# LLMS FOUNDATIONS & INFRASTRUCTURE

# Introduction to NLP

# Natural Language

- Natural languages are different from computer programming languages.
- They aren't intended to be translated into a finite set of mathematical operations, like programming languages are.
- But there are no compilers or interpreters for natural languages such as French and English.
- Also, natural language is complex!

# Natural Language Processing (NLP)

- Natural language processing (NLP) is an area of research concerned with processing natural languages such as French and English.
- This processing generally involves translating natural language into data (numbers) that a computer can use to learn about the world.

# Interest in NLP is High

Cloud service providers released dedicated services for NLP:

- Google, with its Google Cloud: https://cloud.google.com/natural-language
- Microsoft, with its Azure cognitive services: https://azure.microsoft.com/en-us/services/cognitive-services/text-analytics/
- AWS, with its comprehend service: https://aws.amazon.com/comprehend/

# Interest in NLP is High

- A company in 2021 can't do much without data.
- Social media provide loads of user data.
- Most data come in text form.
- Means to analyze said data is NLP

# Why NLP?

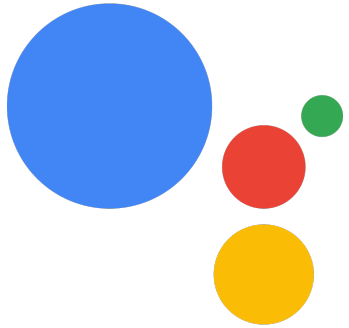
- There are billions of text data being generated every day; In-apps messages (Whatsapp, WeChat, Telegram etc.), social media (Facebook, Instagram, Twitter, YouTube etc.), forums (Quora, Reddit etc.), blogs, news publishing platforms, google searches and many other channels.

- Because of the large volumes of text and the highly unstructured data source, we can no longer use the common approach to understand the text.

- This is where NLP comes in.

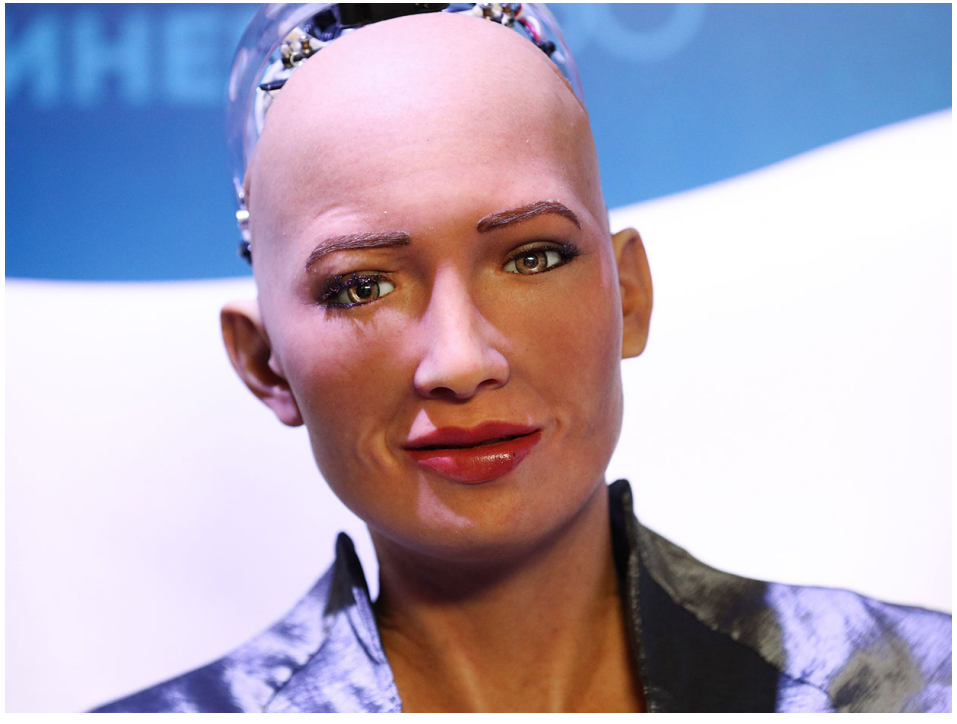
# Why NLP?

- Chances are you've already been using some of the most powerful NLP applications.



# NLP Challenges

- It's hard!
- Machines with the capability of processing something natural isn't natural.
- A machine with perfect language understanding falls into the "Uncanny valley"



# AI Gone Uncanny (Google Duplex)

# NLP Challenges

- Human language contains a structure that is foreign to machine-structured language.
- Unlike machine languages, natural language is much more flexible and can have much complex structures that prohibits regular programming techniques from analyzing it.
- You can rely on statistical relationships between words instead of a deep system of logical rules.

# NLP Challenges

- Imagine if you had to define English grammar and spelling rules in a nested tree of if…then statements. Could you ever write enough rules to deal with every possible way that words, letters, and punctuation can be combined to make a statement?
- Imagine how limited and brittle this software would be.

# NLP Challenges Examples

- When I say "good morning", I assume that:
- You have some knowledge about what makes up a morning:
  - _ mornings come before noons and evenings
  - _ morning comes after midnights
  - _ morning can represent times of day and general experiences of a period.
- You know that "good morning" is a common greeting that doesn't contain much information at all about the morning. Rather it reflects the state of mind of the speaker and their readiness to speak with others.
- You receive my intention of positive sentiment, where the phrase sends a message of friendliness and pleasant behavior.

