times, and the outcome 0 happens  $n - x$  times. It is read as “the probability of getting exactly  $x$  successes in  $n$  independent trials with two outcomes.” The first term is the number of all combinations. It exists in the formula, so the number of times divided by the total number of combinations can become a probability. It is commonly written as a factorial term:

$$\binom{n}{x} = \frac{n!}{x!(n-x)!}$$

So, the probability of a binomial distribution is written as follows:

$$P(x; n, p) = \frac{n!}{x!(n-x)!} p^x (1 - p)^{n-x} \quad Eq. (2)$$

I believe you will get a good idea when you see it visually.

## What does it look like?

Use the **Binomial Distribution Visualization** calculator (<https://shiny.rit.albany.edu/stat/binomial/>) to see the distribution. When **Number of trials (n)** is set to any value more than 1, it becomes a binomial distribution. Let's assume we are flipping a fair coin, so the outcomes are either 1 (head) or 0 (tail). In *Figure 9.2*, I have set the **Probability of a Success** slider at 0.5. Let's observe the results of different  $n$  values.

## Binomial Distribution Visualization

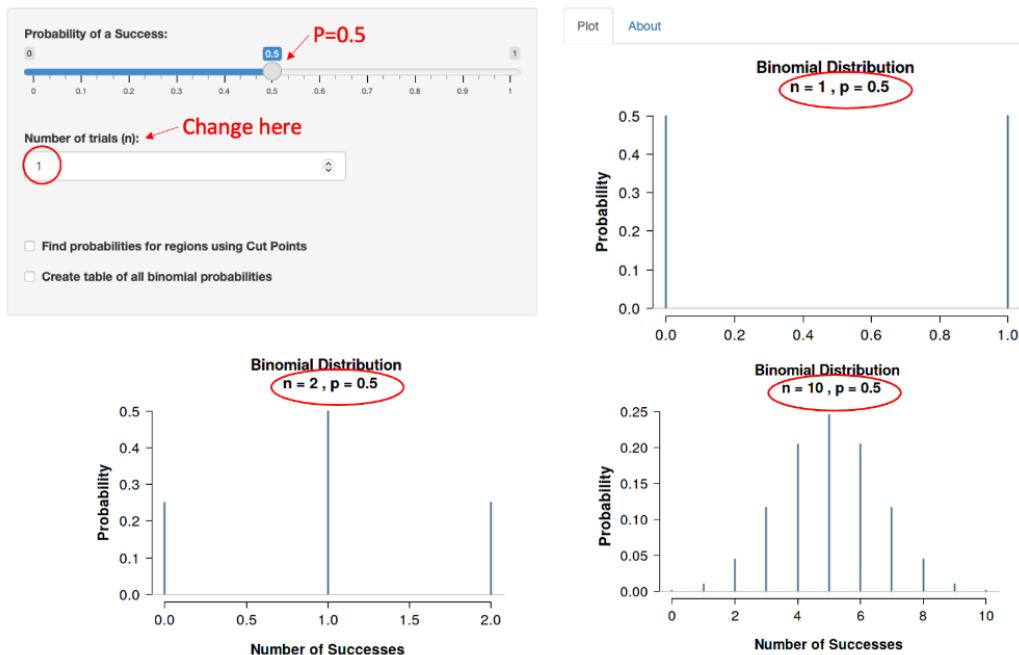


Figure 9.2 – Binomial distribution with  $p = 0.5$

Remember, binomial distribution models the number of successes in a sequence of independent experiments. The probability of success is  $p = 0.50$ . When  $n = 1$ , the number of successes is either 0 or 1 with equal probability. When  $n = 2$ , the number of successes can be 0, 1, and 2. When  $n = 10$ , the number of successes can be 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Let's summarize the values of  $n$ :

- **$n = 1$ :** Bernoulli
- **$n = 2$ :** There are two experiments. The outcomes of flipping the coin two times will be 0 (“success”), 1 (“success”), or 2 (“successes”). The probabilities of getting 0, 1, and 2 are 0.25, 0.50, and 0.25, respectively.
- **$n = 10$ :** The outcomes of flipping the coin 10 times will be 0, 1, ..., 10. Did you notice that the probability distribution becomes a balanced distribution when  $p = 0.5$ ?

Now, let's slide  $p$  to 0.80, which means the chance of success is 80%. If the coin is flipped 10 times, 8 times will be a “success.” The distributions for  $n = 1, 2$ , and  $10$  look like the following. Note that the distribution is a skewed distribution.

## Binomial Distribution Visualization

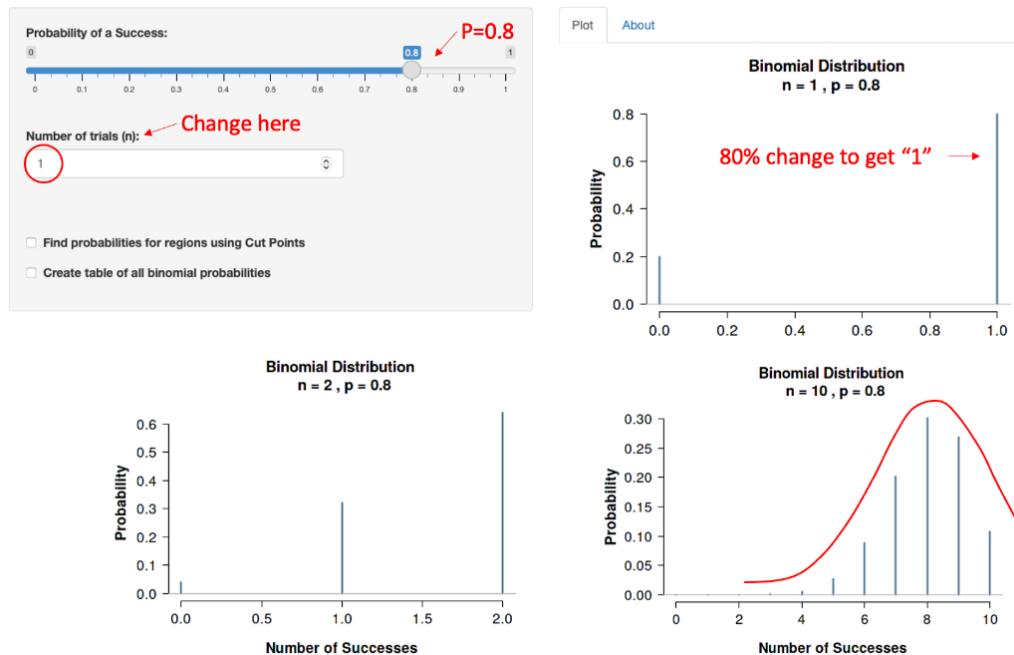


Figure 9.3 – Binomial distribution with  $p = 0.8$

## Plotting it with Python

Let me demonstrate how to use the `stats` module of `scipy`, including how to generate a binomial distribution:

```
import scipy.stats as stats
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
```

The `stats.binom()` function characterizes a binomial distribution. Let's set  $n=10$  for 10 trials and specify the probability with  $p = 0.2$ :

```
binomial=stats.binom(n=10,p=0.2)
```

We will generate samples from this distribution. This is done using `binomial.rvs()`, where `rvs` stands for **random variable samples**. Let's do `size=20`:

```
binomial.rvs(size=20,random_state=123)
```

It generates 20 samples:

```
[3,1,1,2,3,2,5,3,2,2,1,3,2,0,2,3,1,1,2,2]
```

The first value in the output is 3. It means the number of successes is 3. You may have noticed that 2 appears relatively frequently. This is not a surprise because  $p = 0.2$ .

Let's set the size to 1,000:

```
binomial_sample=binomial.rvs(size=1000,random_state=123)
```

Then, let's create a histogram for the 1,000 samples. I will set the range to be between 0 and 10:

```
plt.hist(binomial_sample,
          bins=10,
          range=(0,10),
          density=True,
          align='left',
          rwidth=0.75,
          label="P=0.2",
          alpha=0.75)
```

Let's review these parameters:

- `density`: This is a Boolean (`True` or `False`) that determines whether the histogram should be normalized (`True`) to 1 or not (`False`).
- `align`: This accepts several values to control the alignment behavior. When it is `left` (default), the bins are left-aligned. The left edge of the first bin is aligned with the smallest data point. It can be set as `right` for right-aligned, or `mid` for mid-aligned.
- `rwidth`: This is the `matplotlib` parameter to control the width of the bins. It takes a value between 0 and 1. A value of 1 (the default) means that the bars will touch each other. A value less than 1 (e.g., 0.75) will leave a gap between the bars, creating a clustered or grouped histogram. This is commonly used when you have discrete data and want to show distinct bins.
- `label`: This is the label of the histogram graph.
- `alpha`: This is the `Matplotlib` parameter to control the transparency or opacity of elements within a plot. It takes a value between 0 (completely transparent) and 1 (completely opaque).

Let's add the title, the  $x$  and  $y$  labels, the legend, and the ticks for the  $x$  axis:

```
plt.title("Binomial Distribution")
plt.ylabel("Density")
plt.legend()
t=np.arange(0,10)
plt.xticks(t)
plt.xlabel("Number of successes")
plt.show()
```

The plot looks like this. There are 10 bins because there are 10 values:

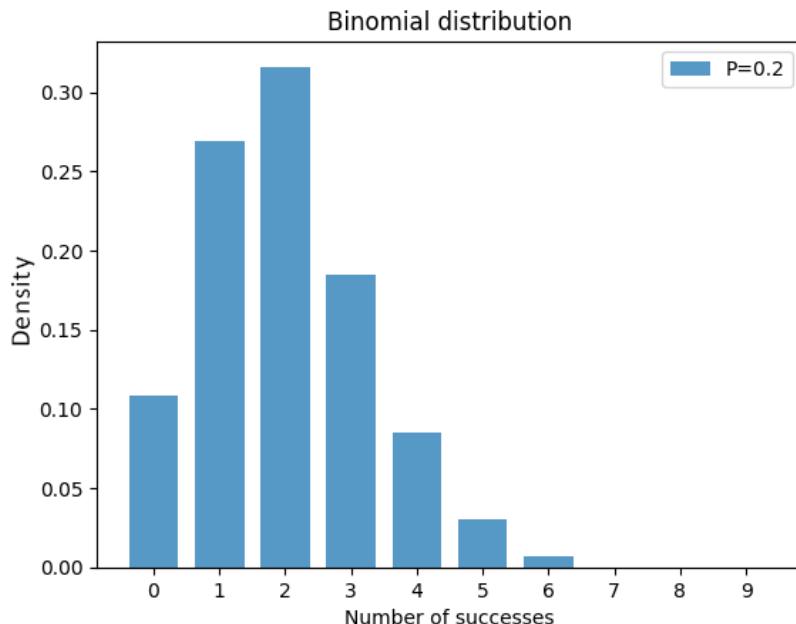


Figure 9.4 – Plotting the binomial distribution  $p=0.2$  with Python

## Fun facts

Not surprisingly, the Swiss mathematician Jacob Bernoulli (1654–1705) [1] is also the inventor of the binomial distribution.

The binomial distribution describes  $n$  independent samples, and each sample has only two choices, such as “yes” and “no.” It seems we can extend the choices to be more than two. That’s the multinomial distribution!

## Multinomial distributions

The Multinomial distribution generalizes the Binomial distribution. A multinomial distribution describes  $n$  independent experiments, and each experiment has more than two choices. Let's see a few real-world examples.

### The real-world examples

In the binomial distribution, we use a coin-flipping example that has two outcomes – 1 and 0. Now, instead of a coin, let's use a dice that has six outcomes. The dice-throwing example is a perfect example of a multinomial distribution.

### The formal definition of a multinomial distribution

The multinomial distribution is an extension of the binomial distribution. It extends Eq. (2) to multiple outcomes, as shown in Eq. (3):

$$P(x_1, \dots, x_k; n, p) = \frac{n!}{x_1! x_2! \dots x_k!} p_1^{x_1} \dots p_k^{x_k} \quad \text{Eq. (3)}$$

Let's suppose a person throws a dice  $n$  times. Let  $p_i$  be the probability of numbers 1 to 6 – that is,  $i = 1, 2, \dots, 6$ . Eq. (3) models the probability of counts for each side of a  $k$ -sided dice rolled  $n$  times.

### What does it look like?

Let's use Eq. (3) to simulate a multinomial distribution. Suppose it is an uneven dice with the probability [1/12, 1/3, 1/12, 1/6, 1/6, 1/6]. (A fair dice should have an equal chance of 1/6 for all sides, but this is a weird dice!) In the following code, I use `stats.multinomial()` to define the probability distribution. Just like the sampling in the binomial distribution, I use `.rvs()` to generate the random variable samples:

```
import numpy as np
import scipy.stats as stats
%matplotlib inline
import matplotlib.pyplot as plt
multinomial=stats.multinomial(
    n=10,
    p=[1/12, 1/3, 1/12, 1/6, 1/6, 1/6])
multinomial.rvs(size=10,random_state=143)
```

The output looks as follows. It has 10 lists because we set  $n=10$ . Each list has six values for the count of a side of the dice:

```
[ [1, 3, 1, 1, 2, 2],
  [2, 1, 0, 1, 4, 2],
```

```
[3, 1, 0, 1, 2, 3],  
[0, 4, 0, 1, 4, 1],  
[1, 1, 0, 0, 5, 3],  
[1, 3, 3, 2, 1, 0],  
[2, 4, 0, 1, 2, 1],  
[0, 5, 1, 1, 2, 1],  
[1, 2, 0, 3, 1, 3],  
[1, 1, 2, 1, 3, 2]]
```

Let's repeat the game 1,000 times ( $n=1000$ ) and store the outcomes as `multinomial_sample`:

```
multinomial_sample=multinomial.rvs(  
    size=1000, random_state=143)
```

It is a list of 1,000 lists:

```
[[1, 3, 1, 1, 2, 2],  
 [2, 1, 0, 1, 4, 2],  
 [3, 1, 0, 1, 2, 3],  
 ...,  
 [1, 3, 1, 3, 1, 1],  
 [1, 4, 0, 2, 2, 1],  
 [2, 3, 1, 1, 1, 2]])
```

Let's plot it out:

```
t=np.arange(0,9)  
plt.hist(multinomial_sample,  
        bins=10,  
        range=(0,9),  
        density=True,  
        align='left',  
        rwidth=0.9,  
        label=['p1=1/12', 'p2=1/3', 'p3=1/12',  
               'p4=1/6', 'p5=1/6', 'p6=1/6'],  
        alpha=0.75)  
plt.title("Multinomial Distribution")  
plt.ylabel("Density")  
plt.legend()  
plt.xticks(t)  
plt.xlabel("Number of successes")  
plt.show()
```

The output is shown in *Figure 9.5*.

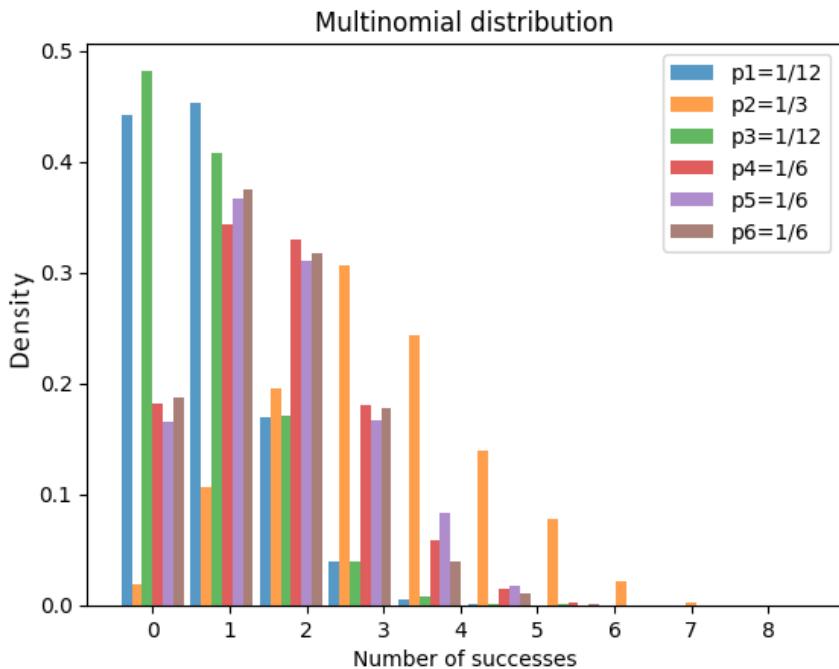


Figure 9.5 – Plot multinomial distribution with Python

## Fun facts

The multinomial distribution was generalized from the binomial distribution by Johann Bernoulli (1667–1748) [2] and Gottfried Leibniz (1646–1716). Johann Bernoulli was the brother of Jacob Bernoulli, and he was 12 years younger than Jacob.

It is quite straightforward to learn the Bernoulli distribution, the binomial distribution, and the multinomial distribution in succession, isn't it? Let's see how easy it is to learn the beta distribution.

## Beta distributions

The beta distribution is a distribution of probabilities that models the uncertainty about the probability of success of an experiment. There are only two outcomes in the beta distribution – “yes” and “no.” We just do not know the actual probability of “yes” or “no” in an experiment. However, we may have prior knowledge to guess their probability. Let's see a real-world example.

## The real-world examples

Suppose you are at a table in a casino that bets on flipping coins. Assume you do not know the probability of whether the coin is 1 = head and 0 = tail, and you do not believe it is a fair coin. You had heard from others that the probability of the unfair coin is 0.52 for 1 = head and 0.48 for 0 = tail. The 0.52 or 0.48 is your “prior belief” in the Bayesian terminology. The outcomes of the coin for “1 = head” in multiple experiments can range from 0.35 to 0.85, and 0.52 is in the same range. The beta distribution provides the distribution of probabilities for the unknown probability. Let’s define it formally.

## The formal definition of a beta distribution

The beta distribution is defined in *Eq. (4)*:

$$f(x; \alpha, \beta) = \text{constant}_2 \cdot x^{\alpha-1} (1-x)^{\beta-1} \quad \text{Eq. (4)}$$

Let’s simplify the binomial distribution in *Eq. (2)* as *Eq. (2a)*:

$$P(x; n, p) = \text{constant}_1 \cdot p^x (1-p)^{n-x} \quad \text{Eq. (2a)}$$

We realize there is no  $p$  in the beta distribution. The binomial distribution wants to know the number of successes ( $x$ ), given  $p$ , while the beta distribution wants to know the probability ( $p$ ) when successes ( $x$ ) happen.

The *constant* terms in both the previous equations are the inverse of the total number of combinations. Their presence is just to count the frequency to probabilities.

What do  $\alpha$  and  $\beta$  represent in *Eq. (4)*? In *Eq. (2)*, the exponents  $x$  and  $n - x$  are the numbers of successes and failures. Similarly,  $\alpha - 1$  and  $\beta - 1$  are the number of successes and the number of failures, respectively.

It is important to point out that the beta distribution is not a direct generalization of the binomial distribution, but there is a relationship between the two distributions. The beta distribution can be used to model certain aspects of the binomial distribution.

Recall that the binomial distribution represents the number of successes in a fixed number of independent Bernoulli trials, where each trial has two possible outcomes – success and failure. It is characterized by two parameters – the number of trials and the probability of success in each trial. Conversely, the beta distribution is typically used to model the probability of success in a binomial experiment. So, while the beta distribution is not a direct generalization of the entire binomial distribution, it is closely related to modeling the probability of success within a binomial experiment.

## What does it look like?

You can use the **beta distribution** calculator (<https://mathlets.org/mathlets/beta-distribution/>) to simulate a beta distribution. *Figure 9.6* shows the PMF of beta distributions with various  $\alpha$  and  $\beta$  values. The  $x$ -axis is a probability between 0 and 1. Let me give the rationale for each case.

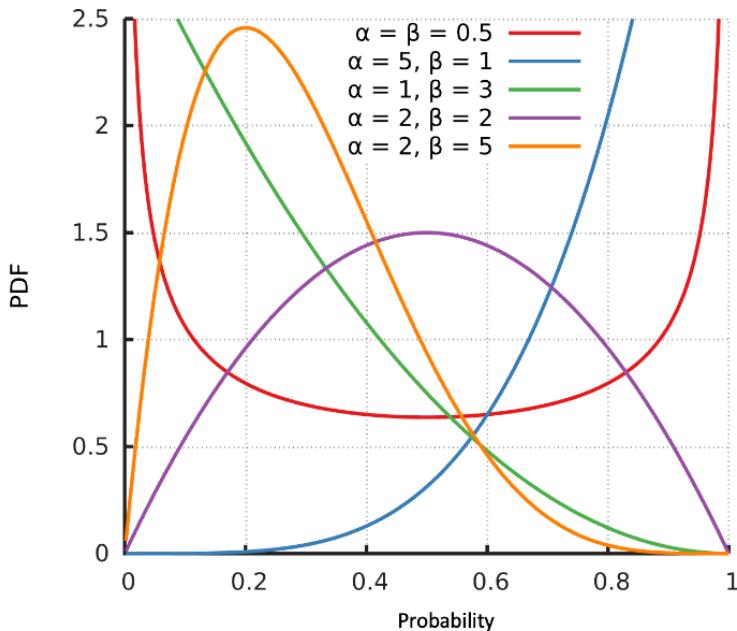


Figure 9.6 – Beta distribution (source [https://en.wikipedia.org/wiki/Beta\\_distribution](https://en.wikipedia.org/wiki/Beta_distribution))

When  $\alpha = 0.5$ ,  $\beta = 0.5$ , and the PMF is a bimodal curve (the red curve in the graph). As mentioned earlier,  $\alpha - 1$  and  $\beta - 1$  can be considered the numbers of successes and failures, respectively. Here,  $\alpha$  and  $\beta$  are not even 1, meaning the number of experiments is very rare. In a rare event, how can we know the average outcomes, except either “0 = failure” or “1 = success”? Hence, the distribution is bimodal.

When  $\alpha = 5$ ,  $\beta = 1$ , and the PMF is an increasing curve towards 1 (the blue curve in the graph). There are more successes than failures ( $\alpha > \beta$ ). The distribution of the success probabilities moves toward 1.

Let's compare the purple curve when  $(\alpha = 2, \beta = 2)$  and the orange curve when  $(\alpha = 2, \beta = 5)$ . The orange curve is skewed to the right when compared to the purple curve. This is because  $\beta - 1$  represents the number of failures. When there are more failures, the distribution of the success probabilities moves to the left.

Let's use `stats` to generate the beta distributions and `matplotlib` to plot them:

```
import numpy as np
import scipy.stats as stats
%matplotlib inline
import matplotlib.pyplot as plt
```

First, let's start with equal values for  $\alpha$  and  $\beta$ :

$$\alpha = 0.5, \beta = 0.5$$

The syntax is similar to what we did in the binomial distributions:

$$a, b = 0.5, 0.5$$

```
my_beta = stats.beta(a, b)
```

Then, I will create 10 random variable samples:

```
my_beta.rvs(size=10, random_state=143)
```

The outcomes are 10 probabilities:

```
[0.82485573, 0.54937533, 0.93437335, 0.58656755, 0.99660327,
0.09302594, 0.98676029, 0.74773465, 0.88870632, 0.98241934]
```

Let me create 1,000 samples:

```
beta_sample=my_beta.rvs(size=1000,random_state=143)
beta_sample
```

You will see 1,000 values:

```
[8.24855734e-01, 5.49375329e-01, 9.34373355e-01, 5.86567546e-
01, 9.96603266e-01, 9.30259374e-02, 9.86760293e-01, 7.47734652e-
01, 8.88706324e-01, 9.82419341e-01, 9.72323678e-01, 6.96615476e-
02, 9.25087251e-01, 1.13238292e-05, 9.72460557e-01, 7.44102421e-01,
1.62609707e-01, 9.50947906e-01, 1.22029377e-01, 3.51031459e-03,
2.32018981e-01, ... ]
```

Then, we can plot the distribution of the probabilities:

```
t=np.arange(0,2)
plt.hist(beta_sample,
          bins=100,
          range=(0,1),
          density=True,
          align='left',
          rwidth=1,
```

```

label=['a = 0.5, b = 0.5'],
alpha=0.75)
plt.title("Beta Distribution")
plt.ylabel("Density")
plt.legend()
plt.xticks(t)
plt.xlabel("Probability")
plt.show()

```

The plot is shown in *Figure 9.7*. Can you see the similarity of *Figure 9.7* to the red curve in *Figure 9.6*?

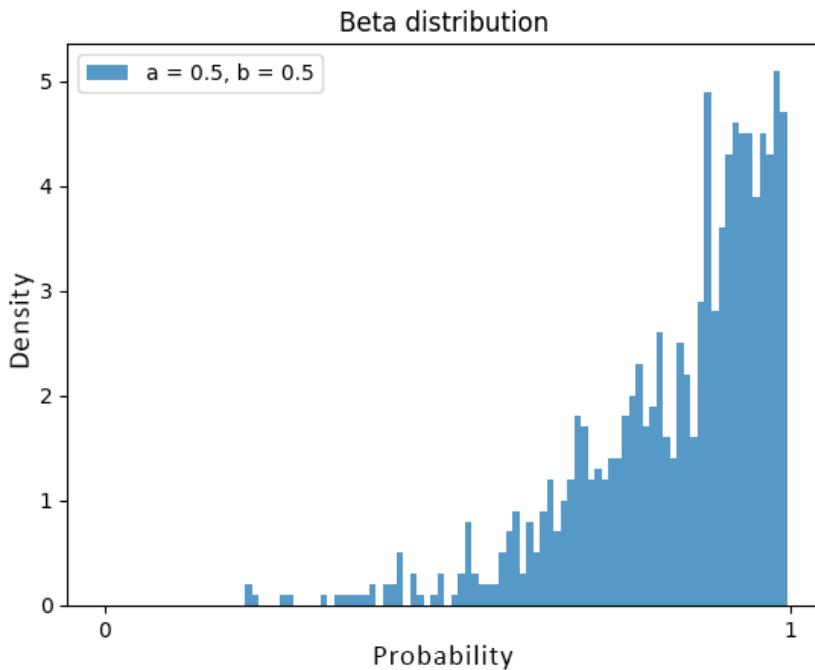


Figure 9.7 – Beta distribution with  $\alpha = 0.5$  and  $\beta = 0.5$

Because the values for  $\alpha$  and  $\beta$  are the same, the distribution in *Figure 9.7* looks very balanced and resembles the red line ( $\alpha = \beta = 0.5$ ) in *Figure 9.6*. Now, let's test another case.

**$\alpha = 5$  and  $\beta = 1$**

Let's produce a plot for the blue curve in *Figure 9.6*:

```

a, b = 5, 1
my_beta = stats.beta(a, b)
beta_sample=my_beta.rvs(size=1000,random_state=143)

```

```
t=np.arange(0,2)
plt.hist(beta_sample,
         bins=100,
         range=(0,1),
         density=True,
         align='left',
         rwidth=1,
         label=['a = 5, b = 1'],
         alpha=0.75)
plt.title("Beta Distribution")
plt.ylabel("Density")
plt.legend()
plt.xticks(t)
plt.xlabel("Probability")
plt.show()
```

The plot is shown in *Figure 9.8*. Again, did you notice that *Figure 9.8* resembles the blue curve in *Figure 9.6*?

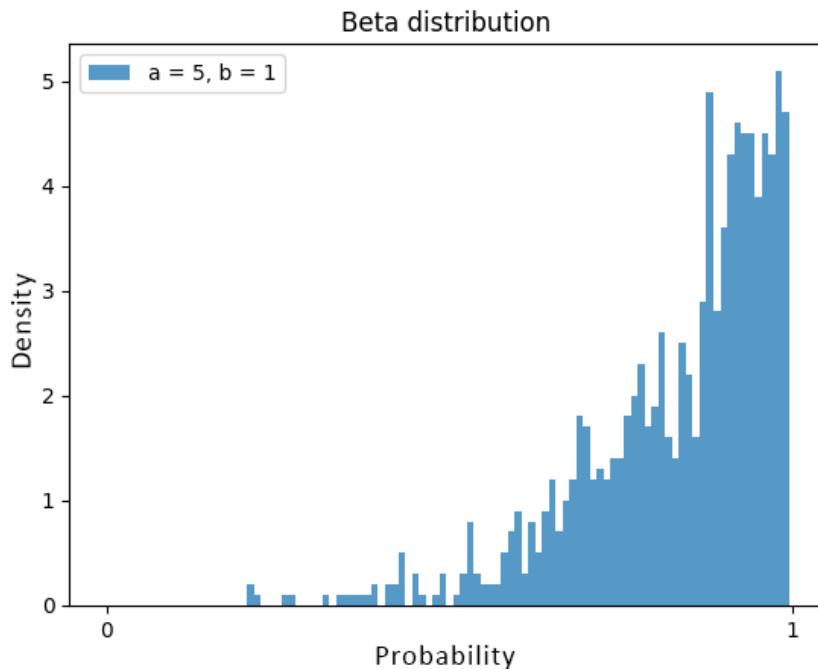


Figure 9.8 – Beta distribution with  $\alpha = 5$  and  $\beta = 1$

$$\alpha = 2, \beta = 2$$

Let's produce the purple curve in *Figure 9.6*:

```
a, b = 2, 2
my_beta = stats.beta(a, b)
beta_sample=my_beta.rvs(size=1000, random_state=143)
t=np.arange(0,2)
plt.hist(beta_sample,
         bins=100,
         range=(0,1),
         density=True,
         align='left',
         rwidth=1,
         label=['a = 2, b = 2'],
         alpha=0.75)
plt.title("Beta Distribution")
plt.ylabel("Density")
plt.legend()
plt.xticks(t)
plt.xlabel("Probability")
plt.show()
```

*Figure 9.9* shows the plot. Can you see how *Figure 9.9* resembles the purple curve in *Figure 9.6*?

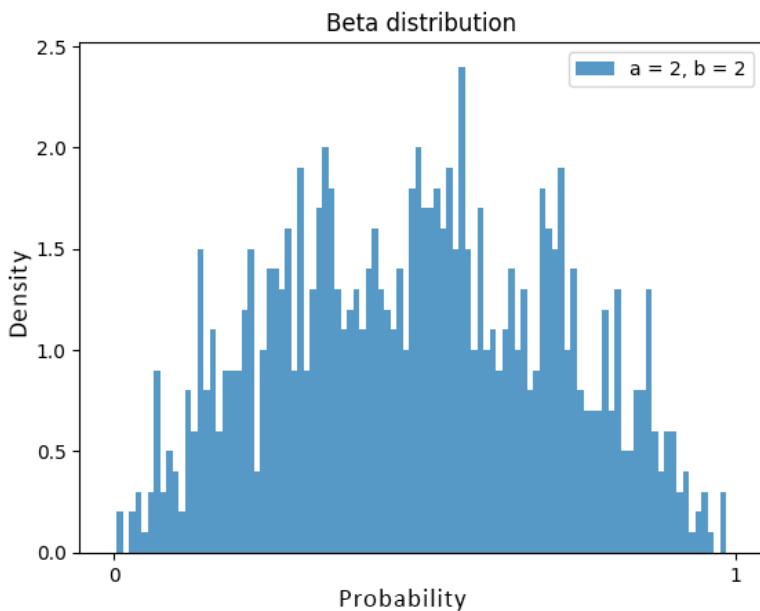


Figure 9.9 – Beta distribution with  $\alpha = 2$  and  $\beta = 2$

We mentioned the word “conjugate” at the beginning of this chapter. Let’s learn what its formal definition is.

