

# KDnuggets

[Subscribe to KDnuggets News](#) || [Contact](#)

- [SOFTWARE](#)
- [News/Blog](#)
- [Top stories](#)
- [Opinions](#)
- [Tutorials](#)
- [JOBS](#)
- [Companies](#)
- [Courses](#)
- [Datasets](#)
- [EDUCATION](#)
- [Certificates](#)
- [Meetings](#)
- [Webinars](#)

[KDnuggets Home](#) » [News](#) » [2020](#) » [Jan](#) » [Tutorials, Overviews](#) » An Introductory Guide to NLP for Data Scientists with 7 Common Techniques ( [20:n02](#) )

## An Introductory Guide to NLP for Data Scientists with 7 Common Techniques

[<= Previous post](#)[Next post =>](#)

Tags: [Data Preparation](#), [NLP](#), [Sentiment Analysis](#), [TF-IDF](#), [Tokenization](#), [Topic Modeling](#), [Word Embeddings](#)

Data Scientists work with tons of data, and many times that data includes natural language text. This guide reviews 7 common techniques with code examples to introduce you the essentials of NLP, so you can begin performing analysis and building models from textual data.

---

By [George Seif](#), AI / Machine Learning Engineer.

[comments](#)



Modern organizations work with huge amounts of data. That data can come in a variety of different forms including documents, spreadsheets, audio recordings, emails, JSON, and so many, many more. One of the most common ways that such data is recorded is via text. That text is usually quite similar to the natural language that we use from day-to-day.

[Natural Language Processing](#) (NLP) is the study of programming computers to process and analyze large amounts of natural textual data. Knowledge of NLP is essential for Data Scientists since text is such an easy to use and common container for storing data.

Faced with the task of performing analysis and building models from textual data, one must know how to perform the basic Data Science tasks. That includes cleaning, formatting, parsing, analyzing, visualizing, and modeling the text data. It'll all require a few extra steps in addition to the usual way these tasks are done when the data is made up of raw numbers.

This guide will teach you the essentials of NLP when used in Data Science. We'll go through 7 of the most common techniques that you can use to handle your text data, including code examples with the [NLTK](#) and [Scikit Learn](#).

## (1) Tokenization

[Tokenization](#) is the processing of segmenting text into sentences or words. In the process, we throw away punctuation and extra symbols too.



This is not as simple as it looks. For example, the word “New York” in the first example above was separated into two tokens. However, New York is a pronoun and might be quite important in our analysis — we might be better off keeping it in just one token. As such, care needs to be taken during this step.

The benefit of Tokenization is that it gets the text into a format that’s easier to convert to raw numbers, which can actually be used for processing. It’s a natural first step when analyzing text data.

```
1 import nltk
2 sentence = "My name is George and I love NLP"
3 tokens = nltk.word_tokenize(sentence)
4 print(tokens)
5
6 # Prints out ['My', 'name', 'is', 'George', 'and', 'I', 'love', 'NLP']
```

nltk\_1.py hosted with ❤ by GitHub

[view raw](#)

## (2) Stop Words Removal

A natural next step after Tokenization is [Stop Words Removal](#). Stop Words Removal has a similar goal as Tokenization: get the text data into a format that’s more convenient for processing. In this case, stop words removal removes common language prepositions such as “and”, “the”, “a”, and so on in English. This way, when we analyze our data, we’ll be able to cut through the noise and focus in on the words that have actual real-world meaning.

Stop words removal can be easily done by removing words that are in a pre-defined list. An important thing to note is that there is no universal list of stop words. As such, the list is often created from scratch and tailored to the application being worked on.

```
1 import nltk
2 from nltk.corpus import stopwords
3
4 sentence = "This is a sentence for removing stop words"
5 tokens = nltk.word_tokenize(sentence)
6
7 stop_words = stopwords.words('english')
8 filtered_tokens = [w for w in tokens if w not in stop_words]
9 print(filtered_tokens)
10
```