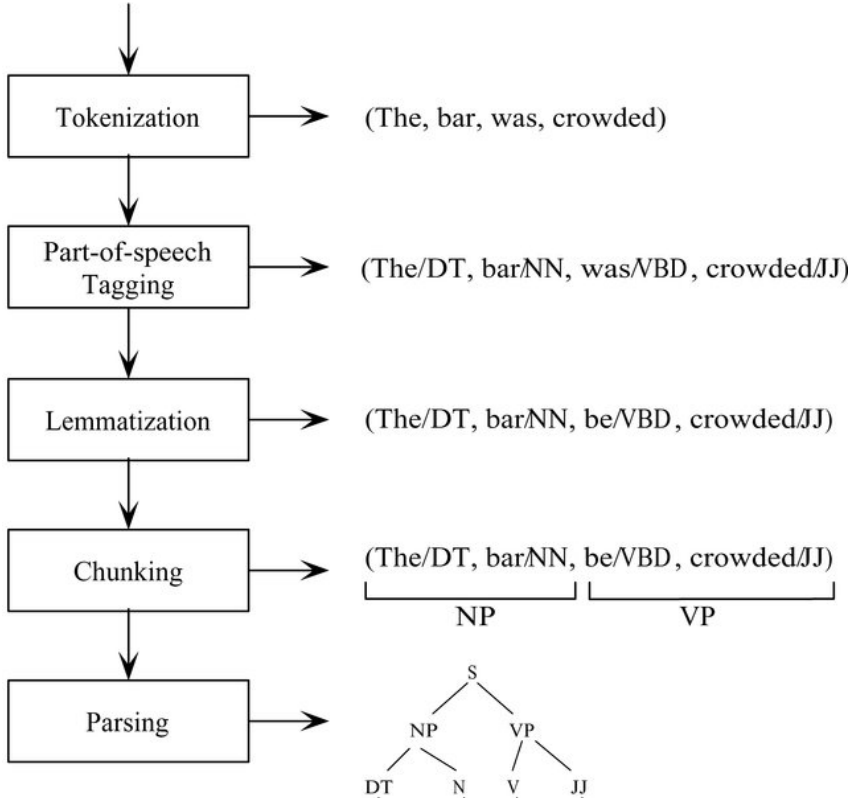
# Note

- In this course, we'll focus on processing English text, but keep in mind that the process is the same for all other languages.
- We'll also be using Python as the programming language for its simplicity and abundancy of nlp libraries.

# Text Preprocessing

# What is Preprocessing

- Any real corpus in its raw form is unusable for analytics without significant preprocessing.
- Text preprocessing is the transformation of raw data into clean usable data sets.
- There are loads of textual data sources on the web, which need to be transformed into a form that is ready for computation and modeling.



# Part 1: Data Collection

# Data Sources

Text data mainly comes from 3 sources:

1. **Open-source datasets**: text datasets that have been collected and sometimes labeled by researchers for specific purposes. Like the Amazon reviews dataset, the IMDB dataset and the Stanford Question Answering Dataset (SQuAD).
2. **The open web**: these are html pages which we can fetch and parse, like web posts and Wikipedia articles.
3. **Social media**: probably your richest source of data. Collecting social media posts is done through their respective APIs, and the data usually requires a bit more pre-processing than usual web posts.

# HTML Parsing

- In the case when we collect text from HTML pages, we need to strip-down any un-needed HTML tags:
  1. They are not needed for NLP
  2. They have a negative impact on any model we might want to train.
- There are tools and libraries that can parse text from HTML content, similar to JavaScript. In our case, we will look at a Python library called **BeautifulSoup**.

# Web Scraping

- Web scraping is the process of gathering information from the Internet, usually in a process that involves automation.
- Some websites don't like it when automatic scrapers gather their data, while others don't mind. Hence, it's always a good idea to do some research to make sure that we're not violating any Terms of Service before we start a large-scale project.

# Web Scraping

- When we scrape the web, we write code that sends a request to the server hosting our target page, downloading that page's source code, just as a browser would.
- But instead of displaying the page visually, it filters through the page looking for the HTML elements that we specify, and extracts whatever content we instruct it to.

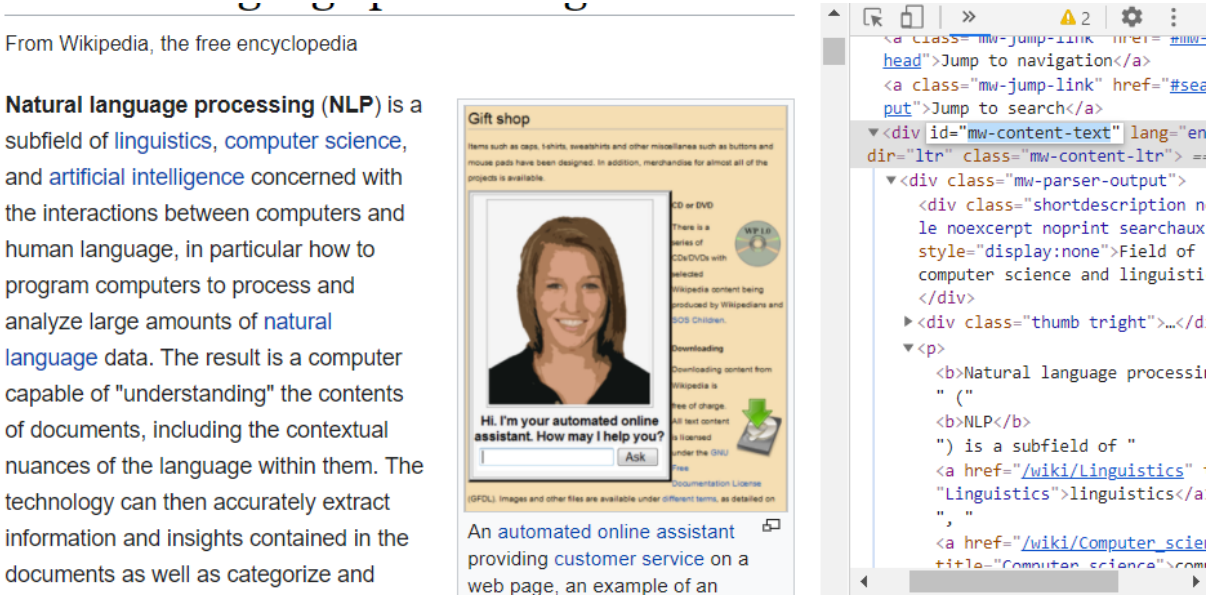
# Web Scraping

```
<div>
        <span>
                <p>This is my target</p>
        </span>
        <b>This is not my target</b>
</div>
```

# Web Scraping

Inspecting web pages:

- First start by inspecting the html content of that page in a browser. Let's take the Wikipedia NLP page (https://en.wikipedia.org/wiki/Natural_language_processing) for example.
- Open the web page, right click and select "inspect". This allows us to go through the different HTML tags of the page.
- We find, for example, that the page's main content exist in a " " element of class "mw-parser-output", and the text exists within "<p>" and "<ul>" tags inside said main .
- This information is very useful because it will help us pick out this content when time comes to start scraping.