## The beta distribution in Bayesian inference

In Bayesian statistics, when we update our beliefs about a parameter or a random variable based on new evidence or data, we use a prior distribution to represent our initial beliefs and a posterior distribution to represent our updated beliefs, after incorporating the new information. These distributions are related through Bayes’ theorem. Let me give a more detailed explanation with a standard Bayes’ theorem. Assume there are two variables X and Y, Bayes’ theorem says that the **posterior probability**  $p(Y|X)$  can be represented as follows:

$$p(Y|X) = \frac{p(X|Y) \cdot p(Y)}{p(X)}$$

Here,  $p(Y)$  is called the **prior probability**. It is the probability distribution that represents any prior belief or known probability.  $p(X|Y)$  is called the **likelihood**.  $p(X)$  is called the **evidence** because it is observed. Bayes’ theorem says the posterior probability is the prior probability times the likelihood of the observed data.

The idea of conjugate distributions comes into play when specific types of prior and posterior distributions have a special mathematical relationship. In particular, if the prior distribution and the posterior distribution both belong to the same parametric family, we say they are **conjugate distributions**. In other words, in such cases, the prior distribution and the posterior distribution share the same functional form but just have different parameter values. The fact that two distributions belong to the same parametric family makes calculations tractable. They can be calculated easily in a closed form without the need for complex numerical methods.

The beta distribution is the conjugate prior probability distribution for a binomial distribution. Thus, the posterior distribution is guaranteed to have the same functional form as the prior, making calculations tractable.

### Fun fact

Who invented beta distribution? (Here’s a hint: Bayesian inference.) The answer is the Rev. Thomas Bayes (1701–1761) [3].

So far, we have learned about the Bernoulli distribution, binomial distribution, multinomial distribution, and beta distribution. Let’s have a quick summary of the four distributions:

- The Bernoulli distribution is the special case ( $n = 1$ ) of the binomial distribution
- The binomial distribution models two choices (such as the “head” and “tail” sides of a coin), and a multinomial distribution models multiple choices (such as the sides of a dice)

- The beta distribution is related to the binomial distribution when we do not know the exact probability of the two choices (“yes” and “no”) but we have some prior knowledge about the probability of the two choices

Since the beta distribution models scenarios where there are only two choices, can we extend it to multiple choices? You got it. The Dirichlet distribution does exactly that!

## Dirichlet distributions

The beta distribution is a distribution of probabilities for two outcomes – 1 and 0. The Dirichlet distribution is an extension of the beta distribution to multiple outcomes. The beta distribution is a distribution of *two probabilities*, and the Dirichlet distribution is a distribution of *k-probabilities*.

### Real-world examples

When we do not know the actual probabilities of the  $k^{\text{th}}$  choice, we can form a prior guess for them. Again, this prior knowledge is called the **prior probability** in Bayesian terminology. Suppose you are at a table in a casino that bets on rolling a dice and you do not know the probabilities of “1”, “2”, “3”, “4”, “5”, or “6.” Suppose the casino is reputable, so you believe the probabilities of all six sides are equal as  $[1/6, 1/6, 1/6, 1/6, 1/6, 1/6]$ . This becomes your “prior belief” in Bayesian inference. The outcomes of the dice in multiple experiments for each side can range from 10% to 25%, while the average is  $1/6 = 16.7\%$ . The Dirichlet distribution, therefore, provides the distribution of the 6-probabilities.

The Dirichlet distribution is especially useful in an LDA model in NLP. The LDA model is a topic modeling algorithm that classifies a document as a list of topics with probabilities. For example, if there are one million documents for three topics, “politics,” “science,” and “business,” an LDA model can classify a document to the topics with probability terms. It may say a document is 90% “politics,” 8% “business,” and 2% “science.” There can be  $k$  multiple topics. Hence, the Dirichlet distribution describes the probabilities of a document for the  $k$  topics.

### The formal definition of a Dirichlet distribution

The Dirichlet distribution is defined in Eq. (5):

$$D(x_1, \dots, x_k ; \alpha_1, \dots, \alpha_k) = \text{constant}_3 \cdot x_1^{\alpha_1-1} \cdots x_k^{\alpha_k-1} \quad \text{Eq. (5)}$$

Sometimes, a Dirichlet distribution can be written as **Dir(alpha)**, where **alpha** could be a scalar or a vector of real number  $\text{Dir}(\alpha)$ , and alpha could be a scalar or a vector of positive real numbers. Let's see how *Eq. (5)* was developed. If we simplify the beta distribution in *Eq. (4)* to *Eq. (4a)* as shown next, we realize *Eq. (5)* is just an extension of *Eq. (4a)* to  $k$  outcomes:

$$f(x; \alpha, \beta) = \text{constant}_2 \cdot x^{\alpha-1} (1-x)^{\beta-1} \quad \text{Eq. (4a)}$$

The *constant* terms in the two equations are the inverse of the total number of combinations. Their presence is just to make count frequency to probabilities.

In *Eq. (4a)*, the parameters  $\alpha - 1$  and  $\beta - 1$  represent the number of successes and the number of failures. Likewise, the parameters  $\alpha_k$  represents the number of successes for the outcome  $x_k$ .

How to visualize the Dirichlet distribution? It is not easy to visualize because the outcomes can be any high dimension. A very useful concept called the “probability simplex” was invented. Let's learn what it is.

## What is a simplex?

Simplexes provide an exceptional solution when we try to visualize high-dimensional data. The simplex is so-called because it represents the simplest possible shape in any dimension. *Figure 9.10* shows how a simplex represents different dimensions:

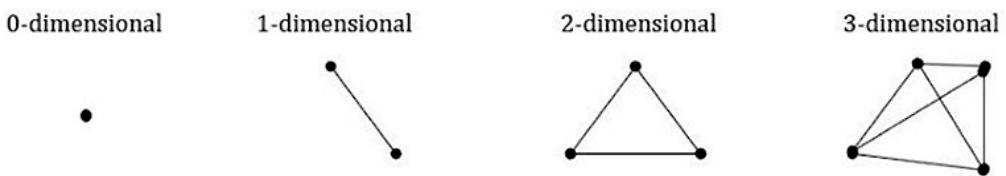


Figure 9.10 – Types of simplex

- A 0-dimensional simplex is a point
- A 1-dimensional simplex is a line segment
- A 2-dimensional simplex is a triangle
- A 3-dimensional simplex is a tetrahedron [6]

An  $n$ -dimensional simplex looks like the following:

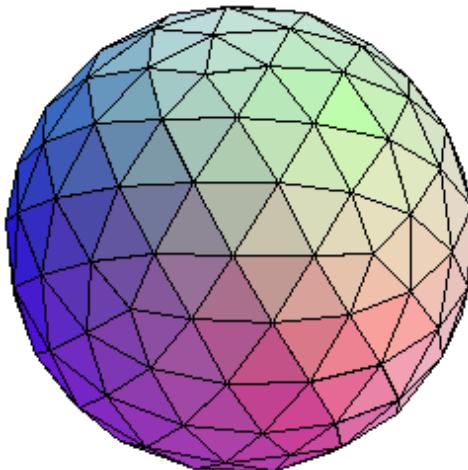


Figure 9.11 – A simplex for a high dimension (image credit: <https://plus.maths.org/content/mathsmind-simplices-atoms-topology>)

## What does the Dirichlet distribution look like?

I will present the Dirichlet distribution with three parameters,  $a_1$ ,  $a_2$ , and  $a_3$ . The three-parameter plot is a triangle. First, let's load the Python library for 3D graphing:

```
import numpy as np
from scipy import stats
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

**$\alpha = [1, 1, 1]$**

Let's plot the Dirichlet distribution for  $[1, 1, 1]$ . When the values of the parameters are the same, the Dirichlet distribution becomes symmetric. The syntax is similar to what we did in the binomial and the beta distributions. I will create 500 random variable samples. The outcomes are 500 lists, and each list has three probabilities:

```
alpha = [1, 1, 1]
D = stats.dirichlet(alpha).rvs(500)
D
```

The output is a list of lists:

```
[ [0.64976022, 0.21789822, 0.13234156],  
[0.78147747, 0.01743882, 0.20108371],  
[0.14826321, 0.74779572, 0.10394107],  
...  
[0.07603431, 0.57609057, 0.34787512],  
[0.10152252, 0.88977604, 0.00870143],  
[0.29219169, 0.04465732, 0.66315099]]
```

We will plot the Dirichlet distributions with different  $\alpha$  values. I have made a utility function for the plotting:

```
Def plot_Dirichlet(alpha):  
    D = stats.dirichlet(alpha).rvs(500)  
    fig = plt.figure(figsize=(6.8, 6), dpi=120)  
    ax = plt.gca(projection='3d')  
    plt.title(r'$\alpha$ = {}'.format(alpha))  
    ax.scatter(D[:, 0], D[:, 1], D[:, 2], c='b')  
    ax.view_init(azim=30)  
    ax.set_xlabel(r'$\theta_1$')  
    ax.set_ylabel(r'$\theta_2$')  
    ax.set_zlabel(r'$\theta_3$')  
    plt.show()
```

When  $\alpha = 1$ , the symmetric Dirichlet distribution becomes a **uniform** distribution over the simplex, as shown in *Figure 9.12*.

```
plot_Dirichlet([1,1,1])
```

The plot is shown in *Figure 9.12*:

$$\alpha = [1, 1, 1]$$

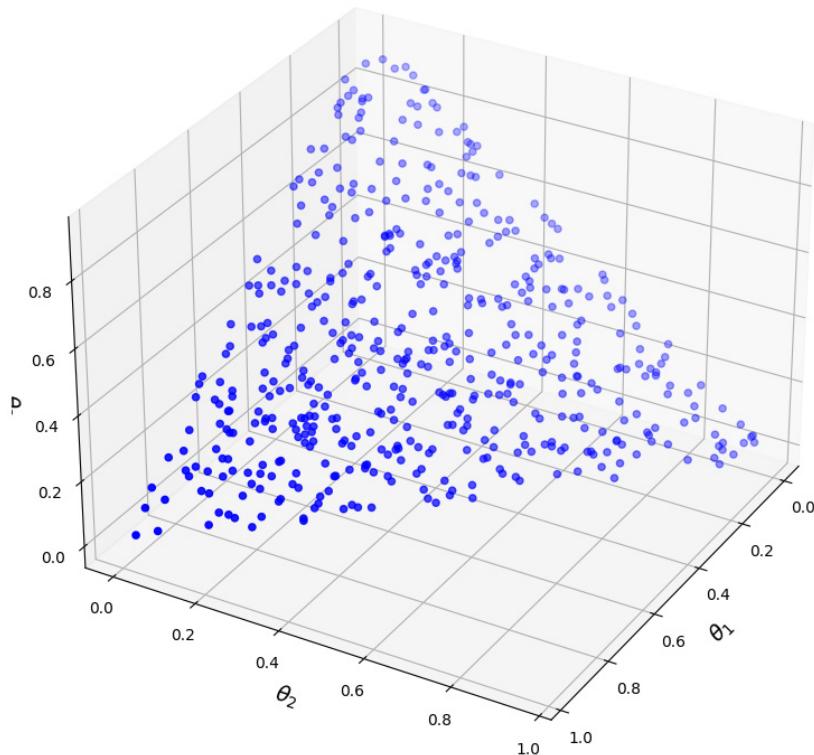


Figure 9.12 –  $\alpha = [1, 1, 1]$

$$\alpha = [1, 1, 5]$$

What happens if  $a1 = a2 = 1$  but  $a3$  is a higher value, such as 5? It means the number of successes in the third dimension is higher than the other two dimensions. It is expected the data points will be distributed closer to the third dimension, as shown in *Figure 9.13*:

```
plot_Dirichlet([1, 1, 5])
```

Figure 9.13 shows the output:

$$\alpha = [1, 1, 5]$$

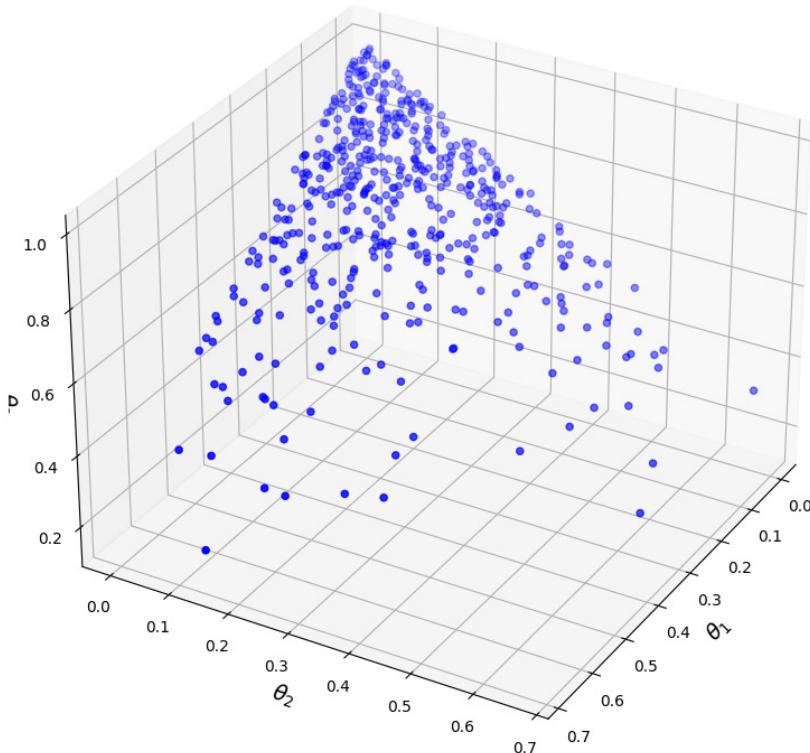


Figure 9.13 –  $\alpha = [1, 1, 5]$

$$\alpha = [0.1, 0.1, 0.1]$$

When  $a1 = a2 = a3 = 0.1$ , the three values are equal and lower than 1. The data points fall on the edges evenly, as shown in Figure 9.14:

```
plot_Dirichlet([0.1, 0.1, 0.1])
```

The output is as follows:

$$\alpha = [0.1, 0.1, 0.1]$$

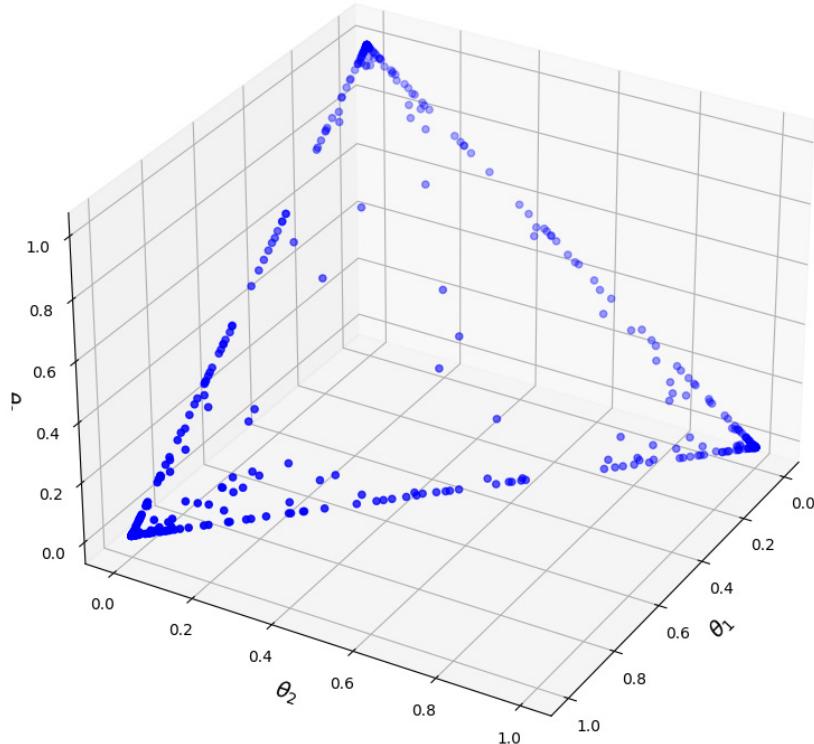


Figure 9.14 –  $\alpha = [0.1, 0.1, 0.1]$

$$\alpha = [5, 5, 5]$$

When  $a1 = a2 = a3 = 5$ , the three values are equal and higher than 1. The data points concentrate in the middle of the simplex, as shown in *Figure 9.15*.