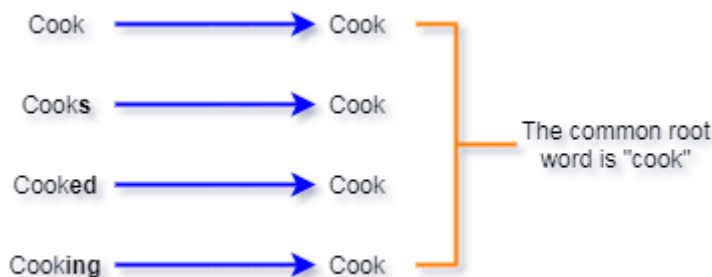
```
11 # Prints out ['This', 'sentence', 'removing', 'stop', 'words']
```

nltk\_2.py hosted with ❤ by GitHub

[view raw](#)

### (3) Stemming

[Stemming](#) is another technique for cleaning up text data for processing. Stemming is the process of reducing words into their root form. The purpose of this is to reduce words which are spelled slightly differently due to context but have the same meaning, into the same token for processing. For example, consider using the word “cook” in a sentence. There’s quite a lot of ways we can write the word “cook”, depending on the context:



All of these different forms of the word cook have essentially the same definition. So, ideally, when we’re doing our analysis, we’d want them to all be mapped to the same token. In this case, we mapped all of them to the token for the word “cook”. This greatly simplifies our further analysis of the text data.

```
1 import nltk
2
3 snowball_stemmer = nltk.stem.SnowballStemmer('english')
4
5 s_1 = snowball_stemmer.stem("cook")
6 s_2 = snowball_stemmer.stem("cooks")
7 s_3 = snowball_stemmer.stem("cooked")
8 s_4 = snowball_stemmer.stem("cooking")
9
10 # s_1, s_2, s_3, s_4 all have the same result
```

nltk\_3.py hosted with ❤ by GitHub

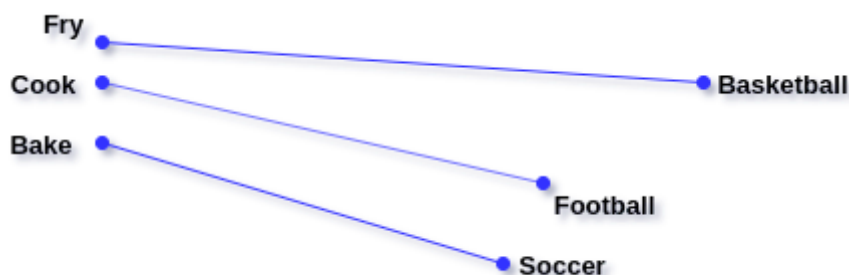
[view raw](#)

### (4) Word Embeddings

Now that our data is cleaned up from those first three methods, we can start converting it into a format that can actually be processed.

[Word embeddings](#) is a way of representing words as numbers, in such a way that words with similar meaning have a similar representation. Modern-day word embeddings represent individual words as real-valued vectors in a predefined vector space.

All word vectors have the same length, just with different values. The distance between two word-vectors is representative of how similar the meaning of the two words is. For example, the vectors of the words “cook” and “bake” will be fairly close, but the vectors of the words “football” and “bake” will be quite different.



A common method for creating word embeddings is called [GloVe](#), which stands for “Global Vectors”. GloVe captures global statistics and local statistics of a text corpus in order to create word vectors.

GloVe uses what’s called a *co-occurrence matrix*. A co-occurrence matrix represents how often each pair of words occur together in a text corpus. For example, consider how we would create a co-occurrence matrix for the following three sentences:

- I love Data Science.
- I love coding.
- I should learn NLP.

The co-occurrence matrix of this text corpus would look like this:

	I	love	Data	Science	coding	should	learn	NLP
I	0	2	1	1	1	1	1	1
love	2	0	1	1	1	0	0	0
Data	1	1	0	1	0	0	0	0
Science	1	1	1	0	0	0	0	0
coding	1	1	0	0	0	0	0	0
should	1	0	0	0	0	0	1	1
learn	1	0	0	0	0	1	0	1
NLP	1	0	0	0	0	1	1	0

For a real-world dataset, the matrix would be much, **much** larger. The good thing is that the word embeddings only have to be computed once for the data and can then be saved to disk.