# Web Scraping

Static Web pages

- The website we're scraping in this section serves static HTML content. In this scenario, the server that hosts the site sends back HTML documents that already contain all the data we'll get to see as users.
- This is what we call a **Static Website**. Static sites are easier to work with because the server sends us an HTML page that already contains all the information as a response. We can parse an HTML response with Beautiful Soup and begin to pick out the relevant data.

# Web Scraping

Dynamic Web pages

- On the other hand, with a dynamic website the server might not send back any HTML at all. Instead, we'll receive JavaScript code as a response, which gets executed and fetches the rest of the page's content.
- Doing a request to a dynamic website in a Python script will not provide the HTML page content.
- The "requests" python library can't execute JavaScript commands to get the page's content, but there are other solutions that can.
- As an example of a dynamic website, we can look at Facebook or Twitter.

# Web Scraping

Parsing with Beautiful Soup

- Beautiful Soup is a Python library for parsing structured data. It allows to interact with HTML in a similar way to how we would interact with a web page using the browser's developer tools.
- We first import the needed libraries, fetch the html content, and create our BeautifulSoup parser.

```
import requests
from bs4 import BeautifulSoup

URL = ' https://en.wikipedia.org/wiki/Natural_language_processing'
page = requests.get(URL)
soup = BeautifulSoup(page.content, 'html.parser')
```

# Web Scraping

Then we can specify what elements are we searching for. Recall that we're interested in the paragraph tags inside the main content div of class "mw-parser-output".

```
main_container = soup.findAll('div',attrs={"class":" mw-parser-
output"})
for content in main_container:
    for par in content.findAll('p', recursive=False):
        print(par.text)
```

# Web Scraping

The output is the text content we're aiming to process

```
Natural language processing (NLP) is a subfield of linguistics,…
Challenges in natural language processing frequently involve speech…
Natural language processing has its roots in the 1950s.
Already in 1950,…
…
```

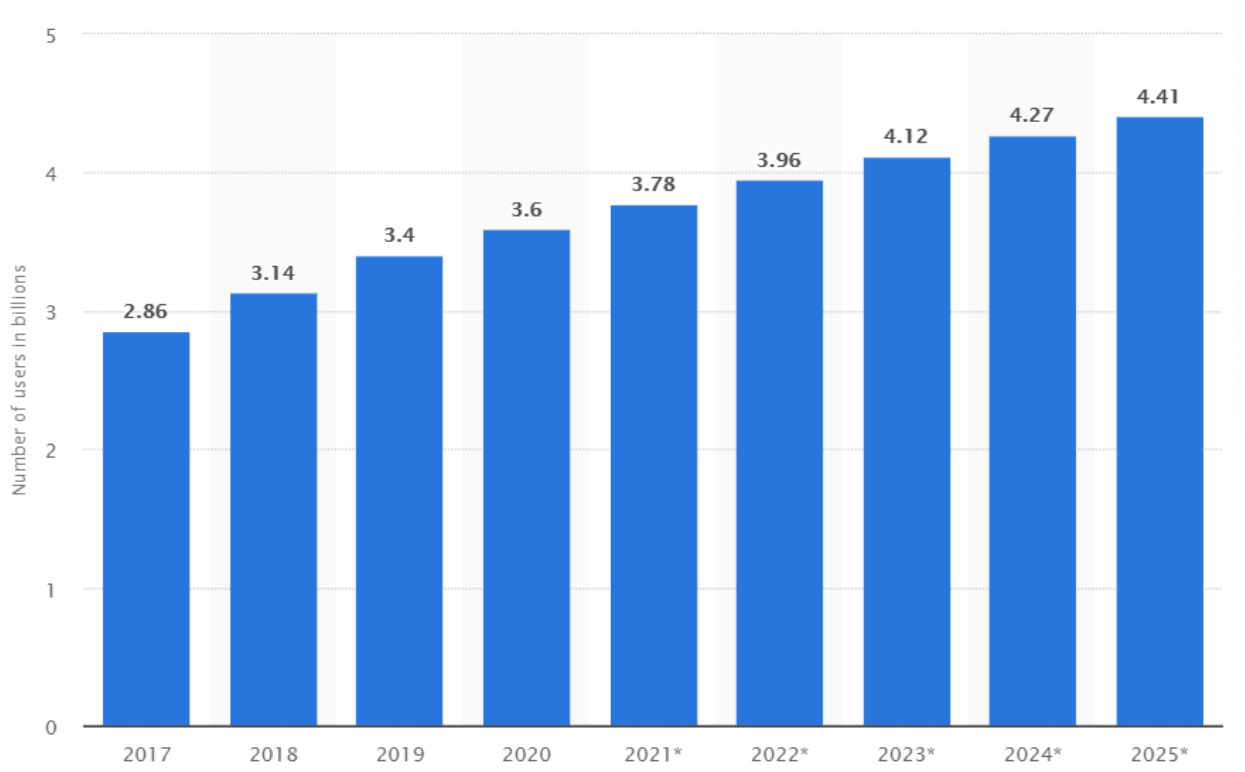
For more information about the BeautifulSoup library, a comprehensive documentation exists here:
https://www.crummy.com/software/BeautifulSoup/bs4/doc/

# Social Media Data Collection

- Social media can be a gold mine of data.
- Platforms such as Twitter and Facebook lend themselves to holding useful information.
- Users may post unfiltered opinions that are able to be retrieved with ease.
- Add to that the sheer number of users and amount of data that users post.