```
plot_Dirichlet([5, 5, 5])
```

Figure 9.15 shows the output:

$$\alpha = [5, 5, 5]$$

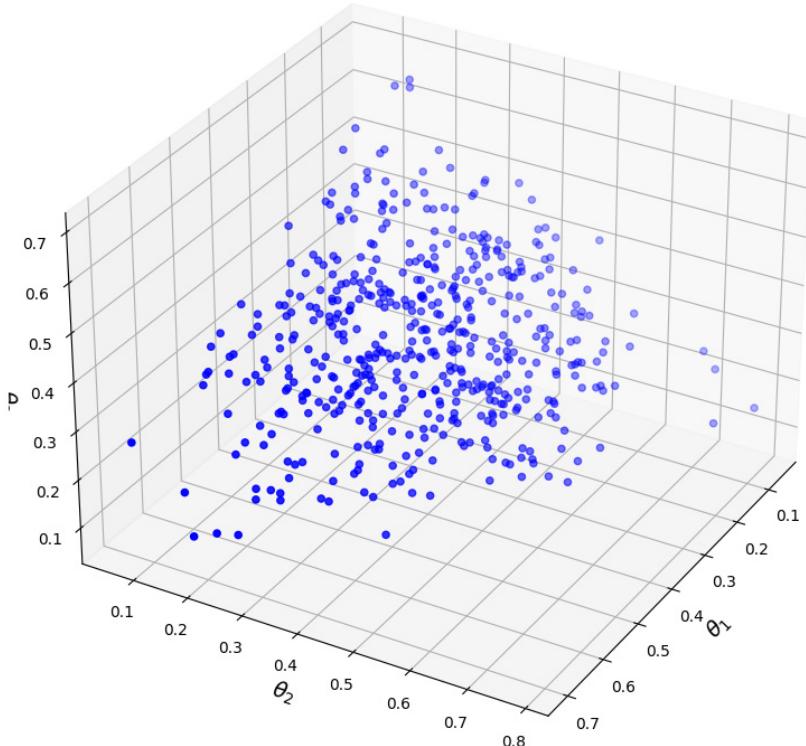


Figure 9.15 –  $\alpha = [5,5,5]$

We've learned the concept of conjugate distributions in the beta distributions. Let's understand how it relates to the Dirichlet distribution.

### The Dirichlet distribution in Bayesian inference

In Bayesian inference, the posterior probability is the prior probability times the likelihood of the observed data. If the prior and the posterior belong to the same parametric family, then the prior and the posterior are called conjugate distributions. Dirichlet distributions are commonly used as the prior in Bayesian inference. It is the conjugate prior of the categorical distribution and multinomial distribution.

## Fun fact

Johann Peter Gustav Lejeune Dirichlet [5], a German mathematician (1805–1859), was the inventor of the Dirichlet distribution.

## Summary

The family of the discrete probability distributions is better understood when presented together. I believe you have gained a comprehensive understanding with this approach. To focus on the applications, we learned each distribution with its real-world examples and then its definition, and then we plotted it with Python. We learned five discrete distributions in sequence – the Bernoulli distribution, the binomial distribution, the multinomial distribution, the beta distribution, and the Dirichlet distribution.

The knowledge of Dirichlet distribution will give you the confidence to learn about LDA in the next chapter.

## Questions

1. Describe a Bernoulli distribution.
2. Describe a binomial distribution and its relationship with a Bernoulli distribution.
3. Describe a multinomial distribution and its relationship with a binomial distribution.
4. Describe a beta distribution and its relationship with a binomial distribution.
5. Describe a Dirichlet distribution and its relationship with a beta distribution.
6. What is a simplex?
7. What is a conjugate distribution in Bayesian inference?

## References

- Jacob Bernoulli: [https://en.wikipedia.org/wiki/Jacob\\_Bernoulli](https://en.wikipedia.org/wiki/Jacob_Bernoulli)
- Johann Bernoulli: [https://mathshistory.st-andrews.ac.uk/Biographies/Bernoulli\\_Johann/](https://mathshistory.st-andrews.ac.uk/Biographies/Bernoulli_Johann/)
- Gottfried Leibniz: [https://en.wikipedia.org/wiki/Gottfried\\_Wilhelm\\_Leibniz](https://en.wikipedia.org/wiki/Gottfried_Wilhelm_Leibniz)
- Thomas Bayes: [https://en.wikipedia.org/wiki/Thomas\\_Bayes](https://en.wikipedia.org/wiki/Thomas_Bayes)
- Johann Peter Gustav Lejeune Dirichlet: [https://en.wikipedia.org/wiki/Peter\\_Gustav\\_Lejeune\\_Dirichlet](https://en.wikipedia.org/wiki/Peter_Gustav_Lejeune_Dirichlet)
- Simplex: <https://en.wikipedia.org/wiki/Simplex>

# 10

## Latent Dirichlet Allocation

When we write an article, we develop it according to a theme or topic and we use certain words from that topic. We may have sub-topics and use the words for the sub-topics too. When we classify articles into topic piles, we recognize specific words and then tag them so that we can place them into topics. An article may have one topic and other sub-topics, so it is possible to tag an article to multiple topics. **Latent Dirichlet Allocation (LDA)** is designed to discover abstract topics in a document. This makes LDA a powerful model that can tag an article with multiple topics.

LDA is the core technique in NLP and is worth investigating thoroughly. It has enabled many commercial products. The knowledge of LDA, such as its architecture, its use of the generative modeling process, and Dirichlet distribution, are transferable to other models. For these reasons, this book has dedicated four chapters to LDA: *Chapter 9, Understanding Discrete Distributions* which explains the Dirichlet distribution; *Chapter 10, Latent Dirichlet Allocation*, which walks through the model architecture and its generative process; *Chapter 11, LDA Modeling*, which provides relevant code examples; and *Chapter 12, LDA Visualization*, which discusses designing a visual tool when the output content is very rich and how to communicate your LDA model outcome.

LDA also involves having background knowledge of certain aspects, including generative modeling, Bayes' theorem, variational **Expectation-Maximization (E-M)**, and Jensen's inequality. To help you navigate LDA effectively, this chapter has divided LDA into a sequence of learning objectives:

- What is generative modeling?
- Understanding the idea behind LDA
- Understanding the structure of LDA
- Variational inference
- Variational E-M
- Variational E-M versus Gibbs sampling

After reading this chapter, you will be equipped to explain the complex structure of LDA and appreciate its generative process. Let's start learning about LDA!

## What is generative modeling?

In statistics, there is a large field called **generative probabilistic modeling** that contrasts with **discriminative modeling**. You may have used discriminative modeling either knowingly or unknowingly. So, let's start by defining discriminative modeling.

### Discriminative modeling

If you build a statistical model such as a regression model, you are already using the *discriminative* modeling approach. Let's use logistic regression,  $Y = a + bX$ , as an example. The parameters,  $a, b$ , are to be estimated.  $Y = a + bX$  means “given the parameters,  $a, b$ , what is the prediction when the value of  $X$  is  $x$ ?” or  $p(X = x|a, b)$ . This discriminative modeling process applies to any classification modeling, such as decision trees, random forest, gradient boosting, and others. Formally, in a discriminative process, we do the following:

- Assume some functional form for  $p(Y|X)$
- Estimate parameters of  $p(Y|X)$  from the training data

### Generative modeling

In contrast, the generative modeling approach *generates* data and then uses it to find the most likely values for the parameters that the data was generated from. In this type of data science problem, we usually observe  $P(Y)$  and  $P(X|Y)$ . Suppose there is a rule that  $X$  is presented given a  $Y$  value, but we do not know the rule. We assume there is a formula, such as a linear relationship with unknown parameters, and that  $X$  is generated given  $Y$ . Hence, the unknown parameters are what we want to estimate. This generative modeling process can be described formally in this way:

1. Assume some formula for  $P(Y), P(X|Y)$ . Notice that it is the opposite of  $P(Y|X)$ .
2. Estimate the unknown parameters for  $P(Y), P(X|Y)$  from the training data.
3. Use Bayes' theorem to calculate  $P(Y|X)$ .

Both the discriminative process and the generative process are there to help us find  $P(Y|X)$ . So, why don't we just use the discriminative process? The answer is that we only observe  $P(X|Y)$ , or only have some parts of the data. Even so, we can still estimate the unknown parameters to find  $P(Y|X)$ .

### Bayes' theorem

How do we get  $P(Y|X)$  from the known  $P(X|Y)$ ? This is what Bayes' theorem is. The relationship between  $P(Y|X)$  and  $P(X|Y)$  was figured out mathematically by an English Presbyterian minister, statistician, and philosopher, Reverend Thomas Bayes (c. 1701–1761):

$$P(Y|X) = \frac{P(X|Y) \cdot p(Y)}{p(X)} = P(Y) \cdot \frac{P(X|Y)}{P(X)} \quad Eq. (5)$$

Here,  $P(Y)$  is called the **prior probability**. It refers to any prior belief or known probability. It is the estimate of the hypothesis,  $Y$ , before the new data,  $X$ , is observed.  $p(X)$  corresponds to new data that was not used in computing the prior probability.  $P(Y|X)$  is called the **posterior probability**. It is the probability of  $Y$  given  $X$  is observed and is what we want to estimate.  $P(X|Y)$  is the probability of observing  $X$  given  $Y$ , so it is called the **likelihood**.  $P(X)$  is called the **evidence** because it is observed.  $P(X|Y)/p(X)$ , which is the ratio between the likelihood and the evidence, is sometimes called the **likelihood ratio**. Using these terms, Bayes' theorem can be rephrased as “the posterior probability equals the prior probability times the likelihood ratio.”

Let's look at an example. I will explain the known values to you, and the unknown that you want to find out by using Bayes' theorem. Let's assume that at a party of 100 people, you tally how many males and females there are, and how many wear glasses and don't. You record the numbers in a box, as shown in *Figure 10.1*:

	X	$\sim X$	
$\sim Y$	Glasses	No	
Y	Male		
Male	10	40	$P(Y) = \frac{10 + 40}{100} = 0.5$
Female	20	30	$P(X) = \frac{10 + 20}{100} = 0.3$

Figure 10.1 – Bayes' theorem

Here are the known values:

- Probability of males,  $P(Y) = 0.5$
- Probability of wearing glasses,  $P(X) = 0.3$
- The conditional probability of those who wear glasses in the male population,  

$$P(X|Y) = \frac{P(X \cap Y)}{P(Y)} = \frac{10}{10 + 40} = 0.2$$

Here are the unknown values:

- The conditional probability of males in the population that wear glasses,  $P(Y|X)$

By using Bayes' theorem, we can easily get the following formula:

$$P(Y|X) = \frac{P(X|Y) \cdot P(Y)}{P(X)} = \frac{0.2 \cdot 0.5}{0.3} = 0.33$$

The example in *Figure 10.1* does not have any parameters to be estimated. In most cases, there are parameters to be solved, especially when some of the data is unobserved. **Expectation-Maximization (E-M)** is often used in Bayesian inference. It is helpful to learn about E-M now because, later, we will learn how LDA uses Bayes' theorem as well as the E-M process to obtain the optimized estimations.

## Expectation-Maximization (EM)

In the Bayesian setting, the posterior distribution is proportional to the product of the likelihood function (how likely the data is under different parameter values) and the prior distribution (prior beliefs about the parameters). However, there are latent or unobserved variables that are not directly measurable from the data. These variables introduce additional complexity to parameter estimation. Models such as **Gaussian mixture models (GMMs)** are the typical applications of E-M.

To solve for the unknown parameters, the E-M algorithm alternates between two main steps: the **Expectation step (E-step)** and the **Maximization step (M-step)**:

1. In the **E-step**, the E-M algorithm calculates the expected values of the latent variables given the current parameter estimates. This step incorporates the observed data and the current parameter values to provide an estimate of the latent variables' values. It assumes random values to start with as the initial values.
2. In the **M-step**, the E-M algorithm updates the model parameters to find the maximum likelihood estimate of model parameters based on the estimates from the E-step. This step considers both the observed data and the estimated values of the latent variables.

This process is shown in *Figure 10.2*:

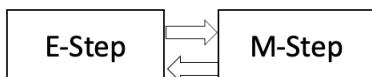


Figure 10.2 – Expectation-Maximization

The E-M process iterates between the E and M steps and updates the parameters in M-step. It checks for convergence by comparing the change in parameter estimates between the iterations. If the change is below the predefined threshold or if the maximum number of iterations is reached, the algorithm stops.

What we've learned so far has covered the necessary background knowledge for LDA. So, let's start to understand the structure of LDA.

## Understanding the idea behind LDA

LDA assumes each topic has a distinctive distribution of words. It analyzes the frequency of words to discover hidden topics and identifies the probability of a word belonging to a topic. Its generative modeling approach is a unique feature. LDA considers hidden topics as templates in a printing shop. Each topic template has a set of words. An article is *generated* from a topic template or a mixture of topic templates. This approach is even described in its title. It contains the word *latent* because it finds the hidden topics in the latent space. The word *Dirichlet* refers to the assumption that both the distribution of topics in a document and the distribution of words in a topic follow Dirichlet distributions. *Allocation* means the mixture of topics and words is generated from the topic templates and allocated to a document.

Typically, a document has some portion of text on a topic and some on other topics. LDA tags a document into several topics with probabilities. I created *Figure 10.3* to show you the intuition of LDA:

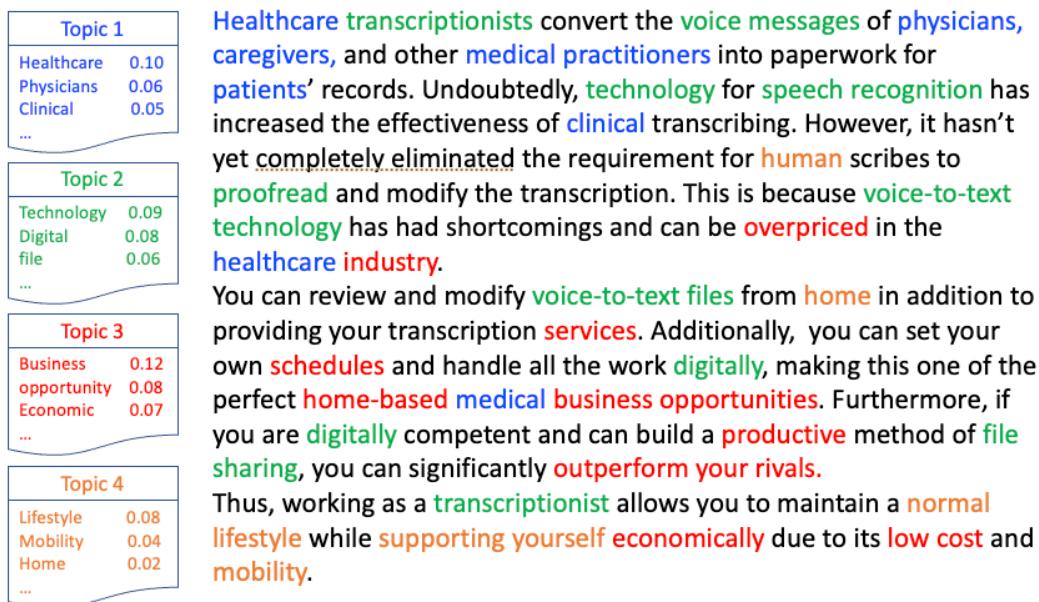


Figure 10.3 – An LDA topic example

This document describes technology innovation in medical services and how it will provide more business opportunities and change the lifestyle of the service providers. LDA dissects this document into four topics, as shown on the left of *Figure 10.3*. *Topic 1* is about healthcare, *Topic 2* is about technology, *Topic 3* is about business, and *Topic 4* is about lifestyle. Various words are highlighted in different colors to indicate which topics they relate to.

This example illustrates several assumptions of LDA. LDA assumes that each document is associated with a distribution of topics, and each topic is associated with a distribution of words. Hence, LDA assigns each document a distribution of topics. It does not force a document to one and only one topic. Notice that the distribution of topics and the distribution of words of each topic are all hidden or *latent*. The purpose of topic modeling is to discover the topic structure from the known information, which is the distribution of words in each document.

## Dirichlet distribution of topics

Now, let's consider a three-topic example. *Figure 10.4* shows the relationships of three topics and five hypothetical documents. *Docs 1, 2, and 3, are close to Topics 1, 2, and 3 respectively. Doc 4 is a mixture of Topics 1, 2, and 3. Doc 5 is a mixture of Topics 2 and 3.*

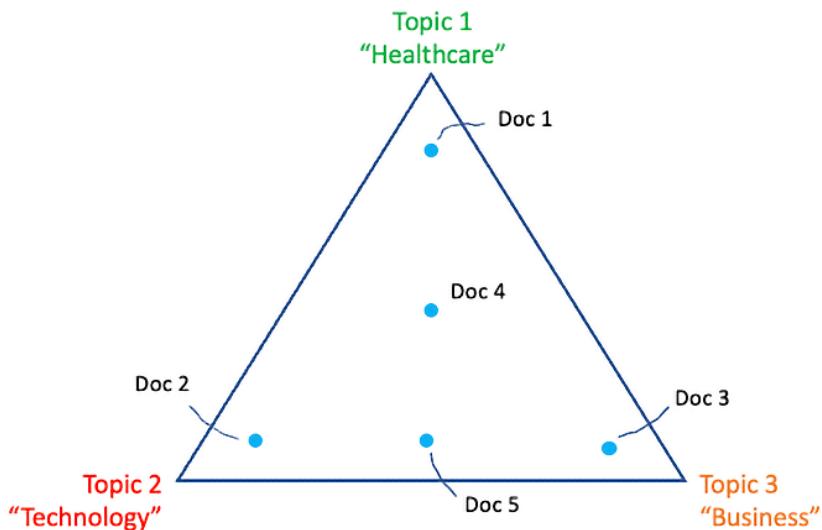


Figure 10.4 – A document is a mixture of latent topics

The preceding figure illustrates that a document can be modeled by LDA as a mixture of topics. This perspective will be helpful as we move on to the next section on the structure of LDA.

## Understanding the structure of LDA

LDA assumes a corpus is a collection of  $D$  documents, and each document is a mixture of  $k$  topics and the word distributions of those topics. Let's imagine a document word "plate" that generates a document,  $d$ , along with its corresponding topics and bag of words. This printing plate is shown in *Figure 10.5*. Let's read it from left to right:

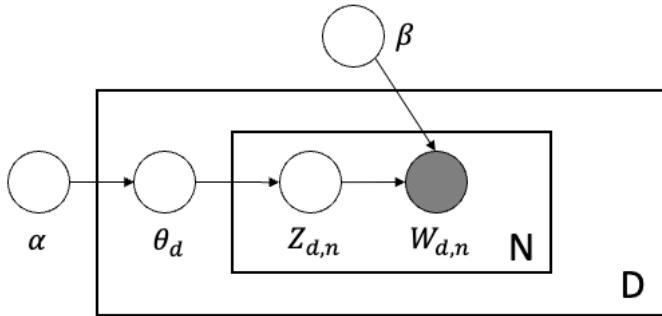


Figure 10.5 – Graphical representation of LDA

First, the outer rectangle,  $D$ , represents the collection of  $D$  documents, and the inner rectangle represents each document,  $d$ . Document  $d$  is drawn from a topic distribution,  $\theta_d$ , that follows a Dirichlet distribution with a scalar parameter,  $\alpha$ :

$$\theta_d \sim \text{Dirichlet}(\alpha) \quad \text{Eq. (1).}$$

The parameter of the Dirichlet distribution should be a vector,  $\alpha = [\alpha_1, \dots, \alpha_k]$ , but because all elements of  $\alpha$  are the same,  $\alpha$  can be written as a scalar.

Next, let's move our eyes to the inner rectangle document,  $d$ . A topic vector,  $z_d$ , is drawn based on the topic distribution,  $\theta_d$ . The element in the vector,  $z_d$ , is denoted as  $z_{d,n}$ . They should sum up to 1:

$$z_{d,n} \sim \text{Multinomial}(\theta_d) \quad \text{Eq. (2).}$$

Next, the word,  $w_{d,n}$ , is drawn according to the topic-word matrix,  $\beta_{z_{d,n}}$ :

$$w_{d,n} \sim \beta_{z_{d,n}} \quad \text{Eq. (3).}$$

$\beta_{z_{d,n}}$  is the  $k \times V$  word probability matrix in which the number of rows is the number of topics, and the number of columns is the number of words.

In *Figure 10.6*, we use a shaded circle for  $w_{d,n}$  to denote it is observable and other hollow circles to denote other latent parameters. For notation efficiency, let's denote vector  $z$  as follows:

$$z = [z_{d,1}, \dots, z_{d,n}],$$

We will denote  $w$  as follows:

$$w = [w_{d,1}, \dots, w_{d,n}].$$

The central inferential problem for LDA determining the marginal posterior of the topic and word variables to represent the probability distribution of topics and words when given a particular document and the observed words in that document:

$$p(\theta, z \mid w, \alpha, \beta) = \frac{p(\theta, z, w \mid \alpha, \beta)}{p(w \mid \alpha, \beta)} \quad \text{Eq. (4)}$$

LDA considers data as arising from a generative process that includes hidden variables,  $\theta, z$ , and unknown parameters,  $\alpha, \beta$ . Let's review LDA in terms of prior probabilities, posterior probabilities, and likelihood with Bayes' theorem. The goal of LDA is to find the posterior probability using Bayes' theorem. The posterior,  $p(\theta, z | w, \alpha, \beta)$ , is what we want to solve. The denominator of Eq. (4),  $p(\theta, z, w | \alpha, \beta)$ , is the evidence that can easily be observed from the data. So, we will need to work on the numerator,  $p(\theta, z, w | \alpha, \beta)$ , to get a trackable state.

The numerator,  $p(\theta, z, w | \alpha, \beta)$ , can be expressed as follows:

$$p(\theta, z, w | \alpha, \beta) = p(\theta | \alpha) p(w | z, \beta) p(z | \theta) \quad \text{Eq. (5)}$$

Let's define Eq. (5) a bit at a time. First, from Eq. (1), we know  $p(\theta | \alpha)$  is the posterior distribution of the topic,  $\theta_d$  for the document given the Dirichlet prior parameter,  $\alpha$ :

$$p(\theta, \alpha) = \frac{\Gamma(\sum_{i=1}^k \alpha_i)}{\prod_{i=1}^k \Gamma(\alpha_i)} \prod_{i=1}^k \theta_i^{\alpha_i - 1} \quad \text{Eq. (6)}$$

$\Gamma(x)$  in Eq. (6) is the Gamma function.

Next, from Eq. (3), we know  $p(w | z, \beta)$  is the probability of observing a document of  $N$  words given topic  $k$  from the word matrix,  $\beta$ . It tells us how words are distributed within each topic. We can decompose  $p(w | z, \beta)$  into individual word probabilities and multiply them together:

$$p(w | z, \beta) = \prod_{n=1}^N \beta_{z_n, w_n}$$

Remember,  $\beta$  is a  $k$  by  $N$  matrix.  $\beta_{z_n, w_n}$  just means the elements of topic  $z_n$  and word  $w_n$ . Let's simplify the subscripts and rewrite them as follows:

$$p(w | z, \beta) = \prod_{i=1}^k \prod_{j=1}^N (\beta_{i,j})^{w_n^j z_n^j}$$

In Eq. (5),  $p(z | \theta)$  is the conditional probability of topic assignments,  $z_n$ , given the topic proportions,  $\theta_d$  for the document. It tells us which topics are likely for each word in the document based on the topic proportions. Thus, it can be simply derived from Eq. (2) as:

$$p(z_n | \theta) = \theta_{z_n} = \prod_{i=1}^k \theta_i$$

Now, Eq. (6) can be re-expressed as follows:

$$p(\theta, z, w | \alpha, \beta) = \left( \frac{\Gamma(\sum_{i=1}^k \alpha_i)}{\prod_{i=1}^k \Gamma(\alpha_i)} \prod_{i=1}^k \theta_i^{\alpha_i - 1} \right) \prod_{n=1}^N \prod_{i=1}^k \prod_{j=1}^N (\theta_i \beta_{i,j})^{w_n^j z_n^j} \quad \text{Eq. (7)}$$

When we integrate over  $\theta$  and sum over  $z$ , we get the marginal distribution of a document:

$$p(w | \alpha, \beta) = \int \left( \frac{\Gamma(\sum_{i=1}^k \alpha_i)}{\prod_{i=1}^k \Gamma(\alpha_i)} \prod_{i=1}^k \theta_i^{\alpha_i - 1} \right) \left( \prod_{n=1}^N \prod_{i=1}^k \prod_{j=1}^N (\theta_i \beta_{i,j})^{w_n^j z_n^j} \right) d\theta \quad \text{Eq. (8)}$$

Finally, if we take the product of the marginal probabilities of document  $d$ , we get the probability of the corpus,  $D$ :

$$p(D | \alpha, \beta) = \prod_{d=1}^D p(D | \alpha, \beta) d\theta_d$$

Back to Eq. (8), this complex equation is very hard to compute. It's the coupling of  $\theta$  and  $\beta$ , which makes it impossible to separate them. Although it is intractable for exact inference, in mathematics, there are approximate inferences to find the approximation. Typical algorithms for this are Laplace's approximation or variational inference. Blei et al. [1] applied the variational inference technique.

## Variational inference

**Variational inference** is a statistical technique that's used to approximate difficult-to-compute probability distributions with simpler, more tractable distributions. A typical, simple function to approximate a complex distribution is the Jensen's inequality.

In mathematics, you can use Jensen's inequality to get a lower bound for a convexity-based function. *Figure 10.6* explains this property:

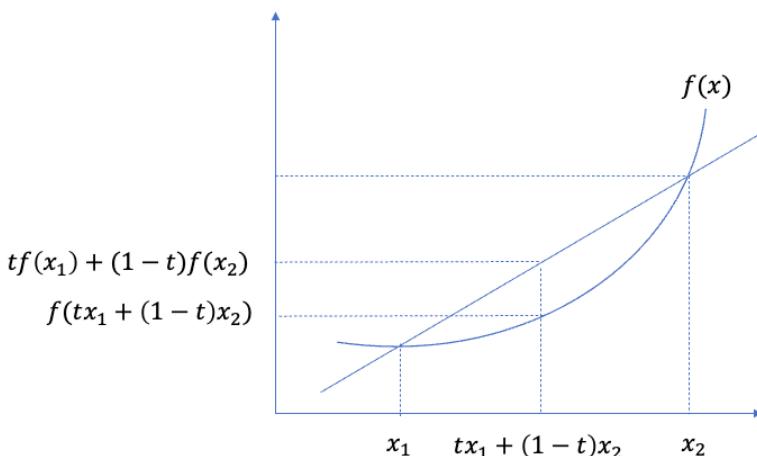


Figure 10.6 – Jensen's inequality

For any convex function,  $f(x)$ , the expected value (average) of  $f(x)$  for a random variable,  $x$ , is greater than or equal to the result of the function applied to the average of those numbers. See the two points,  $x_1$  and  $x_2$ . Any point between  $x_1$  and  $x_2$  is  $tx_1 + (1 - t)x_2$ , where the weight is  $t \in (0,1)$ . The projected value for  $tx_1 + (1 - t)x_2$  is  $f(tx_1 + (1 - t)x_2)$ . With the weight,  $t$ , the expected value of  $f(x_1)$  and  $f(x_2)$  is  $tf(x_1) + (1 - t)f(x_2)$ . Jensen's inequality says the following:

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

This indicates that you can find a lower bound,  $f(tx_1 + (1 - t)x_2)$ , usually easy to compute, for a complex function.

Now, let's see how Blei et al. provided a simplified graph model and used Jensen's inequality to get a lower bound in *Eq .(10)*. *Figure 10.7* shows the original graphical model on the left and the simplified variational distribution to approximate the posterior on the right:

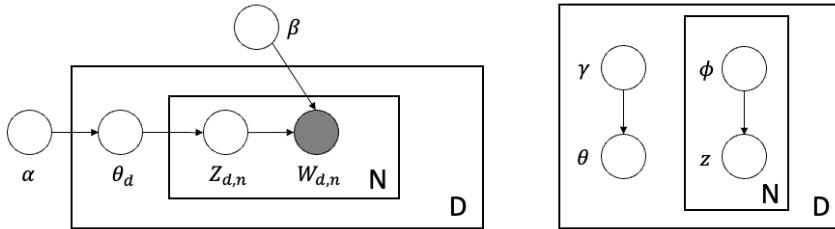


Figure 10.7 – Graphical representation with variational distribution to approximate the posterior

On the left, the coupling between  $\theta$  and  $\beta$  arises due to the edges between  $\theta$ ,  $z$ , and  $w$ . Blei et al. proposed the approximation graph on the right, which decouples  $\theta$  and  $\beta$ . By dropping these edges and the  $w$  nodes, the simplified graph has a posterior for each document in the following form:

$$q(\theta, z | \gamma, \phi) = q(\theta | \gamma) \prod_{n=1}^N q(z_n | \phi_n) \quad \text{Eq. (9)}$$

Here, the Dirichlet parameter,  $\gamma$ , and the multinomial parameters,  $(\phi_1, \dots, \phi_N)$ , are the free variational parameters.

You might be wondering where  $\alpha, \beta$  are, which we intended to solve in the beginning. Blei et al. applied Jensen's inequality to obtain a lower bound on the log-likelihood for  $q(\theta, z | \gamma, \phi)$ . The goal was to find the values for  $\gamma, \phi$  that yielded the tightest lower bound on the log-likelihood, as shown in *Eq. (10)*:

$$(\gamma^*, \phi^*) = \underset{(\gamma, \phi)}{\operatorname{argmin}} D(q(\theta, z | \gamma, \phi) \| p(\theta, z | w, \alpha, \beta)) \quad \text{Eq. (10)}$$

*Eq. (10)* is a minimization of the **Kullback-Leibler (KL)** divergence between the variational distribution and the actual posterior distribution. The KL divergence measures the difference between two probability distributions. Intuitively, you can think of it as quantifying how much one distribution *diverges* from another. It tells you how much extra information or surprise you would encounter when you try to approximate one distribution with another. If KL divergence is zero, it means the two distributions are identical; if it's greater than zero, it indicates there's some mismatch between them, with the value quantifying the extent of that mismatch. We want the KL divergence to be as minimal as possible. So, *Eq. (10)* is the minimization of the KL divergence. It will yield the following two relationships:

$$\gamma_i = \alpha_i + \sum_{n=1}^N \phi_{ni} \quad \text{Eq. (11)}$$

$$\phi_{ni} \propto \beta_{iw_n} \exp\{\log(\theta_{il}|\gamma)\} \quad \text{Eq. (12)}$$

The  $\propto$  symbol means *proportional to*. It simply means we normalize all  $\phi_{ni}$  so that they sum up to one. Finally, we need to solve  $\exp\{\log(\theta_{il}|\gamma)\}$  in *Eq. (12)*. It is the first derivative of the *logGamma* function,  $\Psi$ :

$$E_q[\log(\theta_{il}|\gamma)] = \Psi(\gamma_i) - \Psi(\sum_{j=1}^k \gamma_j) \quad \text{Eq. (13)}$$

*Eq. (13)* is solvable because it can be computed via Taylor approximations. Up to this point, the problem is considered solved. However, in *Eq. (11)* and *Eq. (12)*, we do not know  $(\alpha, \beta)$ , so how can we estimate  $(\gamma, \phi)$ ? We can think of the E-M algorithm. But it is different from the standard E-M algorithm because now, we have the variational inference. So, the earlier variational inference, together with E-M, is called **variational Expectation-Maximization** or **variational EM**. It combines the E-M framework with variational inference, which aims to approximate complex posterior distributions with simpler ones.

Let me explain the use of variational E-M in the context of LDA.

## Variational E-M

Variational E-M is an extension of E-M that incorporates variational inference. In variational E-M, during the “expectation” step, instead of computing the exact posterior distribution of the latent variables as in standard E-M, it approximates this posterior using a simpler distribution from a predefined family. Then, during the “maximization” step, it optimizes the model parameters to maximize a lower bound on the likelihood of the observed data, which is derived from the approximate posterior. Variational E-M iterates between these two steps until convergence, providing a computationally efficient way to perform parameter estimation in complex probabilistic models, especially in Bayesian settings.

Now, let’s describe the variational E-M algorithm in our context:

1. **The E-step:** We get the optimal values of the variational parameters,  $(\gamma, \phi)$ , in *Eq. (11)* and *Eq. (12)* for every document in the corpus by assuming we know the hidden parameters,  $(\alpha, \beta)$ . It assumes random values as the initial values for  $(\alpha, \beta)$  to start with. Notice that the variational inference shown in *Eq. (11)* and *Eq. (12)* is already incorporated in the E-step.
2. **The M-step:** We maximize the lower bound on the log-likelihood concerning  $(\alpha, \beta)$  separately. The lower bound on the log-likelihood is as follows:

$$\ell(\alpha, \beta) = \sum_{d=1}^D \log p(w_d | \alpha, \beta) \quad \text{Eq. (14)}$$

The preceding E-M steps are repeated until *Eq. (14)* converges. Because *Eq. (14)* is the bound on the log-likelihood of the observed data, it is called the **evidence lower bound (ELBO)**.

When solving the optimization problem of LDA, the typical alternative algorithms are variational E-M and **Gibbs sampling**. I will briefly describe Gibbs sampling, and then compare the advantages of the two approaches.

## Gibbs sampling in LDA

Gibbs sampling is used in LDA to iteratively sample the latent variables (topic assignments) for each word while keeping all other variables (including model parameters and topic proportions) fixed. The sampling process is based on the conditional distributions of the latent variables given the current state of the others. During Gibbs sampling iterations, each word’s topic assignment is updated based on the current state of the model and the other assignments. After many iterations, the sampling

process converges to a stationary distribution, which approximates the true posterior distribution over latent variables. Gibbs sampling is a **Markov chain Monte Carlo (MCMC)** method. MCMC is a computational technique that's used to generate approximate samples from complex probability distributions. It employs a Markov chain that iteratively explores the distribution, and at each step, it proposes a new sample based on the current state. Let's take this one step further and compare the two approaches.

## Variational E-M versus Gibbs sampling

These two approaches offer different trade-offs in terms of computational efficiency and accuracy. I think it is worthwhile describing these here. Variational E-M is preferred for efficiency and scalability when dealing with large datasets and when a close approximation to the posterior distribution is acceptable. Gibbs sampling is preferred when higher accuracy is needed, and computational resources allow for the additional computational cost. Variational E-M is computationally efficient and often faster than Gibbs sampling. It provides a closed-form update for the variational parameters and can be scaled to large datasets and high-dimensional topic models. However, the quality of the approximation depends on the chosen variational family and can be less accurate than Gibbs sampling in capturing the true posterior distribution. Gibbs sampling provides a more accurate estimation of the posterior distribution compared to variational inference. It can capture complex dependencies between latent variables and model parameters. However, it can be computationally expensive, especially for large datasets or high-dimensional topic models.

## Summary

In this chapter, we learned about the structure of LDA and its mathematical process to achieve results. LDA assumes a document is generated from a distribution of topics and that each topic has a distinctive distribution of words. Both the distribution of topics and the distribution of words are latent. By using the generative modeling method, which generates data from the assumed distributions to fit the observable data, LDA can discover the hidden distributions. We learned about the variational E-M algorithm for generative modeling to solve the parameters of hidden distributions. We compared the advantages of two algorithms – variational E-M and Gibbs sampling – that are typically used in solving the optimization problem of LDA.

In the next chapter, we will develop Python code in Gensim to conduct LDA topic modeling for real-world data. We will make technical decisions, such as how to determine the optimum number of topics and how to use the model to score new documents.

## Questions

1. What is the main idea of LDA?
2. What does the name LDA mean?
3. What are assumed to be observable and not observable in the LDA setting?
4. Describe the difference between descriptive modeling and generative modeling.
5. Describe E-M.
6. Describe variational E-M.
7. Describe the Gibbs sampling method.

## References

1. David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. *Latent Dirichlet allocation*. J. Mach. Learn. Res. 3, null (3/1/2003), 993 – 1,022.
2. Kuo, Chris, “Chapter 11: XGBOD”, *Handbook of Anomaly Detection: With Python Outlier Detection*: <https://a.co/d/9oL7NHp>
3. Kuo, Chris, “Chapter 7: GMM”, *Handbook of Anomaly Detection: With Python Outlier Detection*: <https://a.co/d/9oL7NHp>



# 11

## LDA Modeling

In *Chapter 9, Understanding Discrete Distribution*, and *Chapter 10, Latent Dirichlet Allocation*, we learned about the Dirichlet distribution and the structure of the LDA model, which equipped you with a sound theoretical background. In this chapter, we will go over the code to build an LDA model. I will touch upon the key decisions in building an LDA model, including text preprocessing, model hyperparameters, the determination of the number of topics, and how to use the model in production to score new documents. This is a special feature in this book that focuses on model implementation in production. In short, we will cover the following topics:

- Text preprocessing
- Experimenting with LDA modeling
- Building LDA models with a different number of topics
- Determining the optimal number of topics
- Using the model to score new documents

With the completion of this chapter, you will be able to develop LDA topic models independently. You will also be able to instruct how to implement the model to score new documents.

### Technical requirements

You will need to install the `gensim` module and load the `LdaModel` class using the following commands:

```
pip install gensim
from gensim.models import LdaModel
```

You can find the code files for this chapter on GitHub at <https://github.com/PacktPublishing/The-Handbook-of-NLP-with-Gensim/tree/main/Chapter11>.

## Text preprocessing

We discussed earlier that good text preprocessing yields a good model outcome. Text preprocessing includes stop word removal and lemmatization. Some domain-specific common words may be considered too common and can be removed as well. You are advised to perform text preprocessing for LDA modeling too. LSA/LSI, LDA, and Ensemble LDA all require text preprocessing for a better modeling outcome. In contrast, Word2Vec, Doc2Vec, text summarization, and **Bidirectional Encoder Representations From Transformers (BERT)** topic modeling do not necessarily need text preprocessing.

This chapter uses the same AG's corpus data of news articles (as mentioned in the *Preface*) so that you can focus more on learning techniques rather than different datasets. The text preprocessing task here is very similar to that in *Chapter 6, Latent Semantic Indexing with Gensim*. Hence, I will just go through the same text preprocessing code without offering much detail.

### Preprocessing

We will tokenize the words of each document and remove stop words. Here's how we do this:

```
import pandas as pd
import numpy as np
pd.set_option('display.max_colwidth', -1)
path = "/content/gdrive/My Drive/data/gensim"
train = pd.read_csv(path + "/ag_news_train.csv")
from gensim.parsing.preprocessing import preprocess_string
text_tokenized = []
for doc in train['Description']:
    k = preprocess_string(doc)
    text_tokenized.append(k)
text_tokenized[0:3]
```

The result is a list of tokenized documents. The first three tokenized documents are shown as follows:

```
[['reuter', 'short', 'seller', 'street', 'dwindl', 'band', 'ultra',
'cynic', 'see', 'green'],
['reuter', 'privat', 'invest', 'firm', 'carlyl', 'group', 'reput',
'make', 'time', 'occasion', 'controversi', 'plai', 'defens',
'industri', 'quietli', 'place', 'bet', 'market'],
['reuter', 'soar', 'crude', 'price', 'plu', 'worri', 'economi',
'outlook', 'earn', 'expect', 'hang', 'stock', 'market', 'week',
'depth', 'summer', 'doldrum']]
```

Similar to LSI, LDA modeling needs documents to be organized in **bag-of-words (BoW)** or **Term Frequency-Inverse Document Frequency (TF-IDF)**, as shown in the following examples:

- Here's an example of BoW:

```
from gensim.corpora import Dictionary
gensim_dictionary = Dictionary()
bow_corpus = [gensim_dictionary.doc2bow(doc,
    allow_update=True) for doc in text_tokenized]
id_words = [(gensim_dictionary[id], count) for id,
    count in line] for line in bow_corpus]
```

- And here's an example of TF-IDF:

```
from gensim.models import TfidfModel
tfidf = TfidfModel(bow_corpus)
corpus_tfidf = tfidf[bow_corpus]
```

Next, we will save the dictionary and the BoW and TF-IDF objects to be used in production.

First, save the dictionary list, as follows:

```
from gensim.test.utils import datapath
dict_file = datapath(path + "/gensim_dictionary_AGnews")
gensim_dictionary.save(dict_file)
```

Then, save the BoW and TF-IDF objects.

You can use `pickle` to save the BoW object, as follows:

```
import pickle
file = open(path + "/BoW_AGnews_corpus.pkl", 'wb')
pickle.dump(bow_corpus, file)
file.close()
```

Likewise, let's save the TF-IDF object:

```
file = open(path + "/Tfidf_AGnews_corpus.pkl", 'wb')
pickle.dump(corpus_tfidf, file)
file.close()
```

Alternatively, the `gensim` module provides a function to let you save in the matrix market format, as seen here:

```
from gensim import corpora
corpora.MmCorpus.serialize(path + "/BoW_AGnews_corpus.mm",
    bow_corpus)
```

```
corpora.MmCorpus.serialize(path +
    "/TFIDF_AGnews_corpus.mm", corpus_tfidf)
```

The preceding tasks are the data preparation steps in NLP. Now, we are ready to model with LDA. To help you get familiar with basic LDA coding and understand the output, let's experiment using an LDA model with only 10 topics.

## Experimenting with LDA modeling

The BoW data and TF-IDF data are two variants of data formation. The model built on BoW data will result in a different outcome from the model built on TF-IDF. We will build models on both variants.

### A model built on BoW data

The basic syntax of LDA is easy. The required input parameter is `corpus`. We assign the BoW data to build the first model, as illustrated in the following code snippet:

```
from gensim.models import LdaModel
lda_bow = LdaModel(bow_corpus,
                    num_topics=10,
                    id2word = gensim_dictionary)
```

I'd like to review several important model inputs, as follows:

- `num_topics`: This is the number of topics. In this experiment, we will just assign 10. We will learn how to determine the optimal number of topics in the *Determining the optimal number of topics* section.
- `random_state=None`: This is helpful for reproducibility.
- `id2word`: We assign our `gensim_dictionary` dictionary from our corpus. If we do not assign a dictionary, Gensim will use its default dictionary and notify you with a warning message.

Other model inputs are optional and already have their default values, as outlined here:

- `distributed=False`: This refers to whether you are using distributed computing or not. The default is not.
- `chunksize=2000`: The number of documents in a chunk.
- `passes=1`: This parameter defines how many times the model will go through the entire corpus during training. It is the same as `epochs` in a **deep learning (DL)** context.
- `update_every=1`: This parameter defines how often the model updates after the number of documents. This is essentially for memory optimization. It is set to 0 for batch learning and 1 for online iterative learning.

- `alpha='symmetric'`: This is an a priori belief on document-topic distribution.
- `eta=None`: This is a priori belief on topic-word distribution, if any. The value can be a 1D array of length `num_words`. It can also be a matrix of shape (`num_topics`, `num_words`) for any known word-topic combination. The default is `None`.
- `iterations=50`: This is the maximum number of iterations through the corpus. It controls how often the model repeats a particular loop over each document through the corpus.
- `gamma_threshold=0.001`: The Gamma parameters are the topic weight variational parameters for each document. Here, `gamma_threshold` refers to the minimum change in the value of the gamma parameters to continue iterating.
- `per_word_topics=False`: If this parameter is `True`, the model can compute a list of topics sorted in descending order for each word. It actually returns the word count for each topic in descending order.
- `minimum_phi_value=0.01`: This is used when `per_word_topics` is `True`. It filters out word terms lower than this minimum threshold.
- `minimum_probability=0.01`: This filters out topics with a probability lower than this threshold.

You can apply the basic syntax without much elaboration and then fine-tune the hyperparameters as needed.

### ***Showing the model result on BoW***

A *topic* is a distribution over words. The words inform us what the topic is about. The function to show a topic is `print_topics()`, as seen in the following code snippet:

```
import pprint as pp
pp.pprint(lda_bow.print_topics())
```

Let's see five topics and their words respectively:

```
[ (0,
  0.039*"quot" + 0.017*"new" + 0.014*"servic" + 0.014*"internet" +
  0.013*"microsoft" + 0.013*"search" + 0.012*"compani" + 0.011*"onlin" +
  0.010*"research" + 0.009*"phone"),
 (1,
  0.023*"said" + 0.014*"presid" + 0.012*"reuter" + 0.010*"kill" +
  0.010*"offici" + 0.009*"minist" + 0.009*"elect" + 0.009*"leader" +
  0.009*"iraq" + 0.008*"govern"),
 (2,
  0.019*"new" + 0.015*"red" + 0.013*"york" + 0.012*"angel" +
  0.011*"citi" + 0.010*"sox" + 0.010*"lo" + 0.010*"san" +
  0.010*"christma" + 0.009*"game"),
```

```
(3,
0.022*"game" + 0.017*"point" + 0.015*"win" + 0.014*"second" +
0.013*"night" + 0.013*"final" + 0.012*"score" + 0.011*"victori" +
0.011*"team" + 0.011*"leagu"),
(4,
0.021*"oil" + 0.014*"china" + 0.010*"state" + 0.010*"said" +
0.009*"year" + 0.009*"price" + 0.008*"world" + 0.008*"new" +
0.008*"govern" + 0.008*"unit"),
(5,
0.015*"dec" + 0.014*"cup" + 0.011*"year" + 0.009*"player" +
0.008*"nextel" + 0.007*"winter" + 0.007*"houston" + 0.007*"england" +
0.007*"real" + 0.007*"defeat") ,
```

A word with a large weight means it has more significance than other words. Did you notice the coefficients of the words in each topic have been sorted in descending order? Let's understand what's in topic 0. The words "new", "servic", "internet", "microsoft", "search", "compani", and "onlin" are likely about new technology companies such as Microsoft offering internet services.

Topic 1 has keywords such as "kill", "presid", "offici", "leader", "elect", and "govern". Apparently, this is about a political topic. Topic 2 has keywords such as "red", "sox", "game", "new", "york", and "christma". It is about a sport topic. Recall our AG news data has four classes: "World", "Sports", "Business", and "Sci/Tech". These topic results coincide with our knowledge.

However, there are some common words such as "reuter", "said", "fullquot", or "quot" appearing in many topics. They are frequent words in news documents. The BoW method considers the significance of a word by its frequency. We may want to *down-weight* frequent words so that special words can emerge. We learned in *Chapter 2, Text Representation*, that the alternative method TF-IDF is designed to reflect the importance of a word in a document of a corpus. Let's see if TF-IDF can improve the results.

## A model built on TF-IDF data

Similar to what we have done on the BoW data, let's develop a different model for the TF-IDF data, as follows:

```
lda_tfidf = LdaModel(corpus_tfidf, num_topics=10,
id2word = gensim_dictionary)
```

### **Showing the model result on TF-IDF**

Let's show the topics and words with TF-IDF:

```
import pprint as pp
pp pprint( lda_tfidf.print_topics() )
```

We get the following output:

```
[ (0,
  0.011*"cup" + 0.010*"nextel" + 0.008*"score" + 0.008*"champion" +
  0.006*"world" + 0.006*"round" + 0.005*"turkei" + 0.005*"david" +
  0.005*"final" + 0.005*"spain"),
(1,
  0.008*"game" + 0.008*"season" + 0.008*"point" + 0.007*"coach" +
  0.007*"night" + 0.006*"team" + 0.006*"sport" + 0.005*"new" +
  0.004*"player" + 0.004*"year"),
(2,
  0.006*"space" + 0.006*"presid" + 0.006*"bush" + 0.005*"nasa" +
  0.005*"elect" + 0.004*"nov" + 0.004*"quot" + 0.004*"johnson" +
  0.004*"state" + 0.004*"scientist"),
(3,
  0.009*"winter" + 0.009*"drug" + 0.007*"nhl" + 0.007*"acquir" +
  0.007*"yahoo" + 0.006*"shop" + 0.006*"heat" + 0.005*"retir" +
  0.005*"search" + 0.005*"jan"),
(4,
  0.009*"arsen" + 0.009*"gaza" + 0.007*"ukrain" + 0.007*"isra" +
  0.007*"assist" + 0.007*"minist" + 0.007*"prime" + 0.006*"sharon" +
  0.006*"european" + 0.006*"insurg"),
(5,
  0.007*"said" + 0.007*"kill" + 0.007*"palestinian" + 0.007*"dec" +
  0.006*"decemb" + 0.006*"minist" + 0.006*"polic" + 0.006*"iraq" +
  0.005*"peopl" + 0.005*"offici")]
```

Indeed, we see fewer of the common words such as "reuter", "said", "fullquot", or "quot" in the TF-IDF topic results. Topic 0 has "world", "cup", "champion", "final", and "round" keywords, so I know it is a sports topic. However, we still see the common words "said" or "quot" in the TF-IDF model results. Maybe their presence in a document conveys some special message. But overall, if they are considered too common to add values, they can be removed in the text preprocessing step. You may recall that we said that a large amount of time in NLP tasks is about preprocessing. The quality of an NLP model depends on the refinement of words in the preprocessing step. That said, in many applications, a gentle treatment for removing words is enough to yield reasonable results.

### Important note

TF-IDF can down-weight common words so that they appear less frequently than the BoW data. If there are domain-specific common words, we can remove them in the beginning to pursue a better model outcome.

So far, we have demonstrated the modeling process with 10 topics. In the next section, we will build more LDA models with a range of topics and learn how to determine a better model with the optimal number of topics.

## Building LDA models with a different number of topics

The number of topics is an exogenous parameter. We will build several models with a range of topics. We will use the BoW data and the TF-IDF data. To economize our code, we will develop a small function, `build_lda_models()`. This function takes three arguments, as follows:

- `input_data`: Either BoW or TF-IDF
- `name`: The filename to save the model
- `k`: The number of topics

The code is shown as follows:

```
def build_lda_models(input_data, name, k):  
    np.random.seed(42)  
    # Train the model  
    lda = LdaModel(input_data,  
                    num_topics=k,  
                    id2word = gensim_dictionary)  
    # Save the model  
    from gensim.test.utils import datapath  
    tempfile = datapath(path + "/LDA_" + name  
    + "_" + str(k))  
    lda.save(tempfile)
```

We will use the `build_lda_models()` function to build a range of models.

### Models built on BoW data

We will build models on BoW data with a range of topics from 1 to 251 with an increment of 50. I chose 50 simply not to build too many models. You will test with different values. The code is illustrated here:

```
import numpy as np  
numTopicsList = np.arange(1,251,50)  
for k in numTopicsList:  
    build_lda_models(bow_corpus, "bow", k)
```

You will see a range of model objects saved in your specified folder.

## Models built on TF-IDF data

We will build models on TF-IDF data with a range of topics too. Again, the incremental value for the number of topics is not critical. In the next section, we will compute the coherence score to find out the optimal number of topics. The code is shown here:

```
np.random.seed(42)
numTopicsList = np.arange(1, 251, 50)
for k in numTopicsList:
    build_lda_models(corpus_tfidf, "tfidf", k)
```

Similar to the models built on BoW data and saved in the folder, you will see a range of model objects saved in your folder.

Let's return to a bigger question: *How to determine the optimal number of topics?* The answer is the coherence score.

## Determining the optimal number of topics

What defines a *topic*? A topic should be distinctive enough that it can represent a concept and the words associated with the concept. On the other hand, if a topic is a mixed BoW such that the topic is not concrete enough, it is better to separate the topic into two or more topics. As a result, the *closeness* of words in a topic is an important measure. Words in the same topic are better being close to each other.

In NLP, the metric to measure the *closeness* of a topic is called the **coherence score**. In *Chapter 5, Cosine Similarity*, we learned the cosine similarity that measures the similarities between any two words. The coherence score is the average or median of the word similarities of the top words in a topic. This definition was given by Röder, Both, and Hinneburg (2015) [2]. There are three metrics to compute the coherence score, as outlined here:

- **Content Vectors (CV):** The default metric of gensim
- **UMass:** A more popular metric
- **UCI:** A modification of UMass

Let's build models with a range of topics and calculate the coherence scores, as follows:

```
from gensim.models.coherencemodel import CoherenceModel
coherence_lda_bow = CoherenceModel(model=lda_bow,
                                    texts=text_tokenized,
                                    dictionary=gensim_dictionary,
                                    coherence='c_v')
coherence_lda = coherence_lda_bow.get_coherence()
print('\nCoherence Score: ', coherence_lda)
```

What we are going to do is to build models of k topics and calculate the coherence scores. The model that yields the smallest coherence score will be the optimal model. Because we will iterate through a range of k values, we will write the code in a function. The function simply contains two parts—the modeling step and the coherence score step:

```
from gensim.models import LdaModel
from gensim.models.coherencemodel import CoherenceModel
from gensim.test.utils import datapath
np.random.seed(42)
def coherenceScore(corpus, name, k):
    # Load the model
    tempfile = datapath(path + "/LDA_"
        + name + "_" + str(k))
    lda = LdaModel.load(tempfile)

    # Compute the score
    coherence = CoherenceModel(model=lda,
                                texts=text_tokenized,
                                dictionary=gensim_dictionary,
                                coherence='u_mass')
    return coherence.get_coherence()
```

Let's create a `coherenceList` list to store all the coherence scores:

```
coherenceList = []
numTopicsList = np.arange(1, 251, 50)
for k in numTopicsList:
    c = coherenceScore(bow_corpus, "bow", k)
    coherenceList.append(c)
```

*Figure 11.1* plots the coherence scores for a different number of topics. It is evident that the minimum coherence score happens when the number of topics is 151:

```
from matplotlib import pyplot as plt
plt.plot(numTopicsList, coherenceList)
plt.savefig(path + "/LDA_bow_coherence")
plt.show()
```

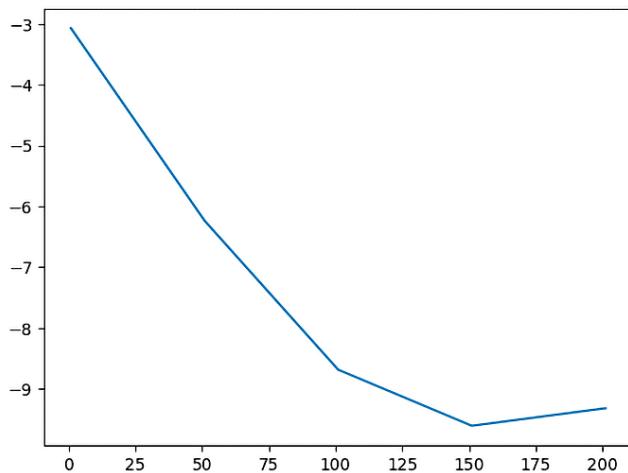


Figure 11.1 – Coherence scores

We feel comfortable that the model with 151 topics should be a better model among all models. Let's load the model and call it `lda`:

```
tempfile = datapath(path + "/LDA_bow_151")
lda = LdaModel.load(tempfile)
```

Let's print out the keywords of the first five topics:

```
import pprint as pp
pp pprint(lda_bow.print_topics() [0:5])
```

The first topic contains keywords such as "people", "kill", "force", "india", "injury", and "soldier". It seems to be about news in India that people were killed. Notice these words together form a concrete concept. The second topic seems to refer to telephone advertisements and business measures:

```
[ (5,
  0.172*"peopl" + 0.143*"kill" + 0.104*"forc" + 0.088*"india" +
  0.061*"said" + 0.047*"southern" + 0.045*"injur" + 0.044*"soldier" +
  0.035*"offici" + 0.033*"reuter") ,
  (67,
  0.271*"like" + 0.214*"look" + 0.049*"measur" + 0.046*"speci" +
  0.046*"telephon" + 0.044*"advertis" + 0.043*"discov" + 0.036*"impos" +
  0.035*"paid" + 0.030*"count") ,
  (17,
  0.134*"busi" + 0.133*"sai" + 0.107*"maker" + 0.095*"product" +
  0.083*"softwar" + 0.070*"compani" + 0.032*"corp" + 0.024*"small" +
  0.023*"book" + 0.023*"announc") ,
```

```
(61,
0.110*"israel" + 0.104*"nuclear" + 0.099*"iran" + 0.066*"said" +
0.056*"debt" + 0.043*"treasuri" + 0.038*"reuter" + 0.038*"lenovo" +
0.034*"wednesdai" + 0.030*"bar"""),
(58,
0.171*"need" + 0.081*"tour" + 0.079*"thing" + 0.058*"stori" +
0.054*"photo" + 0.049*"beach" + 0.048*"present" + 0.048*"tell" +
0.047*"anim" + 0.044*"easi"""),

```

The rich outcome of an LDA model can be visualized through pyLDAvis. We will learn how to use pyLDAvis in the next chapter.

## Using the model to score new documents

In this book, I have kept the perspective for model implementation. Our work does not stop at the modeling step but also considers the implementation. In the previous section, we completed a model. I will now show you how to use it to score new documents. I keyed in three new documents in the `new_texts` list. The first one is a technology news article stating that the new Windows operating system will be released, the second one is also a technology news article, and the third one is an economic news article:

```
new_texts = [
'The new Windows operating system will be released',
'The system uses the most difficult technologies and techniques',
'The U.S. consumer prices dropped for the first time in ten months']
```

### Important note

Any text preparation procedures that are applied to the training data will be applied to the new documents as well. We will perform text preprocessing on the new documents.

The new texts will go through the same text preprocessing step.

## Text preprocessing

To do text processing for new texts, you just need to replicate the text preprocessing code for model training, as follows:

```
from gensim.parsing.preprocessing import preprocess_string
new_texts_tokenized = []
for doc in new_texts:
    k = preprocess_string(doc)
    new_texts_tokenized.append(k)
```

Let's apply BoW to the `new_texts_tokenized` tokenized documents and store them in `new_corpus`:

```
new_corpus = [model_dict.doc2bow(text) for text in  
             new_texts_tokenized]  
new_corpus
```

Now, the BoW form for the three documents looks like this:

```
[(67, 1), (307, 1), (1731, 1), (1903, 1)],  
 [(422, 1), (495, 1), (1701, 1), (1956, 1)],  
 [(27, 1), (37, 1), (66, 1), (127, 1), (632, 1)]]
```

## Scoring new texts

Now, we can score the BoW documents by the model. We want to know the topic distribution of each document. We will use `.get_document_topics()`. It will return the topic distribution for each document BoW as a `(topic_id, topic_probability)` list of 2-tuples. It ignores topics where the probability is below the set minimum probability:

```
new_texts_vector = lda_bow.get_document_topics(  
    new_corpus,  
    minimum_probability=0.05,  
    per_word_topics=False)
```

We set the minimum probability to 5%. Any topic with a probability of less than 5% will be ignored. The second parameter, `per_word_topics`, if set as True, will return a `word_ids` list.

## Outcome

Let's see the output:

```
for doc in new_texts_vector:  
    print(doc)
```

Its outcome is a list of 2-tuples for `(topic_id, topic_probability)`:

```
News article 1: [(48, 0.4013), (65, 0.2014), (100, 0.2012)]  
News article 2: [(43, 0.2013), (52, 0.2013), (80, 0.2013),  
 (101, 0.2013)]  
News article 3: [(22, 0.4381), (85, 0.3973)]
```

The first news article, 'The new Windows operating system will be released', is 40% topic 48, 20% topic 65, and 20% topic 100. Let's print out the keywords of topics 48, 65, and 100:

```
for i in [48,65,100]:
    print("Topic", i,"is:", lda_bow.print_topic(i))
```

40% of this document relates to topic 48, which is about the Microsoft software system:

```
Topic 48 is: 0.116*"european" + 0.105*"microsoft" + 0.101*"manag"
+ 0.099*"oper" + 0.071*"base" + 0.066*"firm" + 0.056*"softwar" +
0.048*"version" + 0.048*"system" + 0.041*"server"
```

20% of this document relates to topic 65:

```
Topic 65 is: 0.241*"releas" + 0.091*"ask" + 0.059*"employe" +
0.043*"captur" + 0.043*"storag" + 0.041*"diplomat" + 0.040*"pension" +
0.037*"hewlett" + 0.037*"packard" + 0.034*"captain"
```

20% of this document relates to topic 100:

```
Topic 100 is: 0.555*"new" + 0.138*"york" + 0.036*"electron" +
0.031*"consum" + 0.026*"manufactur" + 0.025*"reuter" + 0.020*"januari" +
0.019*"unveil" + 0.015*"time" + 0.015*"minnesota"
```

The second news article is 'The system uses the most difficult technologies and techniques'. Let's examine the topic distribution, as follows:

```
for i in [43,52,80,101]:
    print("Topic", i,"is:", lda_bow.print_topic(i))
```

It has about 20% for the four topics 43, 52, 80, 101, as seen here:

```
Topic 43 is: 0.185*"technolog" + 0.093*"decemb" + 0.070*"demand" +
0.065*"chip" + 0.064*"design" + 0.045*"intel" + 0.040*"let" +
0.038*"saddam" + 0.034*"hussein" + 0.033*"respons"
Topic 52 is: 0.131*"possibl" + 0.118*"join" + 0.114*"popular" +
0.105*"area" + 0.091*"competit" + 0.048*"avoid" + 0.045*"rout" +
0.033*"competitor" + 0.029*"surround" + 0.025*"burn"
Topic 80 is: 0.104*"japan" + 0.092*"recent" + 0.081*"tokyo" +
0.080*"gain" + 0.061*"grow" + 0.057*"japanes" + 0.045*"export" +
0.041*"lift" + 0.036*"wednesdai" + 0.035*"reuter"
Topic 101 is: 0.117*"phone" + 0.111*"mobil" + 0.093*"reach" +
0.093*"provid" + 0.092*"half" + 0.081*"us" + 0.076*"custom" +
0.040*"cell" + 0.035*"figur" + 0.032*"suggest"
```

The third news article is 'The U.S. consumer prices dropped for the first time in ten months'. The model assigns 43% to topic 22 and 39% to topic 85:

```
for i in [22,85]:
    print("Topic", i,"is:", lda_bow.print_topic(i))
```

43% of the document relates to topic 22, which is about economic news caused by the oil price shock:

```
Topic 22 is: 0.193*"oil" + 0.183*"price" + 0.058*"stock" +
0.051*"fall" + 0.047*"drop" + 0.045*"reuter" + 0.038*"market" +
0.037*"crude" + 0.031*"short" + 0.029*"week"
Topic 85 is: 0.150*"percent" + 0.074*"month" + 0.069*"depart" +
0.051*"fell" + 0.043*"said" + 0.040*"report" + 0.035*"survei" +
0.031*"fridai" + 0.030*"asian" + 0.030*"deni"
```

The prediction outcome of LDA for a document is a list of 2-tuples in the form of (`topic_id`, `topic_probability`). We will evaluate the quality of model performance by the words that belong to a topic. Although LDA identifies words associated with a topic, human review and feedback are still essential in finalizing an LDA model.

## Summary

In this chapter, we learned to build LDA models with BoW data and TF-IDF data. We compared the LDA model outcomes between BoW and TF-IDF data. We then built more LDA models with a range of topics. We learned what the coherence score is and how to use it to determine the optimal number of topics. We also learned the prediction outcome for a document is a list of 2-tuples for topic IDs and topic probabilities. The LDA model presents a document as a distribution of topics and each topic as a distribution of words. Such rich content in the results incurs another challenge: what is the best way to visualize the rich content?

In the next chapter, we will learn how to design a visual tool to deliver rich content and how to communicate the content.

## Questions

1. Name a few NLP modeling techniques that will yield a better outcome when using text preprocessing.
2. Compare the outcome of an LDA model with BoW data to the outcome of an LDA model with TF-IDF data.
3. What is the coherence score used in LDA?
4. Describe the prediction outcome of LDA for a document.

## References

1. *David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. J. Mach. Learn. Res. 3, null (3/1/2003), 993–1022.*
2. *Röder, M., Both, A., & Hinneburg, A. (2015). Exploring the Space of Topic Coherence Measures. Proceedings of the Eighth ACM International Conference on Web Search and Data Mining.*



# 12

## LDA Visualization

Topic modeling classifies a large volume of corpora into topics, and each topic has a set of distinct words. It produces very rich information for each topic. The next challenge is how to present the rich information. This challenge is a research topic in the development of NLP.

The good news is that several visual tools have provided the solution, including `pyLDAvis`, which we will cover in this chapter. We will first discuss the gap between hard topic modeling and soft human comprehension. Then we will learn how to use `pyLDAvis`.

We will cover the following topics:

- Designing an infographic
- Data visualization with `pyLDAvis`

By the end of this chapter, you will be able to visualize an LDA model using interactive infographics to explain model insights. The design of `pyLDAvis`'s infographics has even influenced the visualization of BERTopic model outcomes, which we will practice in *Chapter 14, LDA and BERTopic*.

### Technical requirements

The code files are available on GitHub at <https://github.com/PacktPublishing/The-Handbook-of-NLP-with-Gensim/tree/main/Chapter12>.

### Designing an infographic

Topic modeling algorithms identify distinct topics through their mathematical operations. Now, the question is how to make the results easily interpretable.

The numerical results of topics are difficult for humans to understand. Humans are not good at processing large numbers of vectors and matrices or deriving insights from an ocean of numbers. Users may lose interest after inspecting a lot of dry numbers. For example, a user who is not a data expert may ask what it means when a word is multiplied by a number, like the following:

```
'0.017*"said" + 0.016*"deal" + 0.013*"agre" + 0.012*"million"  
+ ' '0.012*"compani" + 0.011*"union" + 0.011*"european" +  
0.010*"agreement" + ' '0.010*"billion" + 0.008*"contract"
```

We need a better visual tool!

But in our defense, the topics discovered by models mechanically may be truly nonsense. The result can be poor because it groups two or more concepts into one topic. A topic can be too vague because it has many generic words (for example, “people, like, same”). A topic model could be improved if it removes certain words or builds domain-specific models.

Through the use of infographics or even animations, we can comprehend complex data relationships to gain data-driven insights for easy comprehension. Let’s consider the elements to visualize.

To design an infographic for the results of topic models, we want to consider the relative sizes of topics and the content of a topic. The ideal infographics shall reveal several aspects:

- The distances between topics
- The top words of a topic
- The distribution of words in a topic
- The distribution of words in a topic when compared with that of the whole corpora

Sievert and Shirley designed an interactive infographic tool that meets the previous criteria [1]. This well-received tool is available in Python as `pyLDAvis` and in R as `LDAvis` [2].

## Data visualization with `pyLDAvis`

In *Chapter 11, LDA Modeling*, we saved the LDA model and its dictionary for future use. Now we will load the LDA model and the dictionary. The `MmCorpus()` function loads the BoW object saved previously in the **Matrix Market (MM)** exchange format. The MM exchange format is a format for matrices. You also can use `pickle` dump and load to handle the BOW data, as I explained in the previous chapter:

```
from gensim.models import LdaModel  
from gensim.test.utils import datapath  
from gensim.corpora import Dictionary
```

Load the LDA model on the BoW data:

```
path = "/content/gdrive/My Drive/data/gensim"  
bow_file = datapath(path + "/LDA_bow_151")  
lda_bow = LdaModel.load(bow_file)
```

Load the LDA model on the TF-IDF data:

```
tfidf_file = datapath(path + "/LDA_tfidf_151")  
lda_tfidf = LdaModel.load(tfidf_file)
```

Load the dictionary:

```
dict_file = datapath(path + "/gensim_dictionary_AGnews")  
model_dict = Dictionary.load(dict_file)
```

Load the BoW corpus:

```
from gensim import corpora  
bow_corpus = corpora.MmCorpus(path + "/BoW_AGnews_corpus.mm")
```

Let's import the `pyLDAvis` library. Then we will enable the interactive infographics by using `pyLDAvis.enable_notebook()`:

```
import pyLDAvis  
import pyLDAvis.gensim_models as gensimvis  
pyLDAvis.enable_notebook()
```

We will use the `gensimvis.prepare()` function to produce the infographic:

```
LDAvis_outcome = gensimvis.prepare(lda_bow, bow_corpus,  
model_dict)
```

This function transforms the topic model distributions into the data structures needed for the visualization. This function takes three inputs: the LDA model `lda_bow`, the data object (`bow_corpus`), and the dictionary (`gensim_dictionary`). The object is saved in `LDAvis_outcome`. We will see it next.

## The interactive graph

The infographic has the **intertopic distance map** on the left and the **Top-30 Most Relevant Terms** for a selected topic. The intertopic distance map shows the similarities and differences between topics. The distance map is created via **multidimensional scaling (MDS)**. The MDS method is a way to visualize the similarities of data in a 2D space. In our case, the similarity scores between topics are prepared in a large symmetrical matrix where the rows and columns are the topics. MDS decomposes the large matrix to low dimensions by selecting the top  $k$  eigenvalues (e.g. two for a 2D representation) and their corresponding eigenvectors.

Each bubble in the distance map is a topic, and the size of a bubble represents the number of documents in that topic. When you hover your mouse over a bubble, the distribution of words for the selected topic is shown on the right. Let's inspect Topic 1 first:

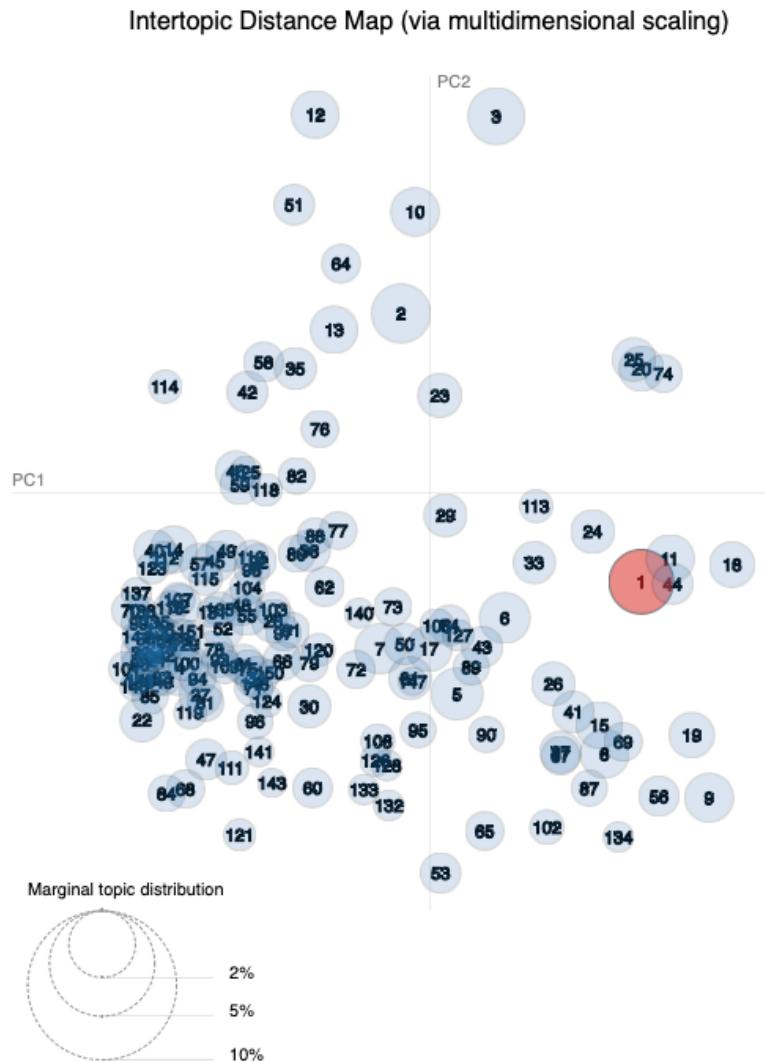


Figure 12.1 – An overview of the intertopic distance map with Topic 1 selected

The scale of the bubble size is shown in the lower-left corner and titled **Marginal topic distribution**. It represents the percentage of documents on that topic in the corpus. It is calculated as the topic proportions given the topic distribution and the document lengths. The distance between any two bubbles represents the similarity difference between the two topics.

Let's inspect the distribution of words for Topic 1, as shown in *Figure 12.2*. The title is **Top-30 Most Relevant Terms for Topic 1 (1.9% of tokens)**. It means 1.9% of the words in the entire corpus are associated with Topic 1. The top 30 most relevant terms for Topic 1 are "reuter," "fullquot," "corn," and so on, in descending order. The red bars represent the estimated frequency of a word. For example, the word "reuter" has appeared almost 5,000 times.

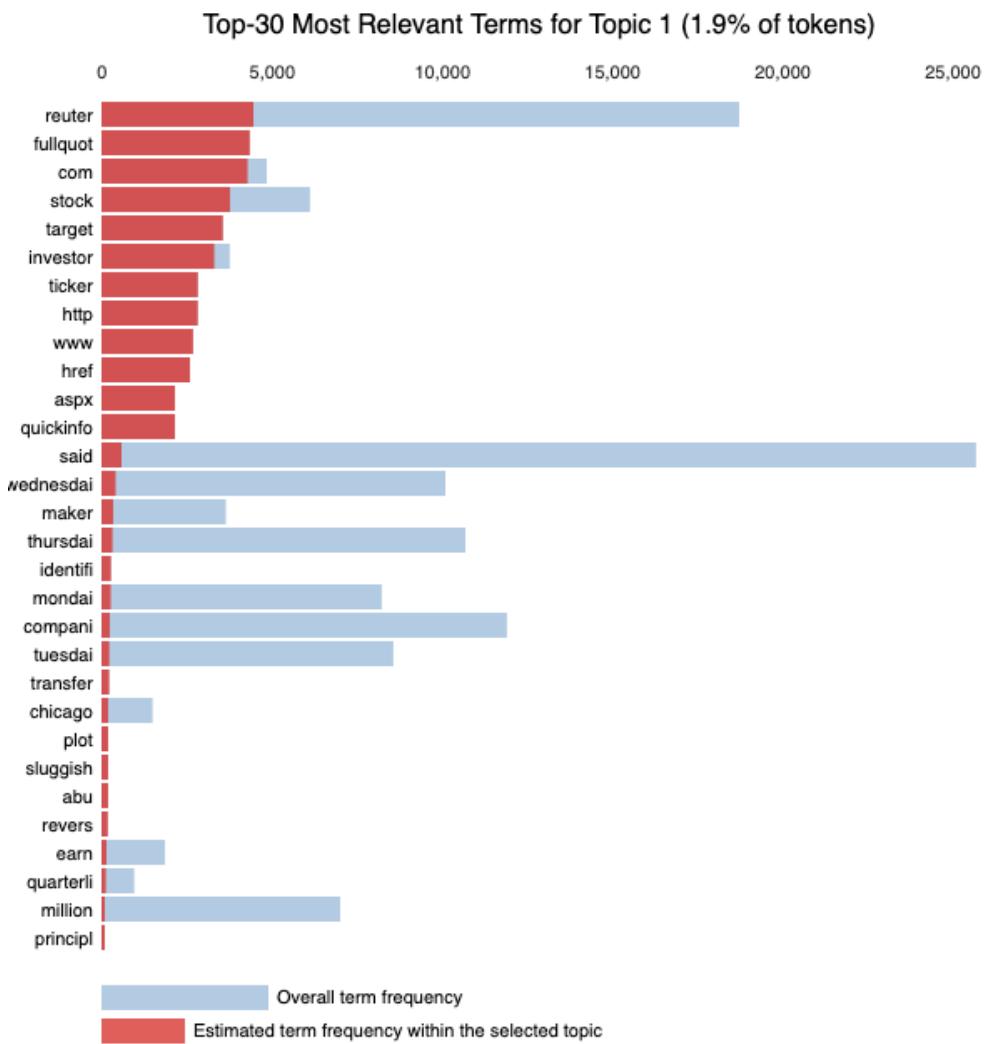


Figure 12.2 – An overview of the top-30 most relevant terms for Topic 1

The light blue bars represent the overall frequency of a word in the corpus. For example, the word “reuter” appears about 18,000 times in the entire corpus, and the word “said” appears close to 25,000 times. The contrast between the length of the red and light blue bars for a word is helpful. It shows the coverage of a topic for words. For instance, the words “corn,” “stock,” and “investor” in the entire corpus are pretty much covered by Topic 1. It means Topic 1 refers to financial news about “corn,” “stock,” and “investor.”

Let's look at another topic. *Figure 12.3* shows Topic 12 in red. The rest of the bubbles are unchanged.

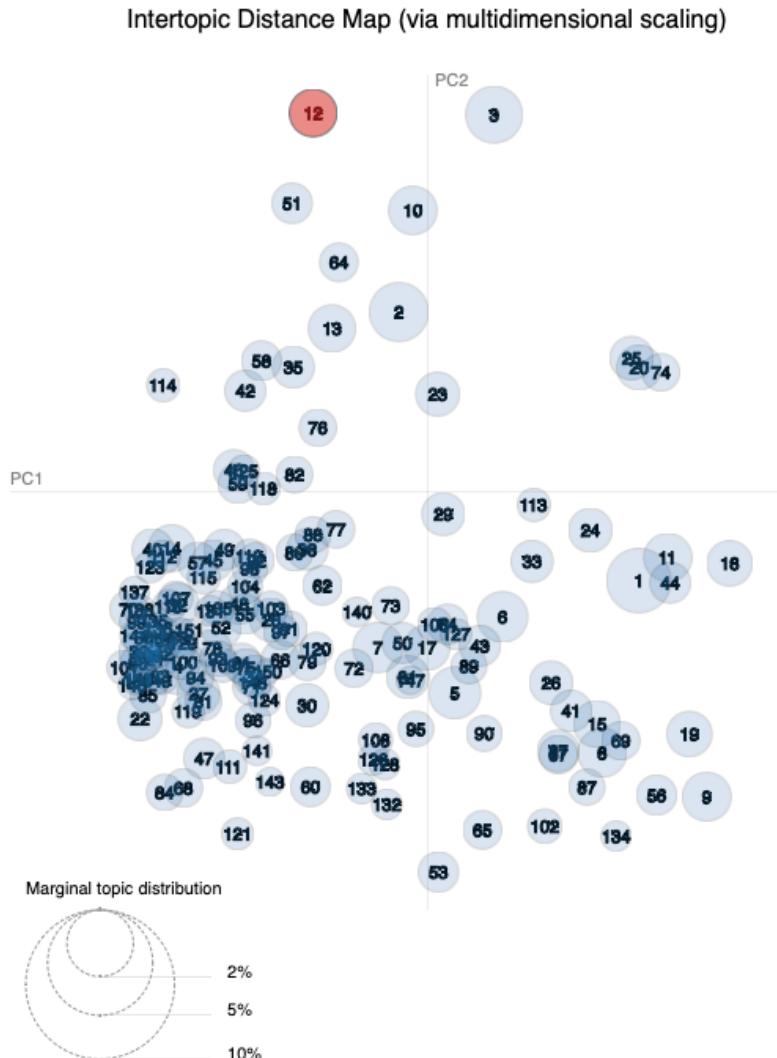


Figure 12.3 – An overview of the intertopic distance map with Topic 12 selected

Now let's see the distribution of words, as shown in *Figure 12.4*. The top words for Topic 12 are "busi" (business), "manag" (manage), "come," "ibm," "system," "custom," "compani" (company), and so on. This tells us Topic 12 is about business news, especially tech companies.

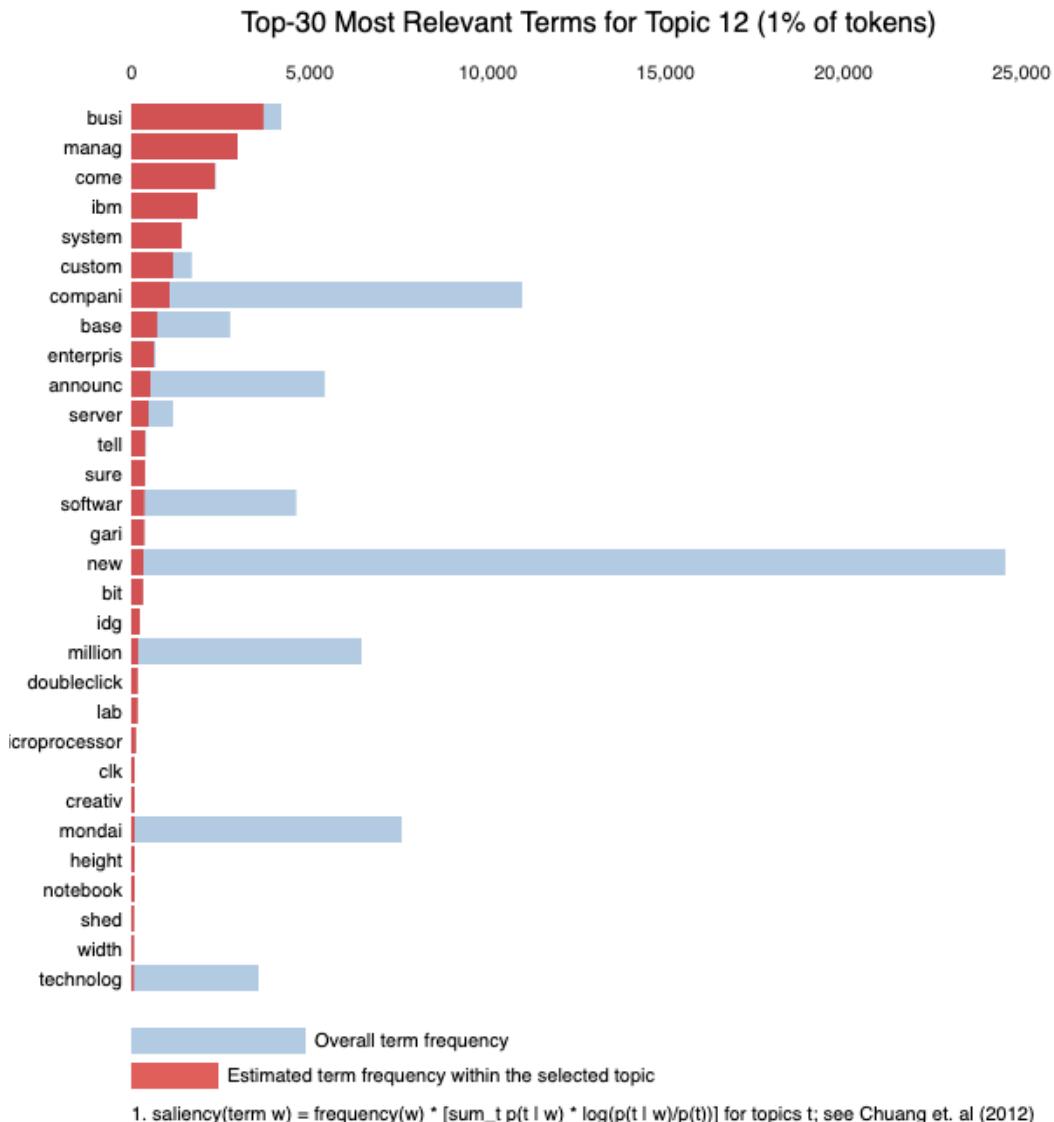


Figure 12.4 – An overview of the top-30 most relevant terms for Topic 12

Finally, the interactive infographic can be saved as an HTML file by using `save_html()`:

```
pyLDAvis.save_html(LDAvis_outcome, path + "/gensim_LDA_AGnews.html")
```

The saved file will enable you to inspect many topics in the infographic.

## Summary

This chapter focused on how to design an infographic to deliver very rich content. LDA topic models result in a set of topics and every topic has a distribution of words. How should we design such an infographic? When we visualize the LDA results, we first want to know the size of a topic, i.e., the percentage of documents for that topic. Then we want to know the similarities or differences between topics. This can be shown by the distances between topics. Then we want to see the distribution of words. It will be ideal to see the distribution of words in the entire corpus, and then be able to choose a topic to see the distribution of words for that topic.

The `pyLDAvis` library facilitates well-designed interactive infographics. It lets us show the similarities and differences between topics. It shows the distribution of words in the entire corpus, then it lets you choose a topic to see the distribution of words for the topic.

What are other ways to conduct topic modeling? With the arrival of transformer-based models, BERTTopic modeling provides a new way to do topic modeling. In the next chapter, we will learn about BERTTopic modeling.

## Questions

1. What are the essential elements of an LDA outcome?
2. What is the marginal topic distribution?

## References

1. Sievert, C., & Shirley, K.E. (2014). *LDAvis: A method for visualizing and interpreting topics*.
2. `pyLDAvis` (<https://pypi.org/project/pyLDAvis/>)

# 13

## The Ensemble LDA for Model Stability

One of the success criteria of topic modeling is to produce a reliable set of topics. However, many experiments with **Latent Dirichlet Allocation (LDA)** have shown that the topics can be unstable and not reproducible. This issue seriously limits the applications of LDA. The instability of the topic results is partly due to the fact that the model settles at a local maximum depending on the random initialization. Even if a seed number is set to control random initialization, noisy topics can be generated during the modeling process, which might influence the quality of the outcome.

The root cause of the instability is that a single LDA model identifies the “true” topics and “pseudo” topics and produces noisy predictions. If the model is trained again, it will identify “true” topics and other “pseudo” topics. The solution is to build multiple models or an ensemble of models to weed out the pseudo topics. By training an ensemble of topic models and throwing out topics that do not reoccur across the ensemble, the Ensemble LDA model can deliver stable outcomes. Furthermore, the Ensemble LDA model can identify the number of topics for a stable outcome, so that the user does not need to determine the exact number of topics ahead of time.

How does Ensemble LDA evolve from LDA? How does Ensemble LDA determine the approximate number of topics? We will cover the following topics in this chapter:

- From LDA to Ensemble LDA
- The process of Ensemble LDA
- Understanding DBSCAN and **Checkback DBSCAN (CBDBSCAN)**
- Building an Ensemble LDA model with Gensim

By the end of this chapter, you will understand how the ensembling method in LDA identifies the “true” topics. You will be able to describe the Ensemble LDA algorithm, including its use of CBDBSCAN. You will build your Ensemble LDA model and score new documents.

## Technical requirements

You will need to install the `gensim` module using the following command:

```
pip install gensim
```

The code files are available at <https://github.com/PacktPublishing/The-Handbook-of-NLP-with-Gensim/tree/main/Chapter13>.

## From LDA to Ensemble LDA

Suppose a corpus has three distinct words, and the three words belong to three topics. This idea is shown in *Figure 13.1*, in which the vertices of the simplex are the three words. The three topics are labeled as **Topic A**, **Topic B**, and **Topic C** in the left simplex. However, LDA may identify a fourth topic from the combination of the three topics. It is a “pseudo” topic, as shown in the middle of the simplex.

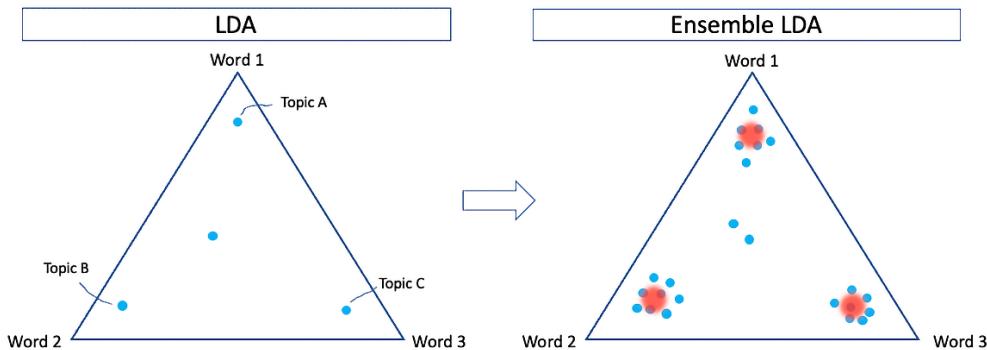


Figure 13.1 – Applying the ensembling method to LDA

Let's take an ensembling approach by building many LDA models on the same data. Most of the LDA models will have topics A, B, and C, and some other LDA models will produce pseudo topics in addition to topics A, B, and C. The “true” topics A, B, and C shall appear more frequently and the “pseudo” topics shall appear less frequently. This idea is demonstrated in the simplex on the right-hand side of *Figure 14.1*. All the blue dots are the topics produced by the ensemble of models. There are many dots clustered around the “true” topics A, B, and C. There are a few standalone dots that are pseudo topics. The Ensemble LDA model can identify three topic clusters as shown by the three red dots. So even if a single LDA model may produce pseudo topics, the Ensemble LDA model will reliably identify the true topics that frequently appear in the ensemble of models.

Next, let me explain how this works.

## The process of Ensemble LDA

The Ensemble LDA algorithm is a combination of LDA and CBDBSCAN. Let's break the process involved down into steps:

1. **Text preprocessing:** Perform any necessary preprocessing steps such as tokenization, stop-word removal, and stemming on the documents.
2. **LDA training:** Build multiple LDA models on the document collection using different random initializations.
3. **Topic assignment:** For each document in the collection, assign a topic distribution based on the trained LDA models. This can be done by calculating the probability of each topic for the document using the LDA models.
4. **CBDBSCAN:** Apply the CBDBSCAN algorithm to cluster the documents based on their assigned topics. CBDBSCAN is an extension of the DBSCAN algorithm that incorporates a checkback step to refine the clustering results. We will learn about DBSCAN and CBDBSCAN in the next section.
5. **Output:** The output of the algorithm is a set of clusters, where each cluster represents a group of documents with similar topics.

By using Ensemble LDA with CBDBSCAN, the algorithm can capture more diverse and complex topics in the data, and produce more accurate and robust results. Next, let's learn about the DBSCAN and CBDBSCAN algorithms.

## Understanding DBSCAN and CBDBSCAN

DBSCAN stands for **Density-Based Spatial Clustering of Applications with Noise**. It is a popular clustering algorithm based on the density of data points to identify clusters. **Checkback DBSCAN (CBDBSCAN)** is an extension of DBSCAN and is employed in Ensemble LDA. Let's first learn about DBSCAN, then why the extension is needed in CBDBSCAN.

### DBSCAN

DBSCAN, an unsupervised machine learning algorithm, is often used for clustering data points based on their density and proximity to each other. Before learning about the algorithm, let's get familiar with a few terms in DBSCAN. The first is **epsilon**. It is a parameter that controls the maximum distance between data points in a cluster. The value of epsilon is set before running the algorithm. It should be small enough to capture the density of the clusters but not so small that it creates too many clusters. The second term is **minPts**. It is the minimum number of neighbors required for a point to be considered a core point. This leads us to learn about the third term: **core points**. Core points are data points that have at least minPts neighbors within a distance of epsilon. These points are the densest points in the dataset.

Here's a high-level overview of the algorithm:

1. **Initialize the dataset:** The algorithm starts by initializing the dataset with all data points.
2. **Identify core points:** The algorithm identifies all data points that have at least minPts neighbors within a distance of epsilon. These points are marked as **core points**.
3. **Expand the cluster:** The algorithm expands the cluster by identifying all data points within a distance of epsilon from each core point. These points are marked as **border points**.
4. **Contract the cluster:** The algorithm contracts the cluster by removing all border points that do not have any neighbors within a distance of epsilon.
5. **Repeat steps 3 and 4:** The algorithm repeats steps 3 and 4 until no new points are added to the cluster.
6. **Identify noise points:** The algorithm identifies all data points that do not have any neighbors within a distance of epsilon. These points are marked as **noise points**.
7. **Output the clusters:** The algorithm outputs the clusters, which are represented by the core points and border points.

This process is demonstrated in *Figure 13.2*. On the left side of *Figure 13.2* is the set of data points to be clustered. If you look carefully, you may have noticed there is an outlier at the top, and another outlier at the bottom. The middle of *Figure 13.2* shows DBSCAN forming the epsilon-radius for each data point and expanding a cluster until it includes all reachable points. This algorithm can find clusters of any shape and size and its computation is fast, making it a great choice in data science.

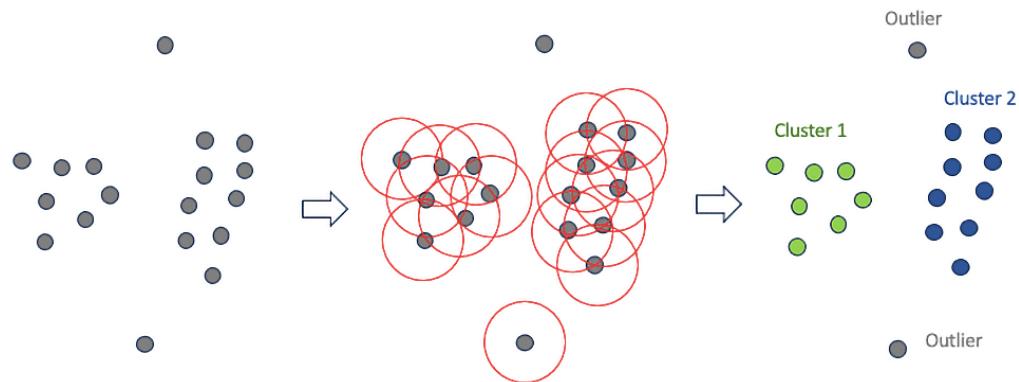


Figure 13.2 – The process of DBSCAN

To understand how DBSCAN works for data of any shape, the website <https://www.naftaliharris.com/blog/visualizing-dbscan-clustering/> presents the dynamic clustering process of DBSCAN. A smiley face is a good choice for data of any shape. DBSCAN starts from a point randomly and then develops according to the smiley face in *Figure 13.3*. Once the circle

is completed and there is no way to go to the remaining points in the eyes or the smile, it again starts with a point randomly. After completion, it identifies four clusters in this case.

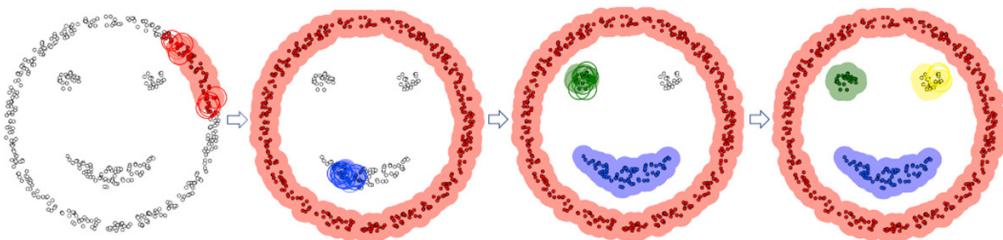


Figure 13.3 – DBSCAN can cluster data points of any shape

DBSCAN applies a fixed radius and a minimum number of neighbors. However, the data density can vary in the core data areas and the areas of noise or outliers. CBDBSCAN is therefore designed to better handle varying data densities.

## CBDBSCAN (Checkback DBSCAN)

CBDBSCAN is a variation of DBSCAN. DBSCAN requires the use of fixed values for the radius and a minimum number of neighbors. This approach can be problematic if data has varying densities or noise levels. The epsilon and minPts parameters may vary if data has varying densities. CBDBSCAN addresses this issue by introducing a dynamic approach to handle noise and outliers. Instead of using a fixed radius, CBDBSCAN uses an **adaptive radius** that is adjusted based on the local density of the data. The algorithm works by expanding the cluster boundary iteratively to include points that are close to the current boundary, and then checking whether the expanded boundary captures a sufficient number of points to be considered a dense region. If the expanded boundary does not capture enough points, the algorithm shrinks the boundary back to the previous size and tries again. This process continues until the algorithm converges on a stable cluster boundary that captures a dense region of the data. The result is a set of clusters that are less susceptible to noise and outliers compared to traditional DBSCAN.

The main steps of the CBDBSCAN algorithm are as follows:

1. Initialize the cluster sequence and the density threshold.
2. Iterate through the dataset, and for each point, find the nearest neighbors using the chosen distance metric.
3. Assign the point to the nearest cluster based on the distance metric and the density threshold epsilon.
4. Perform the checkback. During the clustering process, reassess the density of the points in the neighborhood and adjust the density threshold epsilon accordingly.
5. Repeat steps 2-4 until all points have been assigned to clusters.

The design of CBDBSCAN can thus result in more accurate cluster assignments. It is robust to noise in the data, as it adaptively adjusts the density threshold based on the density of the points in the neighborhood. CBSBSCAN has been shown to be effective in clustering data with varying densities and noise levels and has been applied in a variety of domains. Note that CBDBSCAN can be more computationally expensive than DBSCAN and may not scale well for very large datasets.

Knowing how Ensemble LDA and CBDBSCAN work, let's dive into the Python code to build our model.

## Building an Ensemble LDA model with Gensim

The procedure is very similar to that of building the LDA model in the previous chapter. The Ensemble LDA requires text preprocessing.

### Preprocessing the training data

Let's load the training data:

```
import pandas as pd
import numpy as np
pd.set_option('display.max_colwidth', -1)
path = '/content/gdrive/My Drive/data/gensim'
train = pd.read_csv(path + "/ag_news_train.csv")
```

We will tokenize the words in each sentence:

```
from gensim.parsing.preprocessing import preprocess_string
text_tokenized = []
for doc in train['Description']:
    k = preprocess_string(doc)
    text_tokenized.append(k)
text_tokenized[0:3]
```

The output looks like this:

```
[['reuter', 'short', 'seller', 'wall', 'street', 'dwindl', 'band',
'ultra', 'cynic', 'see', 'green'],
['reuter', 'privat', 'invest', 'firm', 'carlyl', 'group', 'reput',
'make', 'time', 'occasion', 'controversi', 'plai', 'defens',
'industri', 'quietli', 'place', 'bet', 'market'],
['reuter', 'soar', 'crude', 'price', 'plu', 'worri', 'economi',
'outlook', 'earn', 'expect', 'hang', 'stock', 'market', 'week',
'depth', 'summer', 'doldrum']]
```

## Creating text representation with BOW and TF-IDF

The text representation using BOW is done as follows:

```
from gensim.corpora import Dictionary
gensim_dictionary = Dictionary()
bow_corpus = [gensim_dictionary.doc2bow(doc,
                                         allow_update=True) for doc in text_tokenized]
print(bow_corpus[:3])
id_words = [[(gensim_dictionary[id], count) for id,
             count in line] for line in bow_corpus]
print(id_words)
```

Likewise, the text representation using TF-IDF is done with the following code:

```
from gensim.models import TfidfModel
tfidf = TfidfModel(bow_corpus) #, smartirs='npu')
tfidf_corpus = tfidf[bow_corpus]
print(tfidf_corpus[:3])
```

The output is a list of word IDs and counts:

```
[(0, 1), (1, 1), (2, 1), ...]
id_words = [[(gensim_dictionary[id], count) for id,
             count in line] for line in bow_corpus]
print(id_words[0:3])
[('band', 1), ('cynic', 1), ('dwindl', 1), ...]
```

## Saving the dictionary

Later, when we use the model to score new documents, we will need the dictionary and the BOW or TF-IDF. So, we save the dictionary, BOW, and TF-IDF now:

```
from gensim.test.utils import datapath
dict_file = datapath(path + "/gensim_dictionary_AGnews")
gensim_dictionary.save(dict_file)
```

Gensim's corpora class offers the .MmCorpus() function that saves matrix data in the **Matrix Market (MM)** exchange format. The MM format is a basic ASCII file format for matrix data. We will use it to save the BOW and TF-IDF data:

```
from gensim import corpora
```

Save it in the MM format:

```
corpora.MmCorpus.serialize(path + "/BoW_AGnews_corpus.mm",
                           bow_corpus)
```

This is how you load an .mm file:

```
bow_corpus = corpora.MmCorpus(path + "/BoW_AGnews_corpus.mm")
bow_corpus[0]
```

The output is as follows:

```
[(0, 1.0), (1, 1.0), (2, 1.0), (3, 1.0), (4, 1.0), (5, 1.0), (6, 1.0),
 (7, 1.0), (8, 1.0), (9, 1.0), (10, 1.0)]
```

You can also use the Python `pickle` tool to dump and load the BOW or TF-IDF data:

```
import pickle
file = open(path + "/BoW_AGnews_corpus.pkl", 'wb')
pickle.dump(bow_corpus, file)
file.close()
import pickle
# open a file, where you stored the pickled data
file = open(path + "/BoW_AGnews_corpus.pkl", 'rb')
bow_corpus = pickle.load(file)
# close the file
file.close()
bow_corpus[0]
```

## Building the Ensemble LDA model

In LDA, the number of topics needs to be assigned externally as a hyperparameter. In Ensemble LDA, the number of topics will be determined internally. By running multiple LDA models, the Ensemble LDA model can extract stable topics that exist in multiple LDA models. Ensemble LDA has the benefit that we do not need to specify the exact number of topics:

```
from gensim.corpora.dictionary import Dictionary
from gensim.models import ensemblelda
```

Although we will still set the number of topics, as shown later in the code, this determines the *maximum* number of topics, but not the *final* number of topics:

```
elda_bow = ensemblelda.EnsembleLda(corpus=bow_corpus,
                                      id2word=gensim_dictionary, num_topics=600)
elda_tfidf = ensemblelda.EnsembleLda(corpus=tfidf_corpus,
                                       id2word=gensim_dictionary, num_topics=600)
```

The training process can take a while. The best practice is to save the model for future use:

```
from gensim.test.utils import datapath
```

Save the eLDA model trained on the BOW data:

```
elda_bow_file = datapath(path + "/eLDA_bow_Agnews")
elda_bow.save(elda_bow_file)
```

Save the eLDA model trained on the TF-IDF data:

```
elda_tfidf_file = datapath(path + "/eLDA_tfidf_Agnews")
elda_tfidf.save(elda_tfidf_file)
```

The model identifies 20 topics automatically. Let's see the topics with the following command:

```
import pprint as pp
pp pprint(elda_bow.print_topics())
```

The output could be long. Here are a few examples:

```
[(29,
  '0.282*"phone" + 0.272*"mobil" + 0.045*"servic" + 0.040*"oper" +
  0.036*"new" +
  '+ 0.029*"firefox" + 0.026*"mozilla" + 0.025*"market" +
  0.017*"compani" +
  '0.016*"docomo"' ),
 (8,
  '0.426*"research" + 0.148*"profil" + 0.105*"quot" + 0.066*"said" +
  '0.024*"tuesdai" + 0.022*"corp" + 0.020*"mondai" + 0.017*"thursdai"
  +
  '0.016*"vendor" + 0.014*"carl"' ),
 (71,
  '0.442*"england" + 0.058*"mayor" + 0.044*"new" + 0.038**"bloc" +
  '0.035**"squad" + 0.033**"flow" + 0.028**"tourist" + 0.027**"anfield" +
  ,
  '0.026**"world" + 0.026**"pact"' ),
 (76,
  '0.440**"show" + 0.053**"govern" + 0.047**"opec" + 0.036**"wire" +
  0.035**"shake" +
  '+ 0.032**"miner" + 0.031**"output" + 0.029**"report" + 0.027**"gene" +
  ,
  '0.024**"juli"' ),
 (10,
  '0.546**"court" + 0.136**"appeal" + 0.046**"lawsuit" + 0.039**"auction"
  +
  '0.026**"ebai" + 0.019**"thursdai" + 0.016**"fridai" + 0.014**"burn" +
  '
```

```

'0.010*"million" + 0.010*"yesterdai"' ),
(23,
'0.634*"union" + 0.083*"verizon" + 0.037*"countri" + 0.030*"nigeria"
+
'0.026*"falcon" + 0.020*"nation" + 0.020*"said" + 0.019*"plan" +
'0.018*"gotten" + 0.013*"world"' ),
(78,
'0.507*"champion" + 0.059*"pace" + 0.058*"world" +
0.058*"tournament" +
'0.048*"hockei" + 0.042*"qualifi" + 0.036*"tomorrow" + 0.032*"stun"
+
'0.027*"even" + 0.024*"netherland"' ),
...
]
```

Remember that this Ensemble LDA topic already removed the pseudo topics, leaving only the true and reliable topics. Since they are reliable topics, they are reproducible as well. Next, let's learn how to use the saved Ensemble LDA model to score new documents.

## Scoring new documents

Let's use the Ensemble LDA model to score new documents. Suppose there are two new documents with meaningful words: ['champion', 'hockey', 'qualify'] and ['survey', 'tournament', and 'world']:

```

other_texts = [
    ['champion', 'hockey', 'qualify'],
    ['survey', 'tournament', 'world']
]
```

We will create a new corpus by applying the saved dictionary to these new documents:

```

other_corpus = [gensim_dictionary.doc2bow(text) for
                 text in other_texts]
```

Let's take the first document:

```
unseen_doc = other_corpus[0]
```

Let's score it:

```
vector = elda_bow[unseen_doc]
```

Notice the words in the new document [‘champion’, ‘hockey’, ‘qualifi’] are about sports tournaments. If we look at the previously printed output of `elda_bow.print_topics()`, we can guess this document belongs to topic 78. Indeed, when we print out the prediction for this new document, it is topic 78:

```
pp.print(vector)
```

The preceding command returns the topic number and the probability of the prediction:

```
[ (78, 0.7521008) ]
```

I trust you have enjoyed the model-building process. Let’s summarize the chapter.

## Summary

In this chapter, we explored the reasons for the instability and irreproducibility of LDA. The root cause of the instability is that a single LDA model identifies “true” topics and “pseudo” topics and produces noisy predictions. We learned how we can benefit from Ensemble LDA models that can deliver stable outcomes. Ensemble LDA involves training an ensemble of topic models and throwing out topics that do not reoccur across the ensemble. This ensembling method can differentiate “pseudo” topics from “true” topics. Ensemble LDA keeps the “true” topics as the final topics. In addition, we learned how CBDBSCAN develops clusters. We then built some Ensemble LDA models and examined them with new documents.

With the arrival of the transformer algorithm, even more topic-modeling algorithms have been developed in NLP, the most prominent of which is BERT-based topic modeling. In the next chapter, we will explain what BERTopic is, identify its features, and compare these with the LDA algorithm.

## Questions

1. What is the root cause of topic instability in an LDA model?
2. What is the strategy of Ensemble LDA to overcome the model instability of LDA?
3. Describe how DBSCAN works.
4. Describe how CBDBSCAN extends DBSCAN.
5. Describe any text preprocessing methods that can be used for Ensemble LDA.

## References

1. BRIGL, Tobias, 2019, Extracting Reliable Topics using Ensemble Latent Dirichlet Allocation [Bachelor Thesis]. Technische Hochschule Ingolstadt. Munich: Data Reply GmbH. Supervised by Alex Loosley. Available at [https://www.sezanzeb.de/machine\\_learning/ensemble\\_LDA/](https://www.sezanzeb.de/machine_learning/ensemble_LDA/)



# Part 5: Comparison and Applications

In this part, you will understand how the ongoing development of LLMs has contributed significantly to topic modeling for document classification. You will learn about the BERTopic modeling, a milestone topic modeling method that adopts the BERT word embeddings. You will build a BERTopic model and understand its differences with LDA.

This unit then surveys seven NLP applications in the medical, healthcare, social media, legal, and finance fields. The techniques include Word2Vec, LSA, LDA, and BERTopic. In each use case, you will understand the professional background, the need for an NLP model, and their solutions.

This part contains the following chapters:

- *Chapter 14, LDA and BERTopic*
- *Chapter 15, Real-World Use Cases*



# 14

## LDA and BERTopic

Since the Transformer model came to the NLP stage in 2017 in the seminar paper *Attention Is All You Need* [1], many Transformer-based **large language models (LLMs)** such as **BERT (Bidirectional Encoder Representations from Transformers)** [3], ChatGPT, and GPT-4 [4] have seized the technology headlines. The word embeddings by these LLMs can discover more latent semantic relationships between words and documents than those by pre-LLM techniques such as BoW, TF-IDF, or Word2Vec.

The semantic relationships between words and documents naturally extend to document grouping, which is the aim of topic modeling that clusters documents into homogeneous document groups. Can we take advantage of the word embeddings of LLMs for topic modeling? This advantage motivates research in LLMs for topic modeling. An important topic modeling technique of this line is called **BERTTopic**. It adopts the BERT word embeddings and includes multiple techniques such as UMAP, HDBSCAN, c-TFIDF, and MMR to create clusters. Its name speaks for itself – it is a topic modeling technique using the BERT word embeddings. BERTTopic has become a strong competitor to LDA for topic modeling. Given our interest in topic modeling and LDA in this book, we will explore BERTTopic.

The techniques used by BERTTopic are advanced. The goal of this chapter is to provide simple descriptions of the Transformer model to BERT and all the mentioned techniques. This plan will equip you to appreciate the elegance of the use of these techniques. This chapter will address the following topics:

- Understanding the Transformer model
- Understanding BERT
- Describing how BERTTopic works
- Building a BERTTopic model
- Reviewing the results of BERTTopic
- Visualizing the BERTTopic model
- Predicting new documents
- Using the modular property of BERTTopic
- Comparing BERTTopic with LDA

By the end of this chapter, you will be able to build, interpret, and visualize a BERTopic model and know how to use the model to score new documents. You will understand the modularity of the BERTopic technique and be able to use specific techniques for a module. Finally, you will be able to articulate a few noticeable differences between BERTopic and LDA.

## Technical requirements

The code files are available on GitHub: <https://github.com/PacktPublishing/The-Handbook-of-NLP-with-Gensim/tree/main/Chapter14>.

## Understanding the Transformer model

Automatic language translation is an important research topic in NLP. In recent years, sequence-to-sequence models, such as **recurrent neural networks (RNNs)**, **long short-term memory (LSTM)**, and **gated recurrent units (GRUs)** have been proven effective [2]. The sequence “I just love eating Doritos” in English can be translated instantly by a sequence-to-sequence model to “Ich liebe es einfach, Doritos zu essen” in German. RNNs, LSTM, and GRUs suffer from limitations in capturing long-range dependencies in sequential data due to their short-term memory and sequential processing nature. The need for sequential computation results in long computational time as well. The Transformer architecture addresses these weaknesses through a self-attention mechanism. The self-attention mechanism in a Transformer is like looking at each word and deciding how much attention it should give to the other words. It enables the Transformer to understand how words in a sentence or story are connected, and it does this for every word in the text. This helps the Transformer make sense of long and complex sentences, making it great for tasks such as language translation or understanding context in text.

The Transformer model is a deep learning model that adopts the mechanism of self-attention, which is based on the attention mechanism, differentially weighting the significance of each part of the input data. Unlike RNNs, it processes the entire input sequences all at once. Because the Transformer model is instrumental, I would like to provide a little bit more detail in *Figures 14.1 to 14.3*, which lead to *Figure 14.4*, which is the complete structure in the original paper [1].

*Figure 14.1* shows the Transformer as a large black box that takes the input Hello world! and translates it into Hola Mundo!. This supervised learning model is trained on the pairs of input and output sentences. Notice it takes a real sentence without any text preprocessing such as stop word removal or lemmatization.



Figure 14.1 – The Transformer in a very simple form

Let's open the black box in *Figure 14.2*. It shows the Transformer is composed of encoders and decoders. The encoders take an input sequence to produce a sequence of latent values and then feed them to the decoders. The encoders (the orange box) are six identical encoders. They encode the input data to features. The decoders (the green box) are six identical decoders that decode the features to another language.

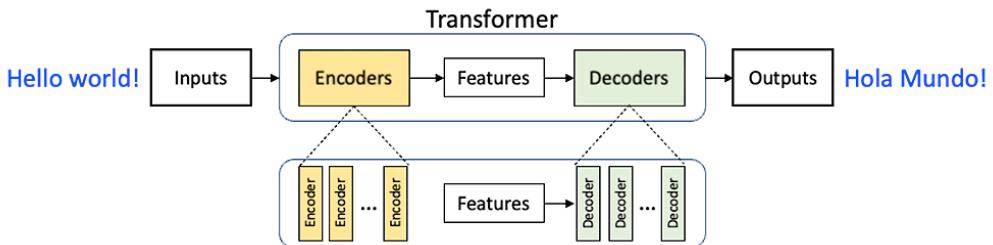


Figure 14.2 – The Transformer showing the encoders and decoders

*Figure 14.3* just reorganizes the previous flowchart vertically to make it similar to the graph in the paper.

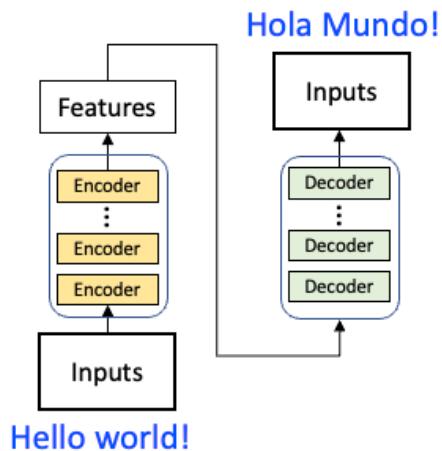


Figure 14.3 – An overview of the Transformer presented vertically

*Figure 14.4* reorganizes the previous flowchart vertically to make it similar to the graph in the paper. The left half of the architecture is the encoding process, and the right half is the decoding process. The left half only draws one encoder to show the inside of the encoder. Each encoder is composed of two sublayers. The first one is a multi-head self-attention mechanism, and the second sublayer is a fully connected feed-forward neural network. The six connected encoders will have different weights, although they are identical. Similarly, the six decoders apply the same structure to encoders.

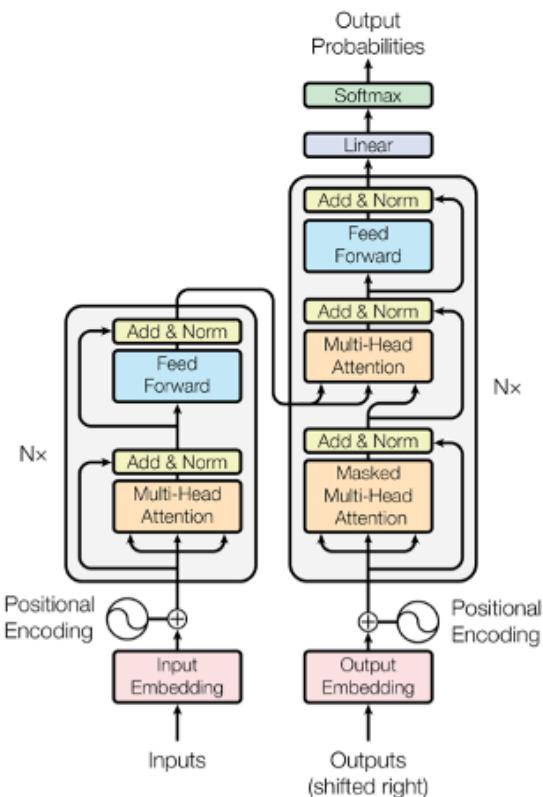


Figure 14.4 – An overview of the full Transformer diagram

With this knowledge of the Transformer model, let's now extend it to BERT.

## Understanding BERT

BERT was published in 2019 by Devlin et al. based on the Transformer architecture [3]. It soon became the prevailing model in NLP. It helps the Transformer model to become even smarter. BERT teaches the Transformer to learn from the words before and after each word, so it knows the context and order better. This helps the Transformer understand tricky things such as jokes or words with multiple meanings, making it excellent at understanding all kinds of text, such as chatting or reading books.

How does it do that? BERT removes the unidirectionality constraint in the Transformer model and uses a **masked language model (MLM)** that randomly masks some of the input tokens. Since some tokens are masked, MLM has to predict the original vocabulary of the masked word based on its before and after context. To use the context before and after the masked word, MLM fits the data by using both left-to-right and right-to-left contexts. This is why it is called **bidirectional**, in contrast

with a Transformer, which uses the left-to-right direction. The results are word embeddings that can comprehend words with multiple meanings. Let's give an example of the BERT word embeddings that can distinguish words in different contexts. The word "crane" in the following three sentences has three different meanings:

- That bird is a crane
- They had to use a crane to lift the object
- She had to crane her neck to see the movie

The "crane" in the first sentence is a bird. It is a machine in the second sentence. In the third sentence, it is a verb to stretch her neck to see the movie. The BoW, TF-IDF, or Word2Vec techniques may interpret all instances of "crane" as being the same and yield the same word embedding. But BERT will embed "crane" differently for each sentence. With this understanding of BERT, we are ready to learn about BERTTopic.

## Describing how BERTTopic works

BERTTopic uses the BERT word embedding vectors for topic modeling [5]. A document is first to be embedded into word vectors. The high-dimensional word vectors then go through dimensionality reduction in order to be clustered into topics. BERTTopic has a sequence of five modular components, as shown in *Figure 14.5*. The five modules are designed to be as independent as possible so that data scientists can choose an alternative technique for a module. For instance, the clustering method HDSCAN can be replaced with K-means. These techniques are the default components for BERTTopic. At the end of the chapter, I will illustrate how to model with their alternative techniques.

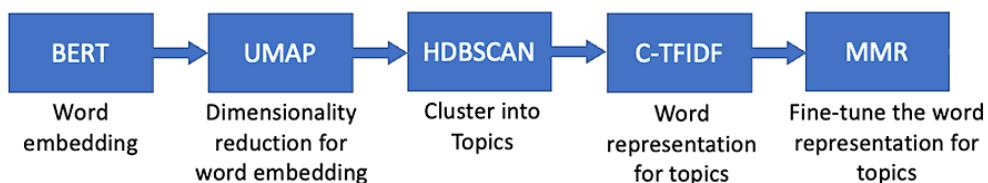


Figure 14.5 – BERTTopic structure

Let's review each technique in the next sections.

### BERT – word embeddings

The first block converts a document into numerical representations. BERTTopic uses the BERT-based word embeddings of paraphrase-MiniLM-L6-v2 (<https://huggingface.co/sentence-transformers/paraphrase-MiniLM-L6-v2#:%text=sentence-transformers%2Fparaphrase-MiniLM-L6-v2%20This%20is%20a%20sentence-transformers%20model%3A%20It%20maps,tasks%20like%20clustering%20or%20semantic%20search.%20Usage%20%28Sentence-Transformers%29>) for

English and paraphrase-multilingual-MiniLM-L12-v2 (<https://huggingface.co/sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2>) for multiple languages [6]. These word embedding models are sentence-based transformers and are optimal for sentence and paragraph embeddings. They map sentences and paragraphs to a 384-dimensional latent vector space. The two models are available on HuggingFace.co. You can find more details about the BERT-based word embeddings on **KeyBERT – Embedding Models – KeyBERT** (<https://maartengr.github.io/KeyBERT/index.html>).

## UMAP – reduce the dimensionality of embeddings

The 384-dimensionality of the word embeddings is indeed very large. BERTopic uses UMAP to reduce the dimensionality of embeddings. **UMAP** stands for **Uniform Manifold Approximation and Projection**. It is a clever way to turn complex data into simpler pictures. Imagine you have a big puzzle with lots of pieces (data points), and you want to arrange them on a board so that similar pieces are close together and different pieces are far apart. UMAP does this by paying attention to how each puzzle piece relates to its neighbors. It keeps the important relationships while squishing the puzzle onto the board, so you can see patterns and groups in the data much more clearly. It's like making a map of your puzzle where the distances show how things are connected in a way that makes sense to your brain.

UMAP is similar to t-SNE but offers a number of advantages over t-SNE. Recall that we learned about t-SNE in *Chapter 7, Using Word2Vec*. UMAP can offer faster speed and can preserve the global structure of data points better than t-SNE [7]. The default dimension of UMAP is five. The reduced dimensionality will greatly help HDBSCAN to create better topics.

## HDBSCAN – cluster documents

HDBSCAN stands for **Hierarchical Density-Based Spatial Clustering of Applications with Noise**. It is an extension of **DBSCAN (Density-Based Spatial Clustering of Applications with Noise)**. We learned about DBSCAN in *Chapter 14, The Ensemble LDA for Model Stability*. It is a powerful clustering algorithm used to identify groups within a dataset, especially when the clusters have irregular shapes and sizes. It works by examining the density of data points in the vicinity of each point. Points that are close to each other and have enough neighbors within a specified radius are considered part of a cluster. This approach allows DBSCAN to automatically find clusters of varying densities and even detect outliers as noise points. It's particularly useful when traditional methods such as K-means struggle with complex data distributions and noise.

HDBSCAN offers more advantages over DBSCAN, especially in scenarios where the data's characteristics are not well known in advance. HDBSCAN is capable of automatically adapting to varying cluster densities and shapes, making it more versatile in identifying clusters. It doesn't require manual specification of a minimum cluster size, which can be a challenge in DBSCAN. Moreover, HDBSCAN distinguishes noise points from clusters more effectively, provides a hierarchical representation for

multi-scale analysis, and is less sensitive to parameter choices, offering a more user-friendly and robust solution for clustering tasks.

BERTopic takes advantage of the HDBSCAN technique to cluster the reduced embeddings. The HDBSCAN technique identifies clusters of documents. These clusters become the topics in the topic modeling.

## c-TFIDF – create a topic representation

Now, we want to interpret a topic by finding the related words. The words in a topic will inform us why a topic is different from another topic. BOW or TF-IDF can be used to do so. Because TF-IDF can show the relative importance of words in documents, it becomes our first choice. Notice that, in BERTopic, we do not use TF-IDF for word embeddings. It is used to represent topics that have been identified by HDBSCAN.

Why does BERTopic use **c-TFIDF (class-based TF-IDF)** but not TF-IDF? We know TF-IDF is designed to show the relative importance of words in documents. However, documents can belong to different categories so the relative importance of words can vary by the class of a document. For example, words in the “astronomy” class are likely to be *mission, moon, earth, lunar, nasa, space, and shuttle*, and words in the “vehicle” class are likely to be *speed, driving, tires, ford, oil, engine, car*, and so on. c-TFIDF defines the word frequency as the word frequency in each class. It preserves the word strength per class. BERTopic uses the **class-based TF-IDF (c-TFIDF)** to generate better topic representations.

## Maximal Marginal Relevance

**Maximal Marginal Relevance (MMR)** is a technique used in text summarization to select the most relevant and diverse set of words. MMR is employed in BERTopic to select representative documents for each topic in a way that maximizes the relevance of the selected documents to the topic while minimizing redundancy with documents already assigned to other topics. This helps to create a more informative and diverse set of documents for each topic. By incorporating MMR, BERTopic addresses the challenge of producing coherent and non-repetitive topics, resulting in a more accurate representation of the underlying themes in the data.

We have explained each component of BERTopic. Now is the time to build a BERTopic model!

## Building a BERTopic model

Because BERTopic is a Transformer-based model, in general, there is no need to preprocess the texts such as with stop word removal or lemmatization. Keeping the original structure of the text is important in the Transformer-based approach. Stop words are usually non-informative. If a document has a lot of stop words such as *he, she, and they*, the document is likely to have the non-informative topic -1, which we will see shortly. That being said, nowadays, many texts have typos and nouns can be singular or plural; the outcome of a BERTopic model on an unlemmatized corpus may have

redundant keywords such as *court* and *courts*, or *cup* and *cups*. You still can apply stop word removal and lemmatization to compare the outcome.

## Loading the data – no text preprocessing

I will load the same AG news data that we have been using in this book:

```
import pandas as pd
import numpy as np
pd.set_option('display.max_colwidth', -1)
path = "/content/gdrive/My Drive/data/gensim"
train = pd.read_csv(path + "/ag_news_train.csv")
```

## Modeling

You may have some prior knowledge of the ideal number of topics, such as 20 topics. You can specify the number of topics as `nr_topics=20`. If you do not specify the number of topics, you may end up with too many. However, many topics are similar, and their similarity score is high. BERTopic can combine two close topics that exceed a minimum similarity of 0.9. You can use `nr_topics='auto'` to let BERTopic combine similar topics. Another hyperparameter that affects the number of topics is the minimum size of a topic, `min_topic_size`. The default value is 10. If you increase the minimum size, the number of topics will decrease. In the example, I have set it to 100. Note that this hyperparameter is used by HDBSCAN. If you are not using HDBSCAN, this hyperparameter will not be used:

```
from bertopic import BERTopic
docs = train['Description']
model = BERTopic(nr_topics="auto",
                  min_topic_size=100)
```

It is straightforward to model with BERTopic by using `.fit()` or `.fit_transform()`. The difference is that `.fit()` trains the model, and `.fit_transform()` trains the model and predicts documents in the training data. I am going to use `.fit_transform()`, so it will also assign the prediction. Another important thing to know is that `.fit_transform()` is also used to predict new documents. Later, we will use it to score new documents:

```
topics, probs = model.fit_transform(docs)
```

The modeling process will take a while. You will see output such as the following. Feel free to take a coffee break.

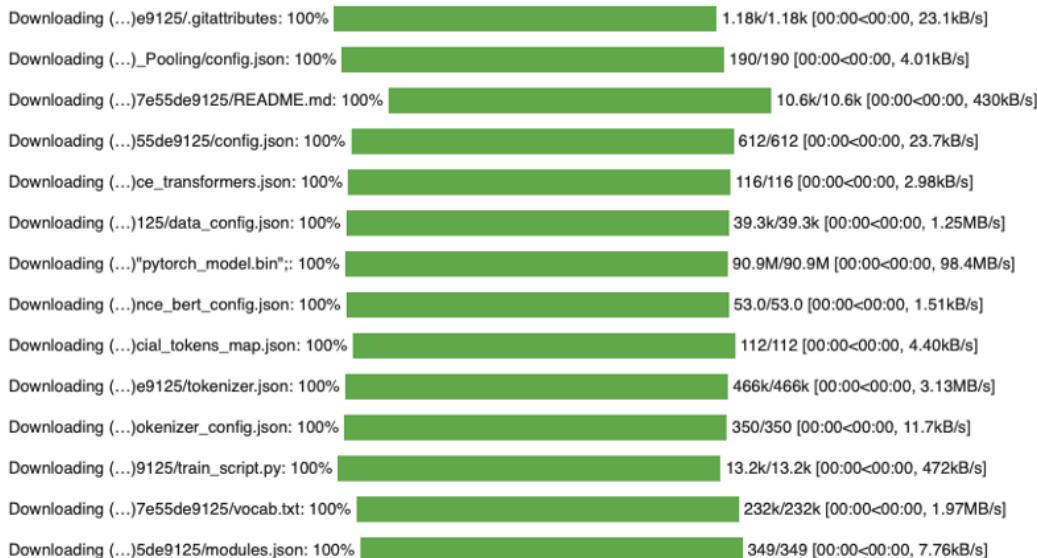


Figure 14.6 – An overview of BERTopic modeling process

Once the model training is complete, remember to save the model for future use:

```
model.save(path + "ag_news_bertopic")
```

Topic modeling usually produces very rich outcomes, as we have seen in LDA. BERTopic has developed several nice functions to query outcomes. Let's learn about these functions to review the results.

## Reviewing the results of BERTopic

Now, we are ready to inspect the outcome. We want to inspect the outcome of a topic model by inspecting the topic keywords and the count distribution across topics and show the representative documents for a topic. The BERTopic module has several convenient functions. They are as follows:

- `.get_topic_info()`: Get all topic information
- `.get_topic_freq()`: Get topic frequency
- `.get_topic(topic=12)`: Access a single topic, such as Topic 12
- `.get_topics()`: Access all topics
- `.get_document_info(docs)`: Get all document information
- `.get_representative_docs()`: Get representative docs per topic

## Getting the topic information

First, I am going to inspect the topic information by using `.get_topic_info()`:

```
topic_model.get_topic_info()[0:20]
```

The top 20 topics are shown in *Table 15.1*:

Topic	Count	Name	%
-1	38,208	-1_to_of_the_and	32%
0	9,720	0_coach_game_points_season	8%
1	8,517	1_baghdad_iraq_iraqi_killed	7%
2	6,019	2_oil_prices_stocks_crude	5%
3	5,851	3_drug_scientists_of_researchers	5%
4	4,394	4_sox_red_yankees_series	4%
5	2,568	5_athens_olympic_gold_medal	2%
6	2,193	6_space_nasa_spacecraft_moon	2%
7	1,985	7_kerry_bush_john_president	2%
8	1,832	8_palestinian_gaza_israeli_arafat	2%
9	1,605	9_insurance_securities_fund_fannie	1%
10	1,485	10_profit_percent_thirdquarter_earnings	1%
11	1,447	11_test_cricket_australia_india	1%
12	1,337	12_apple_music_ipod_itunes	1%
13	1,273	13_manchester_arsenal_united_manager	1%
14	1,130	14_yukos_russian_russia_moscow	1%
15	1,082	15_music_industry_copyright_antitrust	1%
16	1,036	16_open_federer_tennis_roddick	1%
17	945	17_iran_nuclear_tehran_uranium	1%
18	934	18_wireless_mobile_phone_nokia	1%
	...		
TOTAL	120,000		100%

Table 15.1 – The distribution of the topics

The record Topic -1 always refers to the “No group” topic, meaning those records cannot be assigned to any specific topic. Those documents are usually noisy and non-informative. *Table 15.1* shows that the size of Topic -1 can be as large as one-third of the population. This is quite typical in many BERTopic model outcomes.

The `.get_topic_freq()` function shows the document count distribution across topics:

```
model.get_topic_freq()
```

It is the same count frequency as in *Table 15.1*, so I am not going to print out the result.

## Inspecting the keywords of a single topic

Second, let’s inspect the keywords in a topic by using `.get_topic(topic)`. It shows the cosine similarity of the words in descending order. I want to see Topic 0, which is `0_coach_game_points_season` in *Table 15.1*:

```
model.get_topic(0)
```

The output is as follows:

```
[('coach', 0.011971737554558758),  
 ('game', 0.011782394472656704),  
 ('points', 0.011206724610129247),  
 ('season', 0.011075511694875929),  
 ('night', 0.00978769152164243),  
 ('the', 0.009601025462566184),  
 ('yards', 0.009049469490041952),  
 ('his', 0.009014691512566902),  
 ('quarterback', 0.009006802661929762),  
 ('no', 0.008906061716408082)]
```

These words characterize Topic 0. Apparently, this topic is about sports news.

## Getting document information

Third, we want to inspect each document and the topic that it belongs to. We will use `.get_document_info(docs)`. It will produce the documents, the topic that a document belongs to, the name of the topic, the top words in the topic, and the probability that a document belongs to the topic:

```
model.get_document_info(docs)
```

The results are shown in the following table:

Document	Topic	Name	Top_n_words	Probability
Reuters – Short-sellers, Wall Street's dwindling\band of ultra-cynics, are seeing green again.	2	2_oil_prices_stocks_crude	oil - prices - stocks - crude - search - us - google - dollar - barrel - reuters	0.8429
Reuters – Private investment firm Carlyle Group,\which has a reputation for making well-timed and occasionally\ controversial plays in the defense industry, has quietly placed\its bets on another part of the market.	-1	-1_to_of_the_and	to - of - the - and - in - its - for - on - that - said	0
Reuters – Soaring crude prices plus worries\about the economy and the outlook for earnings are expected to\hang over the stock market next week during the depth of the\summer doldrums.	2	2_oil_prices_stocks_crude	oil - prices - stocks - crude - search - us - google - dollar - barrel - reuters	0.7031
Reuters – Authorities have halted oil export\flows from the main pipeline in southern Iraq after\intelligence showed a rebel militia could strike\ infrastructure, an oil official said on Saturday.	-1	-1_to_of_the_and	to - of - the - and - in - its - for - on - that - said	0
AFP – Tearaway world oil prices, toppling records, and straining wallets, present a new economic menace barely three months before the US presidential elections.	2	2_oil_prices_stocks_crude	oil - prices - stocks - crude - search - us - google - dollar - barrel - reuters	0.7116

Table 15.2 – Table showing the topic probability and top words for each document

Remember, LDA and BERTopic come from two different approaches. In LDA, a document is assumed to contain a mixture of topics; in BERTopic, a document is clustered into a topic with a probability of belonging to that topic.

## Getting representative documents

Finally, we want to read the documents of a topic to inspect the model quality. In *Table 15.2*, we know Topic 2 is about the topic of economic news. Let's print out representative documents using `get_representative_docs()`:

```
model.get_representative_docs(2)
```

The documents are as follows:

```
['NEW YORK (Reuters) - U.S. stocks opened higher on Tuesday, as oil  
prices eased and insurance stocks extended gains from their rally late  
on Monday.',  
'NEW YORK (Reuters) - Stocks fell on Tuesday as shares of major oil  
companies tumbled after the price of crude fell more than a \\$1 a  
barrel to a 3-month low.',  
'AP - Tokyo stocks opened lower Tuesday on concerns about higher oil  
prices. The dollar was down against the Japanese yen.]
```

All these documents refer to economic news. They help to verify the quality of the model.

In *Chapter 12, LDA Visualization*, we learned the advantages of visualizing the rich information of a topic model. Let's learn how to visualize the results of the BERTopic model.

## Visualizing the BERTopic model

I would like to call out a few functions that help to produce nice visual presentations:

- `.visualize_topics()`: Visualize topics
- `.visualize_hierarchy()`: Visualize the hierarchy of topics
- `.visualize_barchart()`: Visualize the words of a topic
- `.visualize_heatmap()`: Visualize the similarity of topics

Let's review each in detail.

## Visualizing topics

BERTopic can visualize topics in a way very similar to pyLDAvis and LDAvis, as we have learned. To visualize the topics, run the following:

```
model.visualize_topics()
```

The Distance Map in *Figure 14.7* lets you use the slider at the bottom of the interactive map to select the topic. When you hover over a topic, it shows the size of the topic and its corresponding words:

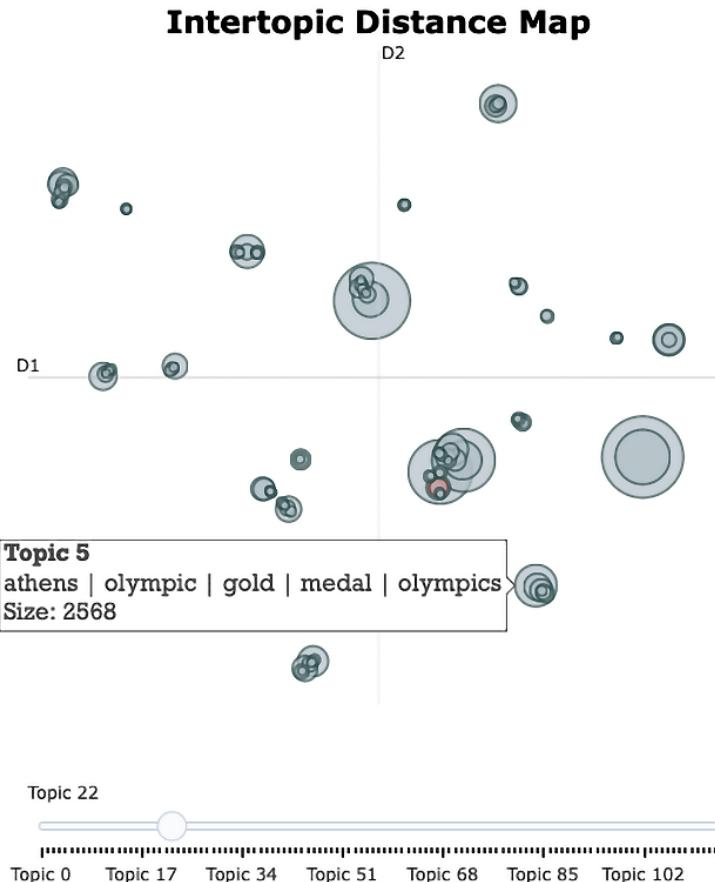


Figure 14.7 – Intertopic distance map

## Visualizing the hierarchy of topics

Because BERTopic uses HDBSCAN, it lets you visualize the hierarchical structure of the topics. To visualize this hierarchy, run the following:

```
model.visualize_hierarchy()
```

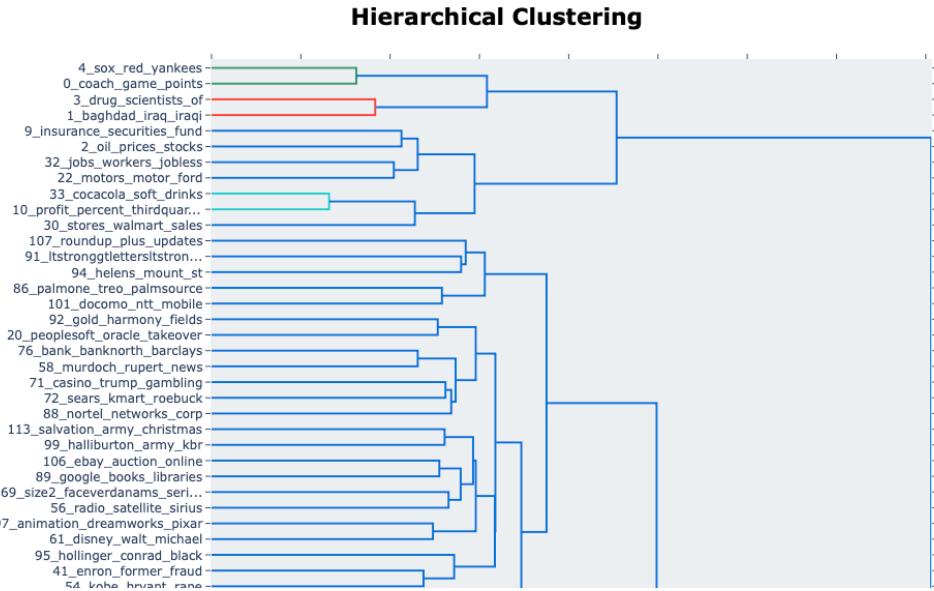


Figure 14.8 – An overview showing the hierarchy of topics through HDBSCAN

The hierarchy graph shows the similarity of topics.

## Visualizing the top words of topics

BERTopic lets you visualize the top words of several topics that can fit on one screen. It shows the c-TF-IDF scores for each topic:

```
model.visualize_barchart()
```

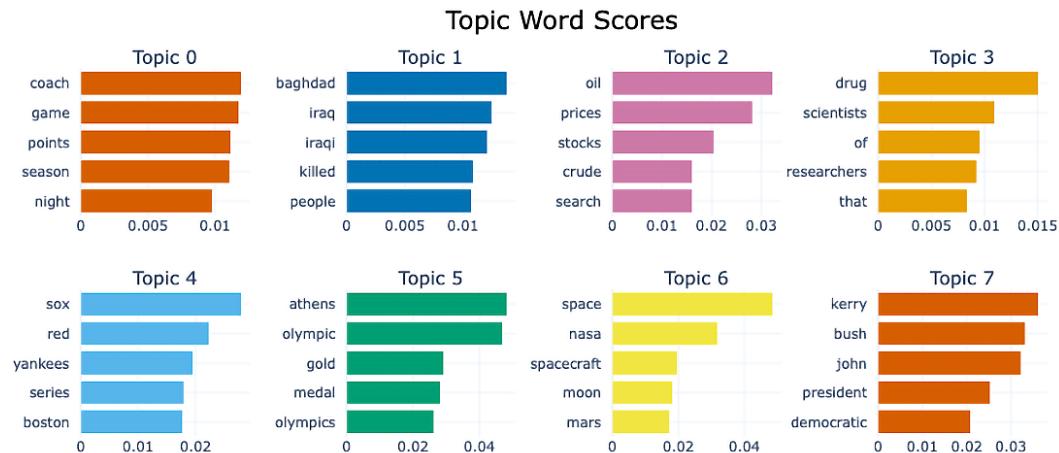


Figure 14.9 – Visualizing the count of the top words for the topics

## Visualizing on a heatmap

Another way to visualize the similarity of topics is by using a heatmap:

```
model.visualize_heatmap()
```

I am not going to print out the heatmap result because it is a large image. You can follow the Python code to see the results.

## Predicting new documents

To classify a new document into one of the topics, we just need to use `.transform()`. As discussed earlier, the difference between `.fit_transform()` and `.transform()` is that `.fit_transform()` trains and predicts documents, whereas `.transform()` predicts documents. Let's load the test dataset:

```
test = pd.read_csv(path + "/ag_news_test.csv")
test_docs = test['Description']
```

We will load the model that has been trained by using the `.load()` function:

```
from bertopic import BERTopic
model = BERTopic.load(path + "/ag_news_bertopic")
```

It is just that simple:

```
predicted_topics, predicted_probs =  
    model.transform(test_docs)  
print(predicted_topics)
```

The output is as follows:

```
[1, 6, 3, -1, 3, 15, -1, -1, 100, 100, 68, 12, 12, 3, 3, 3, 6, 6, 3,  
10, 25, -1,...]
```

BERTopic adopts a modular design approach that consists of five modules. You can choose techniques other than the default techniques. Let's see how to do that.

## Using the modular property of BERTopic

Because BERTopic is modular, you can employ alternative techniques for a model. This section will show you the code structure and then explain what other techniques you can implement. First, you will import the following classes:

```
from umap import UMAP  
from hdbscan import HDBSCAN  
from sentence_transformers import SentenceTransformer  
from sklearn.feature_extraction.text import CountVectorizer  
from bertopic import BERTopic  
from bertopic.representation import KeyBERTInspired  
from bertopic.vectorizers import ClassTfidfTransformer
```

Your BERTopic model, `topic_model`, is a collection of modules as follows. This view is clear because it tells us the components of BERTopic:

```
topic_model = BERTopic(  
    # word embedding  
    embedding_model=embedding_model,  
    # dimension reduction  
    umap_model=umap_model,  
    # clustering  
    hdbscan_model=hdbscan_model,  
    # tokenize topics  
    vectorizer_model=vectorizer_model,  
    # create topic representations  
    ctfidf_model=ctfidf_model,  
)
```

Then, we will define each module (`embedding_model`, `umap_model`, `hdbscan_model`, `vectorizer_model`, and `ctfidf_model`) as shown in the next sections.

## Word embeddings

The `embedding_model` module is a sentence transformer. Here, we use `all-MiniLM-L6-v2`:

```
embedding_model = SentenceTransformer("all-MiniLM-L6-v2")
```

The default embedding methods are `paraphrase-MiniLM-L6-v2` and `paraphrase-multilingual-MiniLM-L12-v2`. Besides these two methods, you can choose any model in the following list:

- `all-mpnet-base-v2`
- `multi-qa-mpnet-base-dot-v1`
- `all-distilroberta-v1`
- `all-MiniLM-L12-v2`
- `multi-qa-distilbert-cos-v1`
- `all-MiniLM-L6-v2`
- `multi-qa-MiniLM-L6-cos-v1`
- `paraphrase-multilingual-mpnet-base-v2`
- `paraphrase-albert-small-v2`
- `paraphrase-multilingual-MiniLM-L12-v2`
- `paraphrase-MiniLM-L3-v2`
- `distiluse-base-multilingual-cased-v1`
- `distiluse-base-multilingual-cased-v2`

## Dimensionality reduction

The default technique is UMAP:

```
umap_model = UMAP(n_neighbors=15, n_components=5,  
min_dist=0.0, metric='cosine')
```

There are many other techniques for dimensionality reduction. For instance, you can use **Principal Component Analysis (PCA)**:

```
from bertopic import BERTopic
from sklearn.decomposition import PCA

dim_model = PCA(n_components=5)
```

Then, your BERTopic model becomes the following. Notice I have highlighted `dim_model` in bold:

```
topic_model = BERTopic(
    # word embedding
    embedding_model=embedding_model,
    # dimension reduction
    umap_model= dim_model,
    # clustering
    hdbscan_model=hdbscan_model,
    # tokenize topics
    vectorizer_model=vectorizer_model,
    # create topic representations
    ctfidf_model=ctfidf_model,
)
```

## Clustering

The default clustering technique is HDBSCAN:

```
hdbscan_model = HDSCAN(min_cluster_size=15,
    metric='euclidean', cluster_selection_method='eom',
    prediction_data=True)
```

A well-known clustering method that you can use is K-means:

```
from sklearn.cluster import KMeans
cluster_model = KMeans(n_clusters=50)
```

Then, your model becomes the following:

```
topic_model = BERTopic(
    # word embedding
    embedding_model=embedding_model,
    # dimension reduction
    umap_model= dim_model,
    # clustering
    hdbscan_model= cluster_model,
```