GloVe is then trained to learn vectors of fixed length for each word, such that the dot product of any two-word vectors equals the logarithm of the words’ probability of co-occurrence, which comes from the co-occurrence matrix. This is represented in the objective function of the paper shown below:

$$J = \sum_{i,j=1}^V f(X_{ij}) (w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij})^2$$

In the equation,  $X$  represents the value from the co-occurrence matrix at position  $(i,j)$ , and the  $w$ ’s are the word vectors to be learned. Thus, by using this objective function, GloVe is minimizing the difference between the dot product of two-word vectors and the co-occurrence, effectively ensuring that the learned vectors are correlated with the co-occurrence values in the matrix.

Over the past few years, GloVe has proved to be a very robust and versatile word embedding technique, due to its effective encoding of the meanings of the words, and their similarity. For Data Science applications, it’s a battle-tested method for getting words into a format that we can process and analyze.

Here's a [full tutorial](#) about how to use GloVe in Python!

## (5) Term Frequency-Inverse Document Frequency

[Term Frequency-Inverse Document Frequency](#), more commonly known as TF-IDF is a weighting factor often used in applications such as information retrieval and text mining. TF-IDF uses statistics to measure how important a word is to a particular document.

- **TF — Term Frequency:** measures how frequently a string occurs in a document. Calculated as the total number of occurrences in the document divided by the total length of the document (for normalization).
- **IDF — Inverse Document Frequency:** measures the importance of a string within a document. For example, certain strings such as “is”, “of”, and “a”, will appear a lot of times in many documents but don't really hold much meaning— they're not adjectives or verbs. IDF, therefore, weights each string according to its importance, calculated as the  $\log()$  of the total number of documents in the dataset divided by the number of documents that the string occurs in (+1 in the denominator to avoid a division by zero).
- **TF-IDF:** The final calculation of the TF-IDF is simply the multiplication of the TF and IDF terms:  $TF * IDF$ .

The TF-IDF is perfectly balanced, considering both local and global levels of statistics for the target word. Words that occur more frequently in a document are weighted higher, but only if they're more rare within the whole document.

Due to its robustness, TF-IDF techniques are often used by search engines in scoring and ranking a document's relevance given a keyword input. In Data Science, we can use it to get an idea of which words, and related information, are the most important in our text data.

```
1  import pandas as pd
2  from sklearn.feature_extraction.text import TfidfVectorizer
3
4  def get_tf_idf(vectorizer):
5      feature_names = vectorizer.get_feature_names()
6      dense_vec = vectors.todense()
7      dense_list = dense_vec.tolist()
8      tfidf_data = pd.DataFrame(dense_list, columns=feature_names)
9      return tfidf_data
10
11
12  vectorizer = TfidfVectorizer()
13
14  doc_1 = "TF-IDF uses statistics to measure how important a word is to " \
15         "a particular document"
16  doc_2 = "The TF-IDF is perfectly balanced, considering both local and global " \
17         "levels of statistics for the target word."
18  doc_3 = "Words that occur more frequently in a document are weighted higher, " \
19         "but only if they're more rare within the whole document."
20  documents_list = [doc_1, doc_2, doc_3]
21
```

```
22 vectors = vectorizer.fit_transform(documents_list)
23
24 tfidf_data = get_tf_idf(vectorizer)
25
26 print(tfidf_data)
27 # Prints the TF-IDF data for all words across all documents
```

scikit\_learn\_1.py hosted with ❤️ by GitHub

[view raw](#)

## (6) Topic Modeling

[Topic modeling](#), in the context of NLP, is the process of extracting the main topics from a collection of text data or documents. Essentially, it's a form of [Dimensionality Reduction](#) since we're reducing a large amount of text data down to a much smaller number of topics. Topic modeling can be useful in a number of Data Science scenarios. To name a few:

- Data analysis of the text — Extracting the underlying trends and main components of the data
- Classifying the text — In a similar way that dimensionality reduction helps with classical Machine Learning problems, topic modeling also helps here since we are compressing the text into the key features, in this case, the topics
- Building recommender systems — topic modeling automatically gives us some basic grouping for the text data. It can even act as an additional feature for building and training the model

Topic modeling is typically done using a technique called [Latent Dirichlet Allocation](#) (LDA). With LDA, each text document is modeled as a multinomial distribution of topics, and each topic is modeled as a multinomial distribution of words (individual strings, which we can get from our combination of tokenization, stop words removal, and stemming).

LDA assumes documents are produced from a combination of topics. Those topics then generate words based on their probability distribution.

We start by telling LDA how many topics each document should have, and how many words each topic is made up of. Given a dataset of documents, LDA attempts to determine what combination and distribution of topics can accurately re-create those documents and all the text in them. It can tell which topic(s) works by building the actual documents, where the building is done by sampling words according to the probability distributions of the words, given the selected topic.

Once LDA finds a distribution of topics that can most accurately re-create all of the documents and their contents within the dataset, then those are our final topics with the appropriate distributions.

```
1 from sklearn.decomposition import LatentDirichletAllocation as LDA
2
3 NUM_TOPICS = 3
4
5 # Here we create and fit the LDA model
6 # The "document_word_matrix" is a 2D array where each row is a document
7 # and each column is a word. The cells contain the count of the word within
8 # each document
9 lda = LDA(n_components=NUM_TOPICS, n_jobs=-1)
```

```
10 lda.fit(document_word_matrix)
```

scikit\_learn\_2.py hosted with ❤ by GitHub

[view raw](#)

## (7) Sentiment Analysis

[Sentiment Analysis](#) is an NLP technique that tries to identify and extract the subjective information contained within text data. In a similar way to Topic Modeling, Sentiment Analysis can help transform unstructured text into a basic summary of the information embedded in the data.

Most Sentiment Analysis techniques fall into one of two buckets: rule-based and Machine Learning methods. The rule-based method follows simple steps to achieve their results. After doing some text pre-processing like tokenization, stop words removal, and stemming, a rule-based may, for example, go through the following steps:

1. Define lists of words for the different sentiments. For example, if we are trying to determine if a paragraph is negative or positive, we might define words like *bad* and *horrible* for the negative sentiment, and *great* and *amazing* for the positive sentiment
2. Go through the text and count the number of positive words. Do the same thing for the negative words.
3. If the number of words identified as positive is greater than the number of words identified as negative, then the sentiment of the text is *positive* — and vice versa for *negative*

Rule-based methods are great for getting a general idea of how Sentiment Analysis systems work. Modern, state-of-the-art systems, however, will typically use Deep Learning, or at least classical Machine Learning techniques, to automate the process.

With Deep Learning techniques, sentiment analysis is modeled as a classification problem. The text data is encoded into an embedding space (similar to the Word Embeddings describe above) — this is a form of feature extraction. These features are then passed to a classification model where the sentiment of the text is classified.

This learning-based approach is powerful since we can automate it as an optimization problem. The fact that we can continuously feed data to the model to get a continuous improvement out of it is also a huge bonus. More data improves both feature extraction and sentiment classification.

There are a number of great tutorials on how to do Sentiment Analysis with various Machine Learning models. Here are a few great ones:

- With [Logistic Regression](#)
- With [Random Forest](#)
- With [Deep Learning LSTM](#)

### Related:

- [Automatic Text Summarization in a Nutshell](#)
- [Text Encoding: A Review](#)
- [Understanding NLP and Topic Modeling Part 1](#)



**What do you think?**

23 Responses



Upvote



Funny



Love



Surprised



Angry



Sad

**2 Comments****KDnuggets****buckets-maestro** ▾

Recommend

Tweet

Share

Sort by Best ▾



Join the discussion...

**justintaylor** • 18 days ago

this is a horrible article. very kindergarten level. Yes these are the basics but you ignore so much. for shame.

^ | ▾ 1 • Reply • Share ›

**Omran Carso** → justintaylor • 9 days ago • edited

This is an "An Introductory Guide to NLP" based on that, it does exactly what it said it would be.

^ | ▾ • Reply • Share ›

[<= Previous post](#)[Next post =>](#)**Top Stories Past 30 Days****Most Popular**

1. [The Book to Start You on Machine Learning](#)
2. [Top 5 must-have Data Science skills for 2020](#)
3. [How To Ultralearn Data Science: summary, for those in a hurry.](#)
4. [Math for Programmers!](#)
5. [I wanna be a data scientist, but... how?](#)

**Most Shared**

1. [The Book to Start You on Machine Learning](#)
2. [I wanna be a data scientist, but... how?](#)
3. [An Introductory Guide to NLP for Data Scientists with 7 Common Techniques](#)
4. [7 Resources to Becoming a Data Engineer](#)
5. [Automated Machine Learning: How do teams work together on an AutoML project?](#)

- ## Latest News

- [Data Validation for Machine Learning](#)
- [How to land a Data Scientist job at your dream company.](#)
- [OpenAI is Adopting PyTorch... They Aren't Alone](#)
- [Geek & Chic: Analytics redefining fashion instincts](#)
- [Top 10 AI, Machine Learning Research Articles to know](#)
- [How to Optimize Your Jupyter Notebook](#)

## Top Stories Last Week

## Most Popular

- ## 1. I wanna be a data scientist, but... how?



2. [Top 10 Technology Trends for 2020](#)
3. [The Book to Start You on Machine Learning](#)
4. [Math for Programmers!](#)
5. [Decision Tree Algorithm, Explained](#)
6. [The Data Science Interview Study Guide](#)
7. [Top 5 must-have Data Science skills for 2020](#)

## Most Shared

1. [I wanna be a data scientist, but... how?](#)
2. [Microsoft Introduces Project Petridish to Find the Best Neural Network for Your Problem](#)
3. [10 Python String Processing Tips & Tricks](#)
4. [Random Forest — A Powerful Ensemble Learning Algorithm](#)
5. [Artificial Intelligence Books to Read in 2020](#)
6. [The 5 Most Useful Techniques to Handle Imbalanced Datasets](#)
7. [Semi-supervised learning with Generative Adversarial Networks](#)

## More Recent Stories

- [How to Optimize Your Jupyter Notebook](#)
- [Amazon Gets Into the AutoML Race with AutoGluon: Some AutoML A...](#)
- [Top tweets, Jan 22-28: The 5 Most Useful Techniques to Hand...](#)
- [Top 25 Session Highlights at ODSC East 2020](#)
- [Managing Machine Learning Cycles: Five Learnings from comparin...](#)
- [Generating English Pronoun Questions Using Neural Coreference ...](#)
- [Google Dataset Search Provides Access to 25 Million Datasets](#)
- [KDnuggets 20:n04, Jan 29: AutoML: If you try it,.you'...](#)
- [A bird's-eye view of modern AI from NeurIPS 2019](#)
- [Data Scientist Archetypes](#)
- [Exoplanet Hunting Using Machine Learning](#)
- [AutoML Poll results: if you try it,.you'll like it more](#)
- [Agenda Sneak Peek for PAW for Industry 4.0 | Munich, 11-12 May](#)
- [The Decade of Data Science](#)
- [Uber Has Been Quietly Assembling One of the Most Impressive Op...](#)
- [Top Stories, Jan 20-26: I wanna be a data scientist, but how?](#)
- [You're Fired: How to develop and manage a happy data science...](#)
- [2 Questions for a Junior Data Scientist](#)
- [Semi-supervised learning with Generative Adversarial Networks](#)
- [Top 7 Location Intelligence Companies in 2020](#)

[KDnuggets Home](#) » [News](#) » [2020](#) » [Jan](#) » [Tutorials, Overviews](#) » An Introductory Guide to NLP for Data Scientists with 7 Common Techniques ( [20:n02](#) )

© 2020 KDnuggets. [About KDnuggets](#). [Privacy policy](#). [Terms of Service](#)

[Subscribe to KDnuggets News](#)



X