# Social Media Data Collection

- Number of social network users worldwide from 2017 to 2025 (in billions)



# Social Media Data Collection

- Fetching information from social media networks is done by using their respective API.
- We'll take a look at how we can scrape Tweets from Twitter using the python library "Tweepy", a Python library for accessing the Twitter API.
- Tweepy is able to accomplish various tasks beyond just querying tweets like:
  - Post, retrieve, and interact with Tweets.
  - Post and receive direct messages
  - Follow, search and get results

# Social Media Data Collection

Limitations:

- The standard twitter API only allows to retrieve tweets up to 7 days ago and is limited to scraping 18,000 tweets per a 15 minute window.
- Using Tweepy, we're only able to return up to 3,200 of a single user's most recent tweets.
- If you want to scrape Twitter, you need to become a twitter developer first. You can read more about this here: https://developer.twitter.com/en/apply-for-access

# Social Media Data Collection

After becoming a twitter developer, you get credentials that you provide in your python code.

```
consumer_key = "XXXXXXXXX"
consumer_secret = "XXXXXXXXX"
access_token = "XXXXXXXXX"
access_token_secret = "XXXXXXXXX"
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)
api = tweepy.API(auth,wait_on_rate_limit=True)

# fetch tweets by username
username = 'target_twitter_username'

# fetch tweets by a search query
text_query = '2020 US Election'
```

# Social Media Data Collection

```
count = 150
try:
    # Creation of query method using parameters with username
    tweets = tweepy.Cursor(api.user_timeline,id=username).items(count)

    # Creation of query method using parameters with search query
    tweets = tweepy.Cursor(api.search,q=text_query).items(count)

    # Pulling information from tweets iterable object
    tweets_list = [[tweet.created_at, tweet.id, tweet.text] for tweet in tweets]

    # Creation of dataframe from tweets list
    # Add or remove columns as you remove tweet information
    tweets_df = pd.DataFrame(tweets_list)
except BaseException as e:
    print('failed on_status,',str(e))
    time.sleep(3)
```

# Social Media Data Collection

You receive a tweet object, which is a JSON object structured as follows:

```
{
  "created_at": "Thu Apr 06 15:24:15 +0000 2017",
  "id_str": "…",
  "text": "",
  "user": {
    "id": …,
    "name": "Twitter Dev",
    "screen_name": "TwitterDev",
    "location": "Internet",
    "url": "https:\/\/dev.twitter.com\/",
    "description": ""
  },
  "place": {},
```

```
  "entities": {
      "hashtags": [],
      "urls": [
        {
          "url": "…",
          "unwound": {
            "url": "",
            "title": "Building…"
          }
        }
      ],
      "user_mentions": [
      ]
    }
}
```

Further information about twitter objects are available here:
https://developer.twitter.com/en/docs/twitter-api/v1/data-dictionary/object-model/tweet

# Part 2: Build your vocabulary (word tokenization)

# Introduction

- **Question:** So you're ready to save the world with the power of natural language processing?

- Well the first thing you need is a powerful vocabulary.


- This section will help you split a **document**, any **string**, into **discrete tokens** of meaning.

# Introduction

- Our **tokens** are limited to:
  - words
  - punctuation marks
  - Numbers

- but the techniques we use are easily **extended** to any other units of meaning contained in a sequence of characters, like:
  - ASCII emoticons
  - Unicode emojis
  - mathematical symbols
  - …

# Introduction

- Retrieving **tokens** from a document will require some string manipulation beyond just the **str.split()** method employed before

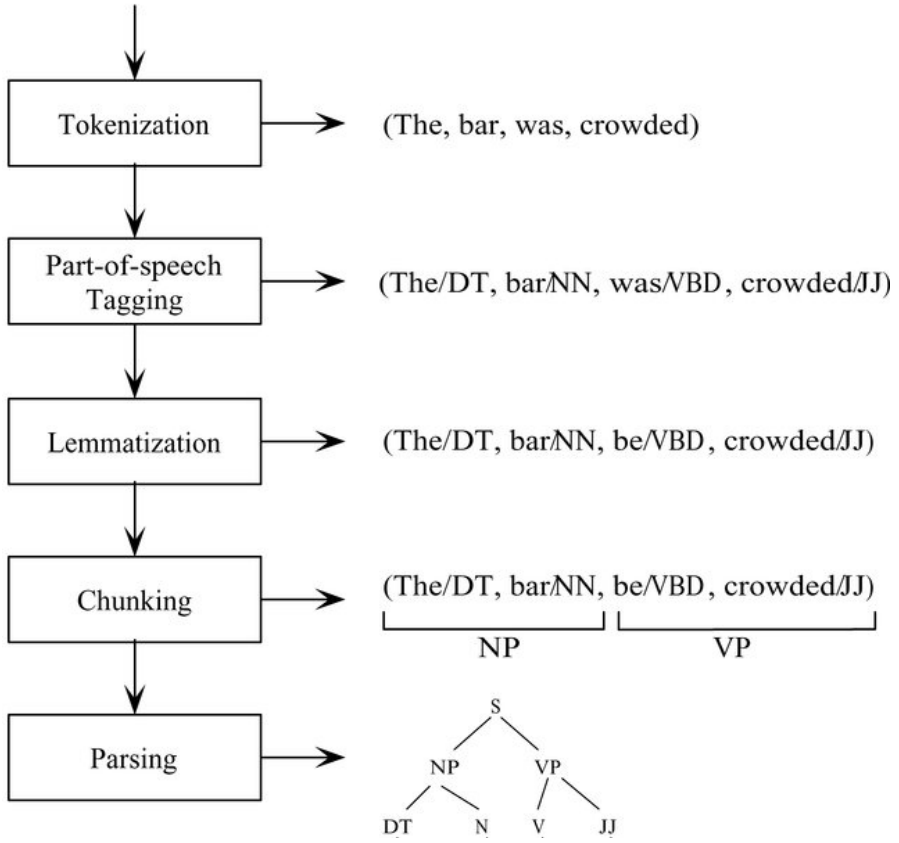
- You'll need to:
  - separate punctuation from words, ex.: quotes at the beginning and end of a statement.
  - split contractions like "we'll" into the words that were combined to form them.

Natural Language Processing

['Natural', 'Language', 'Processing']

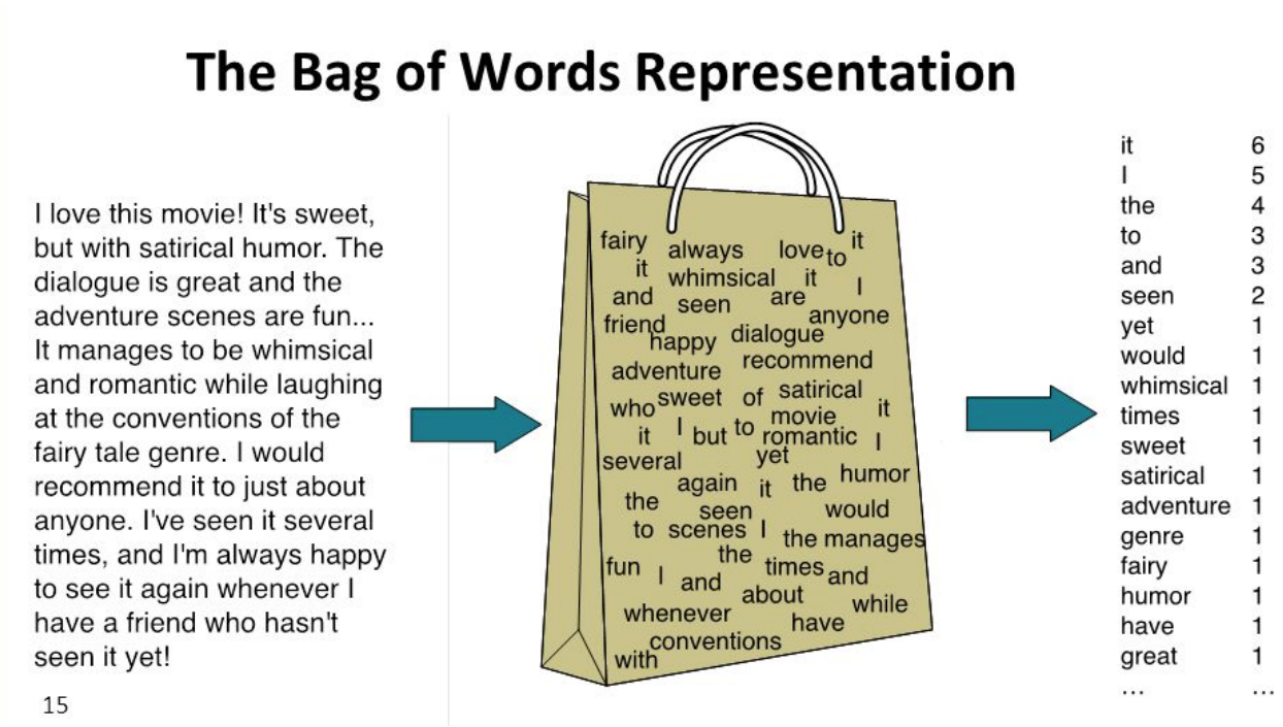
# Introduction

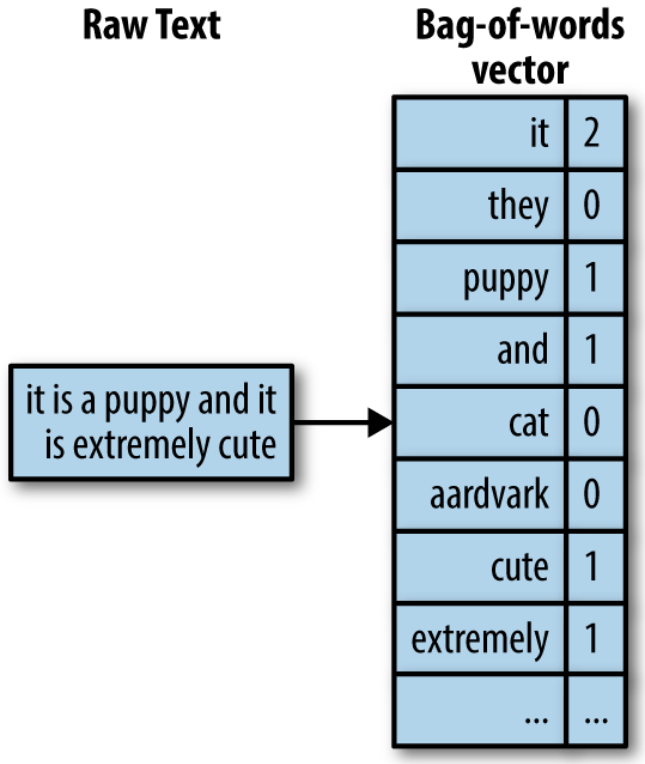
- Why do we apply tokenization? It is an essential part of the preprocessing pipeline that allows us to segment entire documents into "tokens" which we can apply the rest of our processing on.



# Introduction

- After tokenization, you'll assemble a vector representation of your documents called a **bag of words**, and you'll try to use this vector to train ML models.



# Tokenization

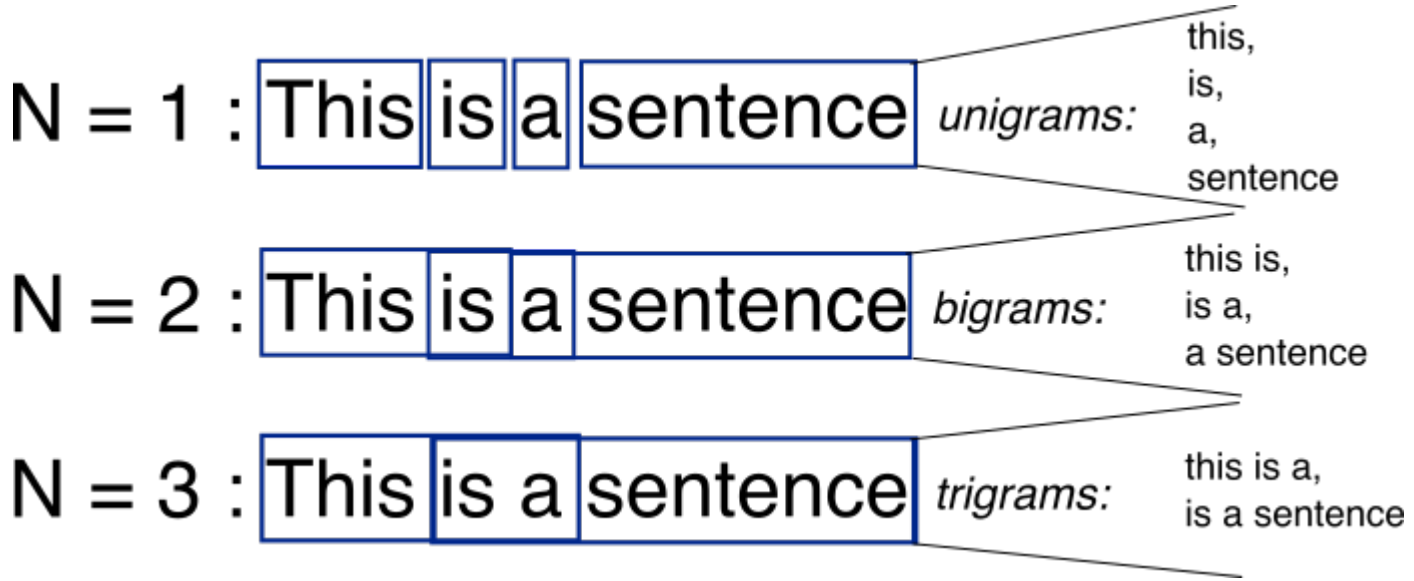
- Generally, we tokenize by word; Meaning: we segment a document into sentences, and the latter into individual words.
- Words themselves can be divided up into smaller meaningful parts.
- Syllables, prefixes, and suffixes, like “re” (reprogrammable), “pre” (precaution), and “ing” (programming) have intrinsic meaning.

# Introduction

- we show you straightforward algorithms for separating a string into words.
- You'll also extract pairs, triplets, quadruplets, and even quintuplets of tokens.
- These are called **n-grams**:
  - Pairs of words are 2-grams (bigrams),
  - Triplets are 3-grams (trigrams),
  - Quadruplets are 4-grams,
  - and so on.

# Introduction

- Example: using n-grams enables your machine to know about "ice cream" as well as the "ice" and "cream" that comprise it.
- Example: another 2-gram that you'd like to keep together is "Mr. Smith." Your tokens and your vector representation of a document will have a place for "Mr. Smith" along with "Mr." and "Smith," too.



# Introduction

- For now, all possible pairs (and short n-grams) of words will be included in your vocabulary.
- You'll find that the approaches we show aren't perfect.
- Feature extraction can rarely retain all the information content of the input data in any machine learning pipeline.

# Introduction

- In natural language processing, composing a numerical vector from text is a particularly "lossy" feature extraction process (order of words – Loss of capitalization – Tense of words …)

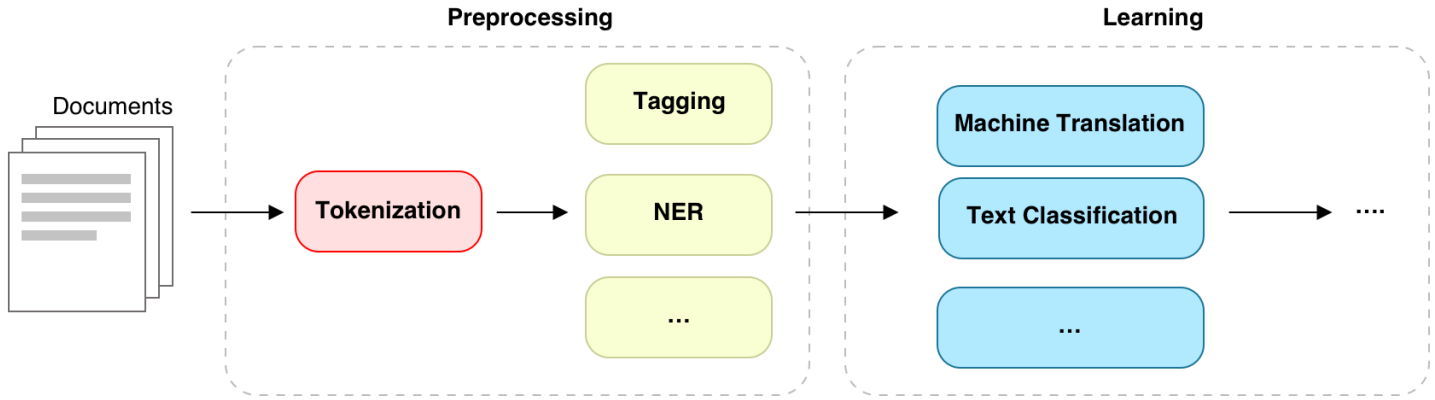
- Nonetheless the bag-of-words (BOW) vectors retain enough of the information content of the text to produce useful and interesting machine learning models.

# Building your vocabulary with a tokenizer

- In NLP, tokenization is a particular kind of document segmentation.
- Segmentation breaks up text into smaller chunks or segments, with more focused information content.
- Segmentation can include breaking:
  - a document into paragraphs,
  - paragraphs into sentences,
  - sentences into phrases,
  - or phrases into tokens (usually words) and punctuation.
- We focus on segmenting text into tokens, which is called tokenization.

# Building your vocabulary with a tokenizer

- Tokenization is the first step in an NLP pipeline, so it can have a big impact on the rest of your pipeline.
- A tokenizer breaks unstructured data, natural language text, into chunks of information that can be counted as discrete elements.
- These counts of token occurrences in a document can be used directly as a vector representing that document.



# Building your vocabulary with a tokenizer

- This immediately turns an unstructured string (text document) into a numerical data structure suitable for machine learning.

- The most common use for bag-of-words vectors created this way is for document retrieval, or search.

# Building your vocabulary with a tokenizer

- The simplest way to tokenize a sentence is to use whitespace within a string as the "delimiter" of words.

- In Python, this can be accomplished with the standard library method **split**, which is available on all **str** object instances as well as on the str built-in class itself.

```
"Thomas Jefferson began building Monticello at the age of 26.".split()
```

```
['Thomas', 'Jefferson', 'began', 'building',
 'Monticello', 'at', 'the', 'age', 'of', '26.']
```

# Building your vocabulary with a tokenizer

- this built-in Python method already does a decent job tokenizing a simple sentence.
- Its only "mistake" was on the last word, where it included the sentence-ending punctuation with the token "26."
- Normally you'd like tokens to be separated from neighboring punctuation and other meaningful tokens in a sentence.

# Building your vocabulary with a tokenizer

- A good tokenizer should strip off this extra character to create the word "26" as an equivalent class for the words "26," "26!", "26?", and "26."

- And a more accurate tokenizer would also output a separate token for any sentence-ending punctuation so that a sentence segmenter or sentence boundary detector can find the end of that sentence.

# Building your vocabulary with a tokenizer

- With a bit more Python, you can create a numerical vector representation for each word, called **one-hot vectors**.
- A sequence of these one-hot vectors fully captures the original document text in a sequence of vectors, a table of numbers.
- That will solve the first problem of NLP, turning words into numbers.

```
import numpy as np
token_sequence = "Thomas Jefferson began building Monticello at the age of 26.".split()
vocab = sorted(set(token_sequence))
num_tokens = len(token_sequence)
vocab_size = len(vocab)
onehot_vectors = np.zeros((num_tokens, vocab_size), int)
for i, word in enumerate(token_sequence):
    onehot_vectors[i, vocab.index(word)] = 1
```

```
print(' '.join(token_sequence))
print(' '.join(vocab))
print(onehot_vectors)
```

```
Thomas Jefferson began building Monticello at the age of 26.
26. Jefferson Monticello Thomas age at began building of the
```

```
array([[0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
       [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
       [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

# Building your vocabulary with a tokenizer

- If you have trouble quickly reading all those ones and zeros, you're not alone.
- Pandas DataFrames can help make this a little easier on the eyes and more informative.
- Pandas wraps a 1D array with some helper functionality in an object called a **Series**.
- And Pandas is particularly handy with tables of numbers like:
  - lists of lists,
  - 2D numpy arrays,
  - 2D numpy matrices,
  - arrays of arrays,
  - dictionaries of dictionaries,
  - and so on.

# Building your vocabulary with a tokenizer

- A **DataFrame** keeps track of labels for each column, allowing you to label each column in our table with the token or word it represents.
- A **DataFrame** can also keep track of labels for each row in the **DataFrame.index**, for speedy lookup.
- But this is usually just a consecutive integer for most applications.
- For now you'll use the default index of integers for the rows in your table of one-hot word vectors for this sentence about Thomas Jefferson, shown in the following listing.

# Building your vocabulary with a tokenizer

```
import pandas as pd
pd.DataFrame(onehot_vectors, columns=vocab)
```

| | 26. | Jefferson | Monticello | Thomas | age | at | began | building | of | the |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| **4** | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **5** | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| **6** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| **7** | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| **8** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| **9** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Building your vocabulary with a tokenizer

```
import pandas as pd
pd.DataFrame(onehot_vectors, columns=vocab)
```

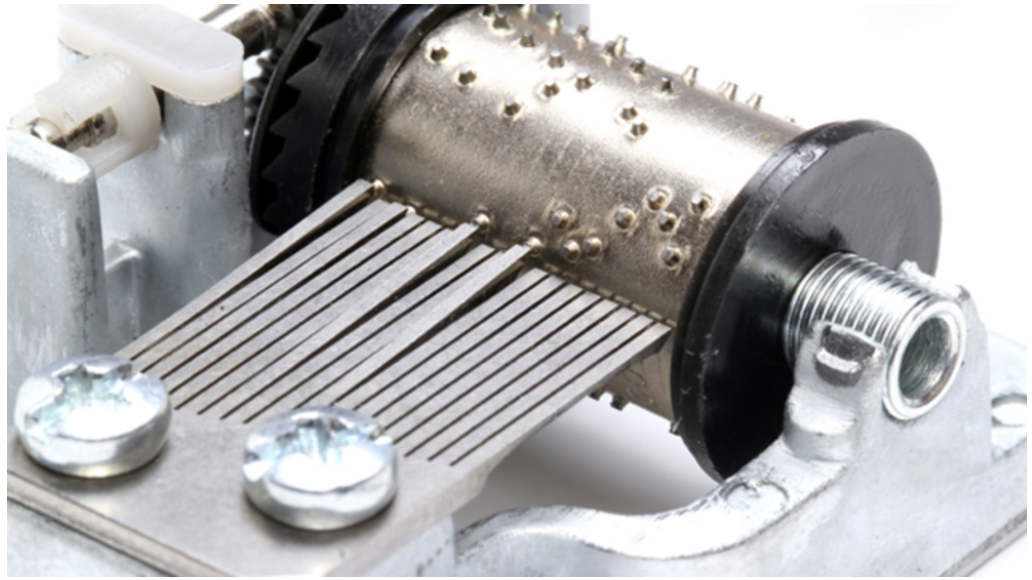
| | 26. | Jefferson | Monticello | Thomas | age | at | began | building | of | the |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 7 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 9 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |



One-hot vector sequence for the Monticello sentence

# Building your vocabulary with a tokenizer

- One-hot vectors are super-sparse, containing only one nonzero value in each row vector.
- But if you just want to see how this one-hot vector sequence is like a mechanical music box cylinder, or a player piano drum, the following listing can be a handy view of your data.



# Building your vocabulary with a tokenizer

```
import pandas as pd
df = pd.DataFrame(onehot_vectors, columns=vocab)
df[df==0] = ''
df
```

| | 26. | Jefferson | Monticello | Thomas | age | at | began | building | of | the |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | | | 1 | | | | | | |
| 1 | | 1 | | | | | | | | |
| 2 | | | | | | | 1 | | | |
| 3 | | | | | | | | 1 | | |
| 4 | | | 1 | | | | | | | |
| 5 | | | | | | 1 | | | | |
| 6 | | | | | | | | | | 1 |
| 7 | | | | | 1 | | | | | |
| 8 | | | | | | | | | 1 | |
| 9 | 1 | | | | | | | | | |

# Building your vocabulary with a tokenizer

- In this representation of your one-sentence document, each row is a vector for a single word.
- The sentence has 10 words, all unique, and it doesn't reuse any words.
- The table has 10 columns (words in your vocabulary) and 10 rows (words in the document).
- A "1" in a column indicates a vocabulary word that was present at that position in the document.
- So if you wanted to know what the third word in a document was, you'd go to the third row in the table.
- And you'd look up at the column heading for the "1" value in the third row (the row labeled 2, because the row numbers start at 0).
- At the top of that column, the seventh column in the table, you can find the natural language representation of that word, "began."

# Building your vocabulary with a tokenizer

- Each row of the table is a binary row vector, and you can see why it's also called a one-hot vector: all but one of the positions (columns) in a row are 0 or blank.
- Only one column, or position in the vector, is "hot" ("1"). A one (1) means on, or hot. A zero (0) means off, or absent.
- And you can use the vector [0, 0, 0, 0, 0, 0, 1, 0, 0, 0] to represent the word "began" in your NLP pipeline.

# Building your vocabulary with a tokenizer

- One nice feature of this vector representation of words and tabular representation of documents is that no information is lost.
- You can reconstruct the original document from this table of one-hot vectors. And this reconstruction process is 100% accurate.
- As a result, one-hot word vectors like this are typically used in:
  - neural nets,
  - sequence-to-sequence language models,
  - generative language models.
- They're a good choice for any model or NLP pipeline that needs to retain all the meaning inherent in the original text.

# Building your vocabulary with a tokenizer

- But this is a big table for a short sentence. For a long document this might not be practical. The document size (the length of the vector table) would grow to be huge.
- Let's assume you have a million tokens in your NLP pipeline vocabulary.
- And let's say you have 3,000 books with 3,500 sentences each and 15 words per sentence.
- 3000 * 3500 * 15 = 157,500,00 rows in out one-hot table
- 157500000 * 1000000 = 157.5 terabytes of data (assuming one byte per cell)
- The size of the original data was 3000 * 3500 * 15 * 7 = 1.1025 gigabytes (assuming the average length of a word to be 7 chars, with 1 byte per char)

# Building your vocabulary with a tokenizer

- So storing all those zeros, and trying to remember the order of the words in all your documents, doesn't make much sense.
- What if we allow to give up perfect "recall."; just capture most of the meaning (information) in a document, not all of it.
- Assume that most of the meaning of a sentence can be gleaned from just the words themselves, ignoring the order and grammar of the words, and collect them all up together into a "bag".
- If a human can still guess what the sentence was about, then so can a machine.

# Building your vocabulary with a tokenizer

- If you summed all these one-hot vectors together, rather than "replaying" them one at a time, you'd get a bag-of-words vector.
- This is also called a word frequency vector, because it only counts the *frequency* of words, not their order.
- You could use this single vector to represent the whole document or sentence in a single, reasonable-length vector.
- It would only be as long as your vocabulary size (the number of unique tokens you want to keep track of).

# Building your vocabulary with a tokenizer

- Here's what your single text document, the sentence about Thomas Jefferson, looks like as a binary bag-of-words vector:

```
token_sequence = "Thomas Jefferson began building Monticello at the age of
26.".split()
sentence_bow = {}
for token in token_sequence:
    sentence_bow[token] = 1
sorted(sentence_bow.items())
```

```
[('26.', 1), ('Jefferson', 1), ('Monticello', 1), ('The', 1), ('Thomas', 1)
, ('age', 1), ('at', 1), ('began', 1), ('building', 1), ('of', 1),('the', 1
)]
```

# Building your vocabulary with a tokenizer

- Let's use an even more efficient form of a dictionary, a Pandas Series, and wrap that up in a Pandas DataFrame so you can add more sentences to your binary vector "corpus" of texts about Thomas Jefferson.

```
pd.DataFrame(pd.Series(dict([(token, 1) for token in token_sequence])),
columns=['sent']).T
```

| | The | Thomas | Jefferson | began | building | Monticello | at | the | age | of | 26. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| sent | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# Building your vocabulary with a tokenizer

- With a quick scan, you can see little overlap in word usage for these sentences. Among
- the first seven words in your vocabulary, only the word “Monticello” appears in more
- than one sentence. Now you need to be able to compute this overlap within your pipeline
- whenever you want to compare documents or search for similar documents. One
- way to check for the similarities between sentences is to count the number of overlapping
- tokens using a *dot product*.

# Part 3: Stopword Removal

# What are stopwords?

- Stopwords are the most common words in any natural language.
- For the purpose of analyzing text data and building NLP models, these