```
# tokenize topics
vectorizer_model=vectorizer_model,
    # create topic representations
    ctfidf_model=ctfidf_model,
)
```

I trust you like BERTopic modeling so far. You may be interested in listing any noticeable features between BERTopic and LDA. Let's find out.

## Comparing BERTopic with LDA

LDA is a classical probabilistic model for topic modeling, whereas BERTopic leverages transformer-based models to create more context-aware and semantically meaningful topic representations. They come from two different literatures, each with its own set of characteristics and applications. The choice between the two depends on the specific needs of your NLP task and the nature of your text data. Here are the key differences between LDA and BERTopic.

### Approach

LDA is a generative probabilistic model for topic modeling. It assumes that documents are mixtures of topics, and topics are mixtures of words. LDA aims to discover these underlying topics and the distribution of words within them.

BERTopic, on the other hand, uses transformer-based language models, such as BERT, to generate document embeddings. It then incorporates UMAP for dimensionality reduction, DBSCAN for initial clustering, c-TFIDF to highlight significant terms, and MMR for keyword selection. These components work together to produce coherent and interpretable topics from text data.

### Word embeddings

LDA uses BoW and TF-IDF representations of text. It does not consider the semantic meaning of words or the context in which they appear.

BERTopic relies on pre-trained transformer BERT word embeddings or other embedding methods listed in this chapter. The word embedding techniques can capture the semantic relationships between words and the context in which they appear.

### Text preprocessing

LDA typically requires text preprocessing such as stop word removal and lemmatization, as I have demonstrated in this chapter for the model outcomes with text preprocessing.

BERTopic, in general, does not require text preprocessing for stop word removal or lemmatization.

## Language understanding

LDA is primarily based on statistical patterns in the distribution of words across documents and topics. It does not inherently understand the semantics or meaning of words.

BERTopic, being based on transformer models, has a better understanding of the meaning of words and their relationships. It can capture nuances in language, including synonyms and context.

## Topic clarity

The topics defined by LDA may be less clear if texts are noisy or short. Besides, LDA does not capture semantic relationships between words.

BERTopic tends to perform better on text data including short and noisy text. It is more robust in capturing context and semantics.

## Determination of the number of topics

LDA typically requires manual tuning of hyperparameters, such as the number of topics, alpha, and beta, to obtain meaningful results. The choice of these hyperparameters can significantly affect the quality of topics.

BERTopic conveniently determines the number of topics by combining similar topics if the similarity of two topics exceeds a minimum similarity of 0.9 (`nr_topic='auto'`).

## Determination of word significance in a topic

LDA topics are represented as probability distributions over words, which can be easily interpreted. This makes it straightforward to understand and label topics.

BERTopic determines the significance of words within a topic and calculates the cosine similarity between each word's embedding and the centroid embedding of the cluster to which it belongs. Words with higher cosine similarity to the cluster centroid are considered more significant to the topic represented by that cluster.

## Summary

LDA, BERTopic, and their variants are prevailing techniques for topic modeling. Motivated by the fact that Transformer-based models can produce better word embeddings, BERTopic adopts BERT word embeddings to better capture the semantic relationships between words and documents. BERTopic applies a modular design approach that consists of five modules: BERT, UMAP, HDBSCAN, c-TFIDF, and MMR. We learned about the advantages of BERTopic, which adopts these components. We also learned how to build, interpret, and visualize a BERTopic model.

In the next chapter, we will survey several real-world NLP applications in healthcare, clinical texts, legal documents, finance, and social media. The chapter aims to inspire you to provide solutions to your NLP challenges. I will also introduce an application that compares the BERTopic and LDA modeling results. You will have a chance to understand the differences.

## Questions

1. Please describe how BERT enhances the Transformer model.
2. Please name the modules of BERTopic.
3. What is UMAP?
4. Please describe DBSCAN.
5. Please describe the advantages of HDBSCAN over DBSCAN.
6. Why does BERTopic use MMR?
7. Please list at least two noticeable differences between BERTopic and LDA.

## References

1. Vaswani, A., Shazeer, N.M., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., and Polosukhin, I. (2017). *Attention is All You Need*. *ArXiv*, *abs/1706.03762*.
2. Kuo, Chris, *Modern Time Series Anomaly Detection with Python Examples*, <https://adamsjkuo.com/dSqxs1i>.
3. Devlin, J., Chang, M., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *ArXiv*, *abs/1810.04805*.
4. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T.J., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). *Language Models are Few-Shot Learners*. *ArXiv*, *abs/2005.14165*.
5. Grootendorst, M. (2022). BERTopic: Neural topic modeling with a class-based TF-IDF procedure. *arXiv preprint arXiv:2203.05794*.
6. Reimers, N. and Gurevych, I. (11 2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing. Retrieved from <http://arxiv.org/abs/1908.10084>.
7. McInnes, L. and Healy, J., *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction*, ArXiv e-prints 1802.03426, 2018

# 15

## Real-World Use Cases

Innovation comes by referencing best practices and applications. While you have learned many NLP techniques throughout this book, it is important to brainstorm to find new applications for your services. How do you find ideas for new applications? In my lectures, I always emphasize the importance of literature review. A good literature review can deepen your technical knowledge for its strength and weakness. It reduces the learning curve when you cross into another industry and learn about a new domain. It simply helps you to gain the confidence to go into an uncharted area while you plan to apply new techniques.

There is a wide range of NLP applications in healthcare, clinical texts, legal documents, finance, and social media. Maybe you are looking for ideas to test your NLP skills or see other Gensim use cases for inspiration. This chapter presents a range of large-scale NLP projects in different industries. You will be inspired by their research questions and their solutions. For each use case, I will brief to you on the following:

- Its background
- The questions
- The NLP solution
- Some ideas for takeaways

The techniques include Word2Vec, LSA, LDA, and BERTopic. They cover industries including medical, healthcare, social media, legal, and finance. We will survey seven use cases:

- Word2Vec for medical fraud detection
- Word2Vec for medical text annotation
- **Latent Semantic Analysis (LSA)** for Twitter/X health data
- Comparing LDA/NMF/BERTopic on Twitter posts
- Interpretable text classification from electronic health records
- BERTopic for legal documents
- Word2Vec for 10-K financial documents to the SEC

By the end of this chapter, you will be able to apply the techniques in these use cases to your projects. You will also gain confidence to manage large NLP projects. Let's start with the first use case.

## Word2Vec for medical fraud detection

While most medical professionals adhere to strict standards in their capacity, medical fraud exists especially in today's digital world. Data scientists and analysts have developed data science models and analysis techniques to detect medical fraud. All types of medical services have been codified together with text descriptions. These medical service codes have become important for data analytic purposes. However, the vast amount of medical codes, many of them being related, can be challenging for researchers. In this example, you will see that Word2Vec can be used to find the semantic relationships between codes. It helps you to group medical codes or to create features for your machine learning models. After reading this use case, you may be inspired by their approach to tackling a large coding system so you can use a similar approach in the medical field, or any other field.

### Background

Medical abuse happens when any practice is inconsistent with providing patients with medically necessary services, doesn't meet professionally recognized standards, and isn't priced fairly. A small percentage of medical resource abuse and waste can add up to trillions of dollars. Medical abuse includes deliberate fraud, unnecessary use and waste, or simply the abuse of medical resources. As stated in [1], in 2019, the total US healthcare spending was 18% of GDP, or \$3.3 trillion, or \$10,348 per person. A 10% fraud rate equals  $\$3.3\text{ trillion} * 10\% = \$330\text{ billion}$  per year.

All the medical procedures and services have been codified so you can find a corresponding code for a specific type of medical treatment. Medical doctors use a five-digit coding system called **Current Procedural Terminology (CPT)** to bill their professional services. For example, 99213 is an office visit for 15 minutes, and 99214 is the code for 25 minutes. The **Centers for Medicare & Medicaid Services (CMS)** has a coding system called the **Healthcare Common Procedure Coding System (HCPCS)** [2].

There are more than 50,000 codes in this system. They can be grouped together at a higher level. Although the codes are sequential, they may not follow the sequence strictly. Let me show you a few examples:

- 70260: X-ray exam of the skull
- 70300: X-ray exam of teeth
- 70488: CT maxillofacial without and with dye
- 70551: MRI of the brain without dye
- 72158: MRI of the lumbar spine without and with dye
- 72197: MRI of the pelvis without and with dye
- 72200: X-ray exam of sacroiliac joints

- 73020: X-ray exam of the shoulder
- 73702: CT of the lower extremity without and with dye
- 73706: CT of the angio lower extremity without and with dye
- 73718: MRI of the lower extremity without dye

If you quickly scan through the code descriptions, you may find that some of them are similar and can be grouped together into X-ray, **CAT scan (CT)**, and **Magnetic Resonance Imaging (MRI)**. If we group them naively by the first two or three digits, we will mix X-ray with MRI or CT. Beyond these few examples, there are more examples that are medically related but do not have the same first two or three digits.

## Questions

Since the codes come with text descriptions, can we apply NLP techniques to group them according to their medical relationships? Effective grouping can assist your analytic work and can be used by data scientists to create predicting features.

## NLP solution

As pointed out in [3], it is common to apply supervised learning modeling in the healthcare industry to predict suspicious activities. Insightful features are needed to produce an effective machine learning model. However, in this case, there are more than 50,000 HCPCS codes. If the codes are encoded through one-hot encoding, there will be 50,000 feature sand the data becomes very high-dimensional and the model may become ineffective.

Johnson and Khoshgoftaar [4] apply Word2Vec to the codes to create embeddings with lower dimensions. The Word2Vec embeddings capture the semantic relationships between codes. They trained the Word2Vec model with **continuous-bag-of-words (CBOW)** and **skip-gram (SG)** while tuning different window sizes for their models. The resulting word embeddings are used as features in their **Extreme Gradient Boosting (XGB)** model. They concluded that their embeddings have outperformed one-hot encodings significantly with 95% confidence.

## Takeaways

Many industries have their unique coding systems. The coding systems enable systematic analysis and are used for machine learning models. Using one-hot encoding for feature creation can create two more problems. First, it makes a sparse, high-dimensional dataframe and makes the modeling ineffective. Second, it loses the semantic relationships between codes.

## Word2Vec for medical text annotation

It is agreed by medical professionals and data professionals that **electronic health records (EHRs)** can help unlock data insights for many effective uses. EHRs enable systematical information retrieval, and medical documents, if annotated, can be used in supervised learning models or related modeling projects.

However, a well-known problem in clinical NLP research is the lack of large, annotated datasets. Annotations are usually performed by clinicians who read through a clinical report and highlight the portion of text that is relevant. The cost of annotating a large corpus by clinicians is high. The size of annotated reports is small compared to the large body of medical texts. These obstacles prevent further clinical NLP research.

### Background

The texts in a clinical report could be "... the patient has not taken medication for a few months and he has been homeless", and "... He is currently unemployed. He previously worked in the labor union...". The texts are usually unannotated and the cost to hire a clinician to annotate all reports is prohibitively expensive. If a physician or a researcher needs to pull clinical reports about "homeless", a simple keyword search for "homeless" will return many false positives and is not recommended.

### Questions

How can we apply NLP techniques to annotate a large number of documents automatically? Aguilar and David [5] try to solve this fundamental challenge. If the documents can be automatically annotated, it will benefit many applications in the healthcare industry.

### NLP solution

How did they solve this problem? They thought of Word2Vec modeling. With the collection of 20 million clinical notes and 1.7 million biomedical journal articles, they built a Word2Vec model that generated embeddings for 108,477 medical concepts. The paper shows the nearest words for three words "homeless," "divorce," and "bankruptcy." Let me list them:

- **Homeless:** "shelter," "shelters," "estranged," "sophomore," "widowed," "banned," "divorced," "roommates," "freshman," and "alcohol"
- **Divorce:** "bankruptcy," "eviction," "unemployment," "probation," "murder," "evicted," "marriage," "teenagers," "teachers," and "economy"
- **Bankruptcy:** "eviction," "evicted," "freelance," "budget," "employment," "profession," "victims," "airline," and "parole"

If a medical report states "he is estranged from his family and his mother," it implies family issues. Also, because he is estranged from his family, he is very likely away geographically from his family. So, the latent concepts "mother," "family," and "geographic" are all appropriate annotations to this report.

When a physician or a researcher searches for reports on family issues or estranged relationships, this and related reports will all be pulled for a comprehensive study. The search results will be more accurate than a simple keyword search. Further, Aguilar and David [5] developed the embeddings for medical text, called **cui2vec**, which is a downloadable set of pre-trained embeddings for other applications in the medical field.

## Takeaways

Annotating texts in a large corpus in any field is usually time-consuming and subjective. Annotation through simple keyword search can be inaccurate and creates a lot of false positives. Word2Vec can help to annotate unstructured text automatically.

Words in professional fields typically have domain-specific meanings. It will be very helpful to create word embeddings for many NLP use cases in a professional field.

To create a large set of word embeddings in a professional field for general usage, the Word2Vec model should be built on a large corpus for comprehensive coverage. The outcome is a pre-trained word embedding file that can benefit more use cases.

## Latent Semantic Analysis (LSA) for Twitter/X health data

We saw the value of annotating a large volume of text medical reports in the previous use case. Now, we want to annotate the large number of social media posts. Since most of the posts are textual and unstructured data, the annotation for them will provide great analytical insights. In this use case, we will learn how LSA can be applied to model social media data.

## Background

Social media is a huge mine of unstructured data. We need to understand people's interests in their posts and find out the dominant topics. Such deep analytics can help answer questions such as "What do people do to maintain their health?", "How many people follow diets?", "How many people do yoga?", and "Do yogi follow vegetarian/vegan diets?".

## Questions

How can we apply NLP techniques to annotate a large number of social media posts? A post may relate to two or more concepts, and a simple classification that assigns a post to one and only one class will not work. What are the NLP techniques?

## NLP solution

In the paper *Yoga-Veganism: Correlation Mining of Twitter Health Data* [6], the author used the parallel and distributed technology Apache Kafka to handle large streaming data from Twitter. He performed

Gensim data preprocessing to generate a useful topic model. Then he applied Gensim LSA and LDA to infer the topics. He used Gensim's coherence score to calculate topic coherence for LSA and LDA. He then used pyLDAvis to visualize the outcome.

## Takeaways

All the techniques in this paper have been included in this book. By following the code examples in this book, you shall get a decent outcome for your project of interest. As he pointed out, the quality of a topic model depends on the preprocessing step. The author was using Gensim's preprocessing function and was able to get a good outcome.

## Comparing LDA/NMF/BERTopic on Twitter/X posts

The previous use case showed us the value of annotating social media posts. Since we have learned many other NLP techniques, can we apply these other techniques? In this use case, we will learn how they applied more techniques to the social media data.

## Background

The richness of unstructured social media data has opened a new avenue for social science research. Topic modeling techniques have been applied to classify data and gain insights into it.

## Questions

Similar to the previous use case on social media text, how can we apply NLP techniques to annotate a large number of social media posts?

## NLP solution

The authors of [7] compared different types of topic modeling algorithms and documented their empirical findings. They collected 31,800 unique Twitter posts relating to travel and the COVID-19 pandemic. They applied Python LDA, NMF, and BERTopic modeling. They showed the standard results of these models, such as the topics visualization of LDA or c-TFIDF and hierarchical DBSCAN topics of BERTTopics. They reported the advantages and disadvantages in their comparison of the topics.

## Takeaways

The report applied the modeling procedures that have been covered thoroughly by the code notebook for this book. For our future modeling practice, it is helpful to understand what they considered the disadvantages of the LDA, NMF, and BERTopic techniques.

First, let's see what they considered disadvantageous for LDA. They are as follows:

- LDA requires detailed assumptions
- LDA requires careful tuning for hyperparameters
- The results of LDA can easily produce overlapping topics because topics are soft clusters
- The number of topics should be defined externally by users
- There is no objective evaluation metric for the results of LDA
- Because the results are not deterministic, the validity and reliability can be challenging

Second, let's see their views on NMF. The disadvantages are as follows:

- NMF often produces incoherent topics.
- The number of topics should be defined externally by users. This is similar to that of LDA.

Third, let's see their idea of the disadvantages of BERTopic:

- BERTopic's embedding approach can generate too many topics. The manual inspection for each topic can be labor-intensive.
- BERTopic can generate many outliers.
- BERTopic assigns each document to a single topic. There are no topic distributions for a single document, unlike in LDA.
- Objective evaluation metrics are missing.

We saw the value of EHRs in a previous use case. We also learned that objective evaluation metrics can be hard. In the next use case, we will learn how they overcome this challenge.

## **Interpretable text classification from electronic health records**

We can use many NLP techniques with EHRs to find their semantic relationships, as we saw in a previous use case. However, how do we know the resulting topics make sense? In this use case, we will see their proposal for objective evaluation metrics for the topic results.

### **Background**

As we said when discussing previous use cases, the clinical notes in EHRs have great possibilities for predictive tasks. Various topic modeling techniques can be applied to texts. Using topic models allows us to use topics as features. Data science researchers come to realize that the interpretability of these classification models is the key aspect.

## Questions

It is one thing to build many topic models, but selecting the most appropriate model for production use is not trivial. Is there an objective and systematic way to compare models? What are good evaluation metrics?

## NLP solution

The authors [8] believe that interpretability should be critical, in addition to the standard prediction accuracy (AUROC) in selecting an appropriate model. They believe the words within each topic (intra-topic) must be semantically related, and the words between topics (inter-topics) should be as distinctive as possible:

- **Intra-topic:** This is quantified by the coherence score.
- **Inter-topic:** This is quantified by the diversity score, which is the proportion of unique words in a topic model to the total number of words. If there are more unique words in a topic, the topic is distinct.

The authors use these metrics, in addition to the predictability metric AUROC, to evaluate 17 topic models, including LSI, LDA, **Fuzzy Latent Semantic Analysis (FLSA)**, and **Non-Negative Matrix Factorization (NMF)**. The modeling procedures in their report are very similar to those in our code notebooks.

## Takeaways

The authors made the point that the outcome of a topic model should be both interpretable and accurate. Their proposal for measuring interpretability in addition to predictability is very useful for practitioners.

## BERTopic for legal documents

NLP techniques can be applied to the large body of unstructured legal documents. One of the NLP applications is document grouping. Because legal documents can relate to each other in terms of topic of interest, topic modeling techniques can help legal professionals to search documents efficiently.

## Background

Search engines in legal databases are transforming to advanced NLP techniques. This technology transformation will save legal professionals' time and increase their productivity. Let's see what the challenges are.

## Questions

However, legal text processing is a challenging task because of legal-specific terminology. How to apply BERTopic modeling to tag the documents?

## NLP solution

The authors of [9] applied BERTopic modeling to legal documents from the US case law dataset. They performed the standard BERTopic modeling procedure with plausible results.

Interpretability is a critical criterion for the results of a topic model. How did they evaluate the quality of the topic model? They asked two legal professionals to conduct a qualitative assessment. The legal professionals inspected the most important words in the topics. Each expert documented whether there was a close relationship between the tagged words and the main topic of a document. They concluded that 84.6% of the topics are agreeable by human experts. Notice that manual inspection is an important way to verify the success of the results.

The authors produced sound topics for the case law data. For example, regarding “first amendment rights,” they identified topics such as “Amish,” “education,” “children,” “religious,” “school,” “life,” “state,” “child,” “parents,” and “compulsory.” These topics are indeed the very concepts that come up in the news about First Amendment rights in the United States.

## Takeaways

BERTopic modeling has been tested on legal documents to find the semantic relationships between documents. This project was executed procedurally and can be performed by using the code notebook in *Chapter 14, LDA and BERTopic*.

## Word2Vec for 10-K financial documents to the SEC

Financial documents have a large amount of unstructured data too. This use case has been chosen in this book because it applies NLP techniques to financial documents. Professionals in the financial services industry may find this use case helpful.

## Background

A 10-K financial document is an annual report filled in by a publicly traded company on its financial performance. It is required by the **US Securities and Exchange Commission (SEC)**. While a 10-K report has many numbers and tables, there are textual sections on Risk Factors (Item 1A), Management’s Discussion and Analysis (Item 7), and Quantitative and Qualitative Disclosures about Market Risks (Item 7A). These textual sections represent the perspective of management about the business of the company.

## Questions

How do we apply NLP techniques to financial documents?

## NLP solution

The author of [8] applied Word2Vec to get the word embeddings. The author then performed sentiment analysis by checking how the word embeddings relate to sentiment words ‘Positive’, ‘Negative’, ‘Litigious’, ‘Constraining’, ‘Uncertainty’, ‘WeakModal’, and ‘StrongModal’. The author produced the changes in sentiment year over year for Amazon. In 2013, Amazon had a new purchase and sale agreement, which involved some litigation perspectives. The author found there was a dramatic change in sentiment in 2013, which confirmed the facts around that time.

## Takeaways

Word2Vec can be applied to financial text corpora. The results of Word2Vec embeddings enable researchers and practitioners to retrieve information for management. The results of Word2Vec also enable additional data analysis. The author [8] performed sentiment analysis by using word embeddings and sentiment words.

## Summary

We reviewed a few selected real-world use cases in this chapter to demonstrate the breadth and depth of the techniques that we have learned about in this book. We trust this chapter has inspired you with new ideas, motivated you to invent new applications, and showed you how to apply the code examples that we have included in this book.

NLP keeps evolving at an unprecedented speed. ChatGPT, CPT-4, Llama 2.0, and so on were all developed in 2023. It is foreseeable that more and more generative AI models will emerge. With the knowledge in this book, you will be able to transition to generative NLP. This book helped you familiarize yourself with the fundamentals of NLP, including concepts such as tokenization, part-of-speech tagging, named entity recognition, syntactic parsing, LSA, LDA, and BERTopic. These techniques form the basis for your journey into generative NLP. Generative NLP heavily relies on neural networks. This book also presented the basics of neural network architectures such as Word2Vec or Doc2Vec. These architectures serve as building blocks for generative models.

Where do you go from here? LLMs are at the forefront of NLP. These models use large-scale transformer architectures to generate coherent and contextually relevant text. You are advised to learn about the self-attention mechanism and how it enables the model to capture long-range dependencies in text. You are also advised to read *Large Language Model Datasets* [11] to become familiar with the datasets that the LLMs are trained on. Fine-tuning an LLM to specific tasks is an important NLP development. You are advised to also read *Fine-tuning a GPT – Prefix-tuning* [12] and *Fine-tuning a GPT – LoRA* [13].

---

Remember that transitioning to generative NLP is a journey that requires patience and continuous learning. Starting with a solid foundation in traditional NLP will provide you with a strong basis for understanding and mastering the more advanced concepts of generative NLP.

## References

1. Kuo, Chris, “Feature Engineering for Healthcare Fraud Detection”, Medium.com (<https://medium.com/dataman-in-ai/2-features-for-healthcare-fraud-waste-and-abuse-7c262ac59859>)
2. CPT code list, CMS, <https://www.cms.gov/license/ama?file=/files/zip/list-codes-effective-january-1-2023-published-december-1-2022.zip>
3. Kuo, Chris, *Handbook of Anomaly Detection: With Python Outlier Detection: Build and modernize your anomaly detection models with examples*, 2023, <https://a.co/d/8VcNnhc>
4. J. M. Johnson and T. M. Khoshgoftaar, “Hcpes2Vec: Healthcare Procedure Embeddings for Medicare Fraud Prediction,” 2020 IEEE 6th International Conference on Collaboration and Internet Computing (CIC), Atlanta, GA, USA, 2020, pp. 145–152, doi: 10.1109/CIC50333.2020.00026.
5. Posada Aguilar and Jose David, “Semantics Enhanced Deep Learning Medical Text Classifier,” (2018) *Semantics Enhanced Deep Learning Medical Text Classifier*. Doctoral Dissertation, University of Pittsburgh.
6. Islam, T. (2019). “Yoga-Veganism: Correlation Mining of Twitter Health Data.” *ArXiv*, *abs/1906.07668*.
7. Egger R, Yu J. A Topic Modeling Comparison Between LDA, NMF, Top2Vec, and BERTopic to Demystify Twitter Posts. *Front Sociol*. 2022 May 6;7:886498. doi: 10.3389/fsoc.2022.886498. PMID: 35602001; PMCID: PMC9120935. <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9120935/>
8. Rijcken, E., Kaymak, U., Scheepers, F., Mosteiro, P., Zervanou, K., & Spruit, M. (2022). Topic Modeling for Interpretable Text Classification From EHRs. *Frontiers in Big Data*, 5.
9. Silveira, R., Fernandes, C.G., Neto, J.A., Furtado, V., & Filho, J.E. (2021). Topic Modelling of Legal Documents via LEGAL-BERT1.
10. Sehrawat, Saurabh, Learning Word Embeddings from 10-K Filings for Financial NLP Tasks (September 5, 2019). Available at SSRN: <https://ssrn.com/abstract=3480902> or <http://dx.doi.org/10.2139/ssrn.3480902>
11. Kuo, Chris, “Large Language Model Dataset,” Medium.com, <https://dataman-ai.medium.com/large-language-model-datasets-95df319a110>
12. Kuo, Chris, “Fine-tuning a GPT – Prefix-tuning,” Medium.com, <https://dataman-ai.medium.com/fine-tune-a-gpt-prefix-tuning-13c263e73141>
13. Kuo, Chris, “Fine-tuning a GPT – LoRA,” Medium.com, <https://dataman-ai.medium.com/fine-tune-a-gpt-lora-e9b72ad4ad3>



# Assessments

## Chapter 1 – Introduction to NLP

1. Natural language processing is a branch of computer science or **artificial intelligence (AI)** that uses computer algorithms to analyze, understand, and generate human language data. The algorithms process human language, either in the form of text or voice data, to “understand” its full meaning.
2. NLU focuses on understanding the meaning of human language. It extracts text or speech input and then analyzes the syntax, semantics, phonology, morphology, and pragmatics in the language.
3. While NLU is concerned with *reading* for a computer to comprehend, NLG is about *writing* for a computer to write. The term *generation* in NLG refers to an NLP model generating meaningful words or even articles.
4. The models offered by Gensim include bag-of-words (BoW), Term Frequency-Inverse Document Frequency (TF-IDF), latent semantic analysis/indexing (LSA/LSI), Word2Vec, Doc2Vec, Latent Dirichlet Allocation (LDA), and Ensemble LDA.
5. Gensim, spaCy, and NLTK.

## Chapter 2 – Text Representation

1. The advantage of Bag-of-*N*-words over BoW is its ability to capture meaningful phrases that exist in two or more words.
2. TF-IDF takes into account the importance of words within a document and across a collection of documents. It considers both the frequency of a term within a document (TF) and how unique or rare that term is across all documents (IDF). This results in TF-IDF giving higher weights to words that are more discriminative and informative, while downweighting common words.
3. Although BoW and TF-IDF may appear simple, they already have real-world applications. One important application is to prevent spam emails from going to the inbox folder of an email account. BoW or TF-IDF helps you to distinguish the characteristics of a spam email from regular emails.

## Chapter 3 – Text Wrangling and Preprocessing

1. Tokenization is the process of splitting a string into a list of tokens.
2. The technique to extract the root form of words is called stemming.
3. There is a slight difference between lemmatization and stemming. Lemmatization converts a word to a meaningful base word. The base word is still an actual word. Stemming converts a word to its root form and may not be a common formal word (such as ‘populated’ becoming ‘popul’).
4. spaCy does not automatically remove stop words but rather gives users full control of stop-word removal. It simply tags stop words for us to remove them.
5. PoS labels the correct meaning of a word in a sentence according to its context. It is a system where a word is assigned a syntactic function such as noun, pronoun, adjective, verb, and so on.
6. Gensim’s **preprocess\_string** class performs all text preprocessing tasks including stop-word removal, punctuation removal, and stemming.

## Chapter 4 – Latent Semantic Analysis with scikit-learn

1. An orthonormal matrix is a real square matrix whose columns and rows are orthogonal vectors.
2. A transformation matrix is a matrix that can transform one vector into another vector through matrix multiplication.
3. An important application of a transformation matrix is image transformation. Convex mirrors can distort an image to make it larger or smaller. A convex safety mirror or security mirror can show a wide angle of view.
4. SVD is a matrix decomposition method that can reduce a large, usually sparse, matrix into three sub-matrices.
5. In short, SVD can decompose an original sparse document-word matrix into three matrices. The first matrix is a document-topic matrix, the second matrix is a square topic-topic matrix, and the third matrix is a topic-word matrix.
6. The topic-word matrix relates latent topic to words.
7. The text variance that can be explained by the latent topics can be obtained by obtaining the explained variance ratio. By using `explained_variance_ratio_`, we can say that the topics can explain xx% of the text variance in documents.
8. To develop an LSI model with TruncatedSVD, you will first load the data, create TF-IDF, and then build the model with TruncatedSVD. Finally, you will provide the interpretations based on the model outcome.

## Chapter 5 – Cosine Similarity

1. A text keyword search relies on specific words or phrases within a document or database to retrieve relevant content. It matches the exact keywords provided by the user to those present in the text without considering the context or meaning of the words. In contrast, a semantic search goes beyond keyword matching and seeks to retrieve results that are conceptually related to the user's query, even if they don't precisely match the keywords.
2. Cosine similarity is a mathematical metric used to measure the similarity between two vectors in a multi-dimensional space. It calculates the cosine of the angle between the vectors, which represents their similarity; a smaller angle indicates greater similarity.
3. Cosine similarity is used for image comparison when images are represented as vectors. Image features can be extracted by image models such as CNNs to become high-dimensional vectors. Cosine similarity is then used to measure the similarity between these vectors. Images with similar content or visual patterns will have higher cosine similarity values, indicating greater resemblance.

## Chapter 6 – Latent Semantic Indexing with Gensim

1. The coherence score in LSI is a measure of how semantically related or coherent words within a topic are.
2. There are three common ways to compute the coherence score. They are UMass coherence, CV coherence, and CP coherence:
  - The Umass coherence compares the co-occurrence statistics of words in the corpus to the co-occurrence statistics within each topic. It uses PMI and considers the logarithm of the ratio of these co-occurrence probabilities. A low U\_mass score suggests more coherent topics.
  - The CV coherence measures the coherence of topics by calculating the pairwise similarity between the top words (usually nouns or adjectives) in each topic. A high CV score indicates more coherent topics.
  - The CP coherence calculates the co-occurrence probabilities between word pairs and compares these probabilities to those in a background corpus. High CP coherence scores indicate more coherent topics.

3. While establishing a search engine involves much engineering, the essential steps in setting up a model to be part of the search engine are as follows:
  - I. Load the saved objects.
  - II. Preprocess the new document.
  - III. Score the document to get the latent topic scores.
  - IV. Calculate the similarity scores with the new document.
  - V. Find documents with high similarity scores.

## Chapter 7 – Using Word2Vec

1. The distributional hypothesis says that words that occur in similar contexts tend to have similar meanings. For example, the words `cat` and `dog`, `temple` and `monk`, or `king` and `queen` are sometimes seen together. In contrast, the words `iron` and `monk`, or `car` and `sky` appear less often in the same contexts.
2. The first advantage of Word2Vec is that it can capture semantic relationships between words. Words with similar meanings or related concepts tend to have vectors that are close together. The second advantage is its ability to present contextual information. By considering the surrounding words, it captures the meaning of a word based on its usage and co-occurrence patterns. This allows Word2Vec to capture nuances in word meaning and understand different senses of a word based on its context. The third advantage is dimensionality reduction. Word2Vec presents the high-dimensional space of words in lower-dimensional word vectors. The fourth advantage is its ability to capture the compositional relationships between words. This compositionality property allows for analogical reasoning and capturing relationships between words. Finally, the fifth advantage is efficiency. Word2Vec training is relatively fast compared to more complex models, such as transformers.
3. Word2Vec has been applied to build recommendation systems for businesses such as e-commerce websites or movie streaming companies.
4. The two modeling techniques of Word2Vec are the Skip-Gram and the Continuous Bag-of-Words models.
5. The Skip-Gram model is a simple neural network. It has an input layer, a hidden layer, and an output layer. We are interested in the weights of the hidden layer, which become the word vectors. The CBOW model is almost the reverse of the SG model.
6. **t-SNE** is the abbreviation for **t-distributed Stochastic Neighbor Embedding**. It is a dimensionality reduction technique used for visualizing high-dimensional data in a low-dimensional space. It preserves the local structure of the data while revealing the underlying global patterns.

## Chapter 8 – Doc2Vec with Gensim

1. The abbreviation **PV-DBOW** stands for **Paragraph Vector with Distributed Bag-of Words** and **PV-DM** stands for **Paragraph Vectors — Distributed Memory**.
2. The neural network architecture of PV-DBOW is similar to that of Skip-Gram in Word2Vec. The architecture of PV-DM is similar to that of CBOW in Word2Vec.
3. Doc2Vec has many real-world applications. For example, when you search for a job on LinkedIn or Indeed.com, you see similar job postings presented next to your target job posting. This is done by Doc2Vec.
4. By giving a whole paragraph a single ID, Doc2Vec considers the paragraph ID as a word. When a word is treated, it can be embedded as a vector.

## Chapter 9 – Understanding Discrete Distributions

1. A Bernoulli distribution is a simple discrete distribution that describes the probability of binary outcomes in an experiment.
2. A binomial distribution has two outcomes in many independent trials. When there are many independent trials and each trial has two exact outcomes, a Bernoulli distribution becomes the binomial distribution.
3. A multinomial distribution describes  $n$  independent experiments, and each experiment has more than two choices. The multinomial distribution generalizes the binomial distribution from two outcomes to more than two outcomes.
4. A beta distribution is a distribution of probabilities describing the uncertainty about the probability of success or failure of an experiment. It is an alteration of the binomial distribution. While the binomial distribution describes the number of successes, the beta distribution describes the probability that successes happen.
5. A Dirichlet distribution is a distribution of probabilities for multiple outcomes. The beta distribution is a distribution of probabilities for binary outcomes. The Dirichlet distribution is an extension of the beta distribution to multiple outcomes.
6. A simplex represents the simplest possible shape in any dimension. A 0-dimensional simplex is a point, a 1-dimensional simplex is a line segment, a 2-dimensional simplex is a triangle, and a 3-dimensional simplex is a tetrahedron.
7. In Bayesian inference, the posterior probability is the prior probability times the likelihood of the observed data. If the prior and the posterior belong to the same parametric family, then the prior and the posterior are called conjugate distributions.

## Chapter 10 – Latent Dirichlet Allocation

1. LDA assumes that a document is the result of topics and that a topic is a distribution of words. By discovering the hidden topics and words, LDA can assign a document to topics with probabilities. LDA considers hidden topics as templates in a printing shop. Each topic template contains a set of words. An article is “generated” from a topic template or a mixture of topic templates.
2. The name *Latent Dirichlet Allocation* describes its technical approach. It contains the word *latent* because it finds the hidden topics in the latent space. The word *Dirichlet* (pronounced as *Deer-e-kh-let*) refers to the assumption that both the distribution of topics in a document and the distribution of words in a topic follow Dirichlet distributions. *Allocation* means that a mixture of topics and words is generated from the topic templates and allocated to a document.
3. The distribution of topics and the distribution of words in a topic are all hidden variables. Only the distribution of words in each document is observable. LDA simulates the assumed distribution of topics and the distribution of words of a topic to match the observable data. By optimizing the parameters of the assumed distributions to the observable data, LDA can derive the hidden distributions.
4. Descriptive modeling focuses on understanding and summarizing the relationships and patterns within the data. Its primary goal is to describe the existing data distribution or relationships between variables. Generative modeling is a method that generates new samples to resemble the original data to learn the underlying data patterns. It creates new data points based on the patterns it has learned from the training data.
5. E-M is an iterative optimization algorithm that’s used in statistics and machine learning. It’s particularly useful for estimating parameters in models with unobserved or latent variables. E-M alternates between two key steps: the E-step, where it computes the expected values of the latent variables given the current parameter estimates, and the M-step, where it updates the model parameters to maximize the expected likelihood of the observed data based on the estimates from the E-step. This iterative process continues until convergence, providing parameter estimates that maximize the likelihood of the observed data, even when some variables are not directly observed.
6. Variational E-M is an extension of the E-M algorithm that’s used in statistics and machine learning. It combines E-M with variational inference, a technique that approximates complex probability distributions with simpler ones. In variational E-M, during the E-step, the posterior distribution of latent variables is approximated using a predefined family of tractable distributions. Then, during the M-step, the model parameters are optimized to maximize a lower bound on the likelihood of the observed data based on the approximated posterior. This iterative process continues until convergence, providing parameter estimates that maximize the lower bound of the likelihood, making it useful for handling models with latent variables and complex likelihood functions.

7. Gibbs sampling is an MCMC technique that's used for generating samples from complex, high-dimensional probability distributions. It is particularly effective for situations where sampling or directly computing from these distributions is challenging. In Gibbs sampling, each step updates one variable at a time while keeping the others fixed, based on their conditional distribution given the current state. This iterative process continues until convergence, generating a sequence of samples that eventually approximate the desired distribution. Gibbs sampling is widely used in Bayesian inference, machine learning, and other fields to estimate posterior distributions and explore complex probability spaces.

## Chapter 11 – LDA Modeling

1. NLP modeling techniques that can yield a better outcome when using text preprocessing include LSA/LSI, LDA, and Ensemble LDA.
2. There are common words in a corpus even after text preprocessing. BoW data considers the significance of a word by its frequency. TF-IDF data can down-weight frequent words in a corpus, resulting in a possible improvement of LDA outcome.
3. In NLP, the metric to measure the closeness of a topic is called the coherence score. The coherence score is the average or median of word similarities of top words in a topic.
4. The prediction outcome of LDA for a document is a list of 2-tuples in the form of `(topic_id, topic_probability)`. It ignores topics where the probability is below the set minimum probability.

## Chapter 12 – LDA Visualization

1. The essential elements of an LDA outcome are as follows:
  - The distances between topics
  - The top words of a topic
  - The distribution of words in a topic
  - The distribution of words in a topic when compared with that of the whole corpora
2. The marginal topic distribution represents the count percentage of documents on that topic in the corpus. It is calculated as the topic proportions given the topic distribution and the document lengths.

## Chapter 13 – The Ensemble LDA for Model Stability

1. LDA can produce unstable topic outcomes because the model settles at a local maximum depending on the random initialization. This cannot be avoided even if a seed number is set to control for random initialization. The root cause of LDA's topic instability is that a single LDA model identifies the “true” topics and “pseudo” topics and produces noisy predictions. LDA can produce “true” topics as well as “pseudo” topics.
2. Ensemble LDA builds multiple models – an ensemble – to weed out the pseudo topics. By training an ensemble of topic models and throwing out topics that do not reoccur across the ensemble, the Ensemble LDA model can deliver stable outcomes.
3. DBSCAN stands for Density-Based Spatial Clustering of Applications with Noise. It defines an epsilon-neighborhood around each data point. If the number of data points within its epsilon-radius is more than a specified minimum number of points, the data point is called a core point and a cluster is formed. DBSCAN expands a cluster for all the points reachable from this core point until it includes all reachable points. This allows DBSCAN to capture clusters of varying shapes and sizes.
4. CBDBSCAN (short for Checkback DBSCAN) is an extension of DBSCAN that incorporates a “checkback” mechanism to improve the accuracy and efficiency of the clustering process. DBSCAN applies fixed values for the radius and the minimum number of neighbors even if the data has varying densities or noise levels. Instead of using a fixed radius, CBDBSCAN uses an adaptive radius that adjusts based on the local density of the data. This allows the algorithm to capture clusters of varying densities and shapes better than DBSCAN.
5. Ensemble LDA, just like LDA, requires text preprocessing. The two most popular text preprocessing methods are BOW and TF-IDF.

## Chapter 14 – LDA and BERTopic

1. BERT enhances the Transformer model by teaching the Transformer to learn from the words before and after each word so it knows the context and order better. This helps the Transformer understand tricky things such as jokes or words with multiple meanings, making it excellent at understanding all kinds of text, such as chatting or reading books. BERT removes the unidirectionality constraint in the Transformer and uses an MLM that randomly masks some of the input tokens. Since some tokens are masked, MLM has to predict the original vocabulary of the masked word based on its before and after context.
2. BERT consists of five modules: BERT, UMAP, HDBSCAN, c-TFIDF, and MMR.
3. UMAP stands for Uniform Manifold Approximation and Projection. It is a clever way to turn complex data into simpler pictures. Imagine you have a big puzzle with lots of pieces (data points), and you want to arrange them on a board so that similar pieces are close together and different pieces are far apart. UMAP does this by paying attention to how each puzzle piece relates to its neighbors. It keeps the important relationships while squishing the puzzle onto

the board, so you can see patterns and groups in the data much more clearly. It's like making a map of your puzzle where the distances show how things are connected in a way that makes sense to your brain.

4. DBSCAN stands for Density-Based Spatial Clustering of Applications with Noise and is a powerful clustering algorithm used to identify groups within a dataset, especially when the clusters have irregular shapes and sizes. It works by examining the density of data points in the vicinity of each point. Points that are close to each other and have enough neighbors within a specified radius are considered part of a cluster. This approach allows DBSCAN to automatically find clusters of varying densities and even detect outliers as noise points. It's particularly useful when traditional methods such as k-means struggle with complex data distributions and noise.
5. HDBSCAN offers an advantage over DBSCAN by providing a more flexible and informative way to identify clusters in data. While both algorithms are based on density, HDBSCAN constructs a hierarchy of clusters at different levels of granularity. This means it can uncover clusters within clusters, offering a richer understanding of the data's structure.
6. Maximal Marginal Relevance, otherwise known as MMR, is a technique used in text summarization to select the most relevant and diverse set of words. MMR is employed in BERTopic to select representative documents for each topic in a way that maximizes the relevance of the selected documents to the topic while minimizing redundancy with documents already assigned to other topics.
7. WE can describe several noticeable differences. First, BERTopic conveniently determines the number of topics by combining similar topics if the similarity of two topics exceeds a minimum similarity of 0 . 9. Second, BERTopic uses BERT-based word embedding while LDA applies BoW or TF-IDF. Third, LDA employs a generative algorithm but BERTopic does not.



# Index

## A

**adaptive radius** 217  
**artificial intelligence (AI)** 4

## B

**Bag-of-N-grams** 20  
coding 26  
using, with NLTK 29, 30  
using, with scikit-learn 28, 29  
with Gensim 26-28  
**Bag-of-Visual-Words (BoVW)** 21  
**Bag-of-Words (BoW)** 19, 61, 233  
coding 22  
real-world applications 21  
with Gensim 22-24  
with scikit-learn 24-26  
used, for performing word embedding 79-81  
**Bayes' theorem** 176, 178  
**Bernoulli distribution** 149  
binary outcomes 150  
defining 149  
example 149  
overview 149  
**BERT-based word embeddings** 231

**Bidirectional Encoder Representations from Transformers (BERT)** 190, 227, 230, 231  
**BERTopic** 227  
BERT 231  
building 233  
c-TFIDF 233  
data loading 234  
document information, obtaining 237, 239  
HDBSCAN 232  
keywords of single topic, inspecting 237  
Maximal Marginal Relevance (MMR) 233  
modeling 234, 235  
modular property 243, 244  
representative documents, obtaining 239  
results, reviewing 235  
structure 231  
topic information 236, 237  
UMAP 232  
used, for topic modeling 13  
visualizing 239  
working 231  
**BERTopic, for legal documents** 256  
background 256  
NLP solution 257  
questions 257  
takeaways 257

**BERTopic, versus LDA** 246

approach 246  
language understanding 247  
number of topics determination 247  
text preprocessing 246  
topic clarity 247  
word embeddings 246  
word significance determination,  
in topic 247

**beta distribution** 158

defining 159  
examples 159  
in Bayesian inference 165  
overview 160-164

**bidirectional** 231**bigram** 20**binomial distribution** 150

defining 151  
example 150  
overview 151, 152  
stats module, using 153-155

**border points** 216**BoW data**

models, building on 196

**BoW object**

loading 93

**C****CAT scan (CT)** 251**CBOW model**

loading 118  
saving 118  
using 118, 119

**Centers for Medicare & Medicaid  
Services (CMS)** 250**ChatGPT** 7**Checkback DBSCAN**

(CBDBSCAN) 217, 218

**clustering, BERTopic** 245, 246**coherence score** 197

using, to find optimal number  
of topics 85-87

**coherence score methods**

CP coherence 86

CV coherence 86

umass coherence 86

**conjugate distributions** 165**context window** 7**Continuous Bag-of-Words**

(CBOW) 109, 110, 251

**continuous random variable** 148**Convolutional Neural Network (CNN)** 74**core points** 215**cosine similarity** 71-73

computing, with scikit-learn 74, 75  
in images 73, 74

**count vectorizing** 18

c-TFIDF (class-based TF-IDF) 233  
topic representation, creating 233

**cui2vec** 253**Current Procedural Terminology (CPT)** 250**D****data visualization**

interactive graph 207-212  
with pyLDAvis 206, 207

**Density-Based Spatial Clustering  
of Applications with Noise**  
(DBSCAN) 215-217, 232

overview 216

**dictionary list**

loading 93

**dimensionality reduction, BERTopic** 103, 244

**Dirichlet distribution** 166

- defining 167
- example 166
- in Bayesian inference 173
- of topics 180
- overview 168-173
- simplex 167, 168

**discrete probability distribution**

- basics 148

**discrete random variable** 148

**discriminative modeling** 176

**distributional hypothesis** 102

**Doc2Vec** 127, 129

- real-world applications 134
- versus Word2Vec 123

**Doc2Vec model**

- tips, on building 142

**Doc2Vec modeling, with Gensim** 134

- modeling 136, 137
- model, saving 137
- text preprocessing, of Doc2Vec 135, 136
- training data, saving 137

**Doc2Vec model, putting into production** 138

- model, loading 138
- training data, loading 138
- use case, for finding relevant documents based on keywords 140, 141
- use case, for finding similar articles 138-140

**document**

- finding, with high similarity scores 96-98
- predicting 242, 243
- preprocessing 94, 95
- scoring, to obtain latent topic scores 95
- similarity scores, calculating with 96

**dot product** 73

**E**

**eigenvalues** 58, 59

**eigenvectors** 58, 59

**electronic health records (EHRs)** 127, 252

**encoding methods, word embedding**

- Bag-of-N-grams 20
- BoW 19
- one-hot encoding 18

**Ensemble LDA** 13

- CBDSCAN 215
- migrating, to Latent Dirichlet Allocation (LDA) 214
- output 215
- text processing 215
- topic assignment 215
- training 215

**Ensemble LDA model, building with Gensim** 218, 220-222

- dictionary, saving 219, 220
- new documents, scoring 222, 223
- text representation, creating with BOW and TF-IDF 219
- training data, preprocessing 218

**epsilon** 215

**evidence** 177

**evidence lower bound (ELBO)** 185

**Expectation-Maximization (E-M)** 175, 178

**Expectation step (E-step)** 178

**Extreme Gradient Boosting (XGB) model** 251

**F**

**FastText**

- versus Word2Vec 124

**Fuzzy Latent Semantic Analysis (FLSA)** 256

**G**

**garbage in, garbage out (GIGO)** 35  
**gated recurrent units (GRUs)** 228  
**Gaussian mixture models (GMMs)** 178  
**generative AI** 6  
**generative modeling** 176  
**Generative Pre-trained Transformer 4 (GPT-4)** 7  
**generative probabilistic modeling** 176  
**Gensim** 8, 9  
 Bag-of-N-grams, using with 26-28  
 BoW, using with 22-24  
 Ensemble LDA model, building with 218  
 NLP modeling techniques 9  
 for preprocessing 44  
 TD-IDF, using with 31-33  
 used, for coding 44  
 using, for stemming 45  
 using, for stop word removal 45  
**Geographic information systems (GIS)** 71  
**Gibbs sampling**  
 in LDA 185, 186  
 versus variational E-M 186  
**Global Vectors (GloVe)**  
 versus Word2Vec 124  
**Google Colab** 42  
**GPT-H2O.ai** 7

**H**

**HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise)** 232  
 cluster documents 232  
**Healthcare Common Procedure Coding System (HCPCS)** 250  
**HuggingFace.co** 8

**I**

**interactive graph** 207-212  
**interpretable text classification, from electronic health records** 255  
 background 255  
 NLP solution 256  
 questions 256  
 takeways 256  
**intertopic distance map** 207  
**inter-topics** 256  
**intra-topics** 256

**J**

**Jensen's inequality** 183

**K**

**Kullback-Leibler (KL)** 184

**L**

**large language models (LLMs)** 3, 227  
**Latent Dirichlet Allocation (LDA)** 4, 12, 147, 175  
 Dirichlet distribution, of topics 180  
 Ensemble LDA, migrating to 214  
 Gibbs sampling 185, 186  
 idea behind 179, 180  
 structure 180-183  
**Latent Dirichlet Analysis (LDA)** 53  
**Latent Semantic Analysis (LSA)** 53, 147  
**Latent Semantic Indexing (LSI)** 53  
 truncated SVD, using for 61  
 truncated SVD, using with real data 65

**LDA modeling**

- experimenting with 192
- model result, showing on BoW 193, 194
- model result, showing on TF-IDF 194, 195
- models, building on BoW data 192, 193
- models, building on TF-IDF data 197

**LDA models**

- building, with different number of topics 196

**LDA/NMF/BERTopic on Twitter/X posts, comparison 254**

- background 254
- NLP solution 254
- questions 254
- takeaways 254, 255

**lemmatization 14, 37**

- NLTK, using for 43
- spaCy, using 40

**likelihood 177****likelihood ratio 177****long short-term memory (LSTM) 228****lowercase conversion 36****LSA, for Twitter/X health data 253**

- background 253
- NLP solution 253
- questions 253
- takeaways 254

**LSI, modeling with Gensim 81**

- Bag-of-Words (BoW) 82, 83
- Term Frequency-Inverse Document Frequency (TF-IDF) 83-85

**M****Magnetic Resonance Imaging (MRI) 251****Markov chain Monte Carlo (MCMC) 186****masked language model (MLM) 230****matrix**

- determinant 55

**Matrix Market (MM) exchange**

- format 219, 221**

**matrix operations 54**

- orthogonal matrix 54

**Maximal Marginal Relevance (MMR) 233****Maximization step (M-step) 178****minPts 215****model**

- loading 93
- saving, for production 88-92
- using, to score new documents 200

**model, using as search engine 92**

- dictionary list, loading 93
- document, finding with high similarity scores 96-98
- document, preprocessing 94, 95
- document, scoring to obtain latent topic scores 95
- similarity scores, calculating with document 96

**model, using to score new documents**

- new texts, scoring 201
- outcome 201-203
- text preprocessing 200, 201

**modular property, BERTopic 243, 244**

- clustering 245, 246
- dimensionality reduction 244
- word embeddings 244

**morphemes 6****multidimensional scaling (MDS) 207****multinomial distribution 156**

- defining 156
- example 156
- overview 156-158

## N

- naive (standard) softmax approach** 108
- named entity recognition (NER)** 14
- natural language generation (NLG)** 6
  - ChatGPT 7
  - GPT-4 7
  - Gpt.h2o.ai 7
  - HuggingFace.co 8
- natural language processing (NLP)** 4
  - applications 4
  - NLG 6
  - NLU 5
- Natural Language Toolkit (NLTK)** 36
  - Bag-of-N-grams, using with 29, 30
  - using, for coding 42
  - using, for lemmatization 43
  - using, for stop word removal 43
  - using, for tokenization 42
- natural language understanding (NLU)** 5
  - morphology 6
  - phonology 6
  - pragmatics 6
  - semantics 5
  - syntax 5
- negative sampling** 108, 132
- NLP modeling techniques, Gensim**
  - BoW 9
  - Doc2Vec 12
  - Ensemble LDA 13
  - LDA 12
  - TF-IDF 9
- NLP operations, in spaCy**
  - lemmatization 14
  - named entity recognition (NER) 14
  - Part-of-speech (PoS) tagging 14
  - rule-based matching 14

tokenization 14

word vectors 14

## NLP preprocessing

- lowercase conversion 36
- punctuation removal 37
- stemming 37
- stop word removal 36
- tokenization 36

## NLP Python modules

NLTK 15

spaCy 14

## Non-Negative Matrix Factorization (NMF)

256

## O

### one-hot encoding

### optimal number of topics

determining 197-199

### orthogonal matrix

### out-of-vocabulary (OOV)

124

## P

### Paragraph Vector (PV)

### Paragraph Vectors - Distributed

Memory (PV-DM) 129, 132-134

### Paragraph Vector with Distributed Bag-of Words (PV-DBOW)

129, 130

hidden layer 131

input layer 131

model optimization 132

output layer 131

### part-of-speech (PoS)

### tagging

14

spaCy, using 41

### pointwise mutual information (PMI)

86

### PorterStemmer

44

**posterior probability** 165, 177

**pre-trained model**

using, for semantic search 110-113

**Principal Component Analysis**

(PCA) 120, 245

**prior probability** 165, 177

**Probability Density Function (PDF)** 148

**Probability Mass Function (PMF)** 148

**production**

model, saving for 88-92

**pyLDAvis**

used, for data visualization 206, 207

## R

**randomized\_SVD** 63, 64

**random variable** 148

**random variable samples (rvs)** 154

**real-world use cases**

BERTopic, for legal documents 256

interpretable text classification, from electronic health records 255

Latent Semantic Analysis (LSA), for Twitter/X health data 253

LDA/NMF/BERTopic on Twitter/X posts, comparision 254

Word2Vec, for 10-K financial documents to SEC 257

Word2Vec, for medical fraud detection 250

Word2Vec, for medical text annotation 252

**recurrent neural networks (RNNs)** 228

**reinforcement learning from human feedback (RLHF)** 7

**rule-based matching** 14

## S

**scikit-learn**

Bag-of-N-grams, using with 28, 29

BoW, using with 24-26

cosine similarity, computing with 74, 75

TD-IDF, using with 33

used, for coding truncatedSVD 62

**search efficiency** 71

**semantic relationships** 103

**semantic search** 53

**Singular Value Decomposition**

(SVD) 53, 59, 60

truncated SVD 61

**singular values** 60

**Skip-Gram (SG)** 105

data preparation 105, 106

hidden layer 108

input and output layers 107

model computation 108

stop words, removing 108

using 120

Word2Vec model, training 119

**spaCy** 14

coding with 38-40

for lemmatization 40

for PoS 41, 42

pipeline, building with 45-48

**stemming** 37

**stop word removal** 36

NLTK, using for 43

## T

**t-distributed Stochastic Neighbor Embedding (t-SNE)**

Word2Vec model, visualizing with 120-123

**technical language processing (TLP)** 36

**TensorBoard**

- Word2Vec, visualizing with 115, 116

**Term Frequency-Inverse Document Frequency (TF-IDF)** 3, 20, 21, 61

- coding 31
- real-world applications 21
- used, for performing word embedding 79-81
- with Gensim 31-33
- with scikit-learn 33

**text preprocessing** 190, 191

- performing 78, 79

**TF-IDF data**

- used, for building model 194

**TF-IDF object**

- loading 93

**token** 7

**tokenization** 14, 36

- NLTK, using for 42

**tokenized** 7

**topic modeling**

- with BERTopic 13

**transformation matrix** 55, 56

- examples 56, 57

**transformer** 7

**Transformer model** 228-230

**trigram** 20

**truncated SVD** 61

- for LSI 61
- using, for LSI with real data 65

**truncatedSVD**

- coding, with scikit-learn 62
- using 62, 63

**truncated SVD, using with real data for LSI**

- data, loading 65, 66
- outcome, interpreting 67-69
- TF-IDF, creating 66
- used, for building model 67

**U**

**UMAP (Uniform Manifold Approximation and Projection)** 232

- dimensionality of embeddings, reducing 232

**US Securities and Exchange Commission (SEC)** 257

**V**

**variational E-M** 185

- Gibbs sampling, in LDA 185, 186
- versus Gibbs sampling 186

**variational Expectation-Maximization** 185

**variational inference** 183-185

**vector database** 71

**visual presentations, BERTopic** 239

- heatmap, visualizing 242
- hierarchy of topics, visualizing 241
- topics, visualizing 240
- top words of topics, visualizing 241

**W**

**Word2Vec** 10, 102, 129

- advantages 103
- real-world applications, reviewing 104
- versus Doc2Vec 123
- versus FastText 124
- versus GloVe 124
- visualizing with TensorBoard 115, 116

**Word2Vec, for 10-K financial documents to SEC** 257

- background 257
- NLP solution 258
- questions 258
- takeaways 258

**Word2Vec, for medical fraud detection 250**

background 250  
NLP solution 251  
questions 251  
takeaways 251

**Word2Vec, for medical text annotation 252**

background 252  
NLP solution 252  
questions 252  
takeaways 253

**Word2Vec model**

data, loading 117  
text preprocessing 117  
training, in CBOW 118  
training, in CBOW and Skip-Gram 117  
training, in Skip-Gram 119  
visualizing, with t-SNE 120-123

**word embedding 6, 18**

encoding methods 18  
performing, with Bag-of-Words (BoW) 79  
performing, with Term Frequency-Inverse Document Frequency (TF-IDF) 79

**word embeddings, BERTopic 244**

creating, with BOW and TF-IDF 219

**WordNet 15, 42****words/concepts**

adding 114, 115  
subtracting 114, 115

**word to vector 10****word vectors 14**





[www.packtpub.com](http://www.packtpub.com)

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## Why subscribe?

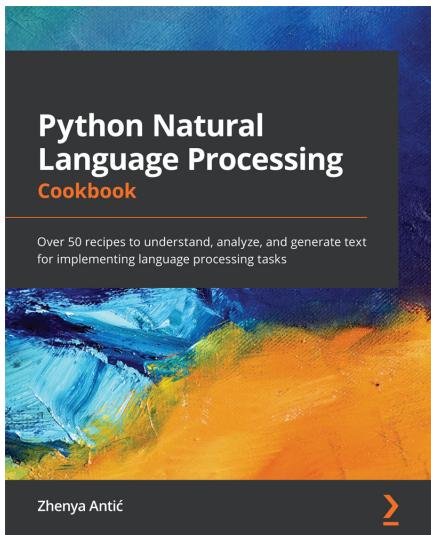
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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

At [www.packtpub.com](http://www.packtpub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

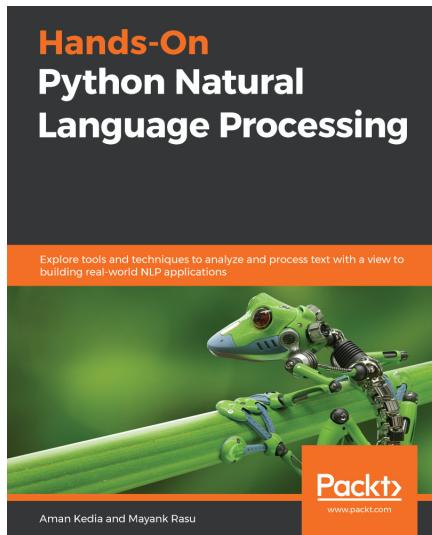


## Python Natural Language Processing Cookbook

Zhenya Antić

ISBN: 978-1-83898-731-2

- Become well-versed with basic and advanced NLP techniques in Python
- Represent grammatical information in text using spaCy, and semantic information using bag-of-words, TF-IDF, and word embeddings
- Perform text classification using different methods, including SVMs and LSTMs
- Explore different techniques for topic modeling such as K-means, LDA, NMF, and BERT
- Work with visualization techniques such as NER and word clouds for different NLP tools
- Build a basic chatbot using NLTK and Rasa
- Extract information from text using regular expression techniques and statistical and deep learning tools



## Hands-On Python Natural Language Processing

Aman Kedia, Mayank Rasu

ISBN: 978-1-83898-959-0

- Understand how NLP powers modern applications
- Explore key NLP techniques to build your natural language vocabulary
- Transform text data into mathematical data structures and learn how to improve text mining models
- Discover how various neural network architectures work with natural language data
- Get the hang of building sophisticated text processing models using machine learning and deep learning
- Check out state-of-the-art architectures that have revolutionized research in the NLP domain

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](https://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Share Your Thoughts

Now you've finished *The Handbook of NLP with Gensim*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

---

## Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781803244945>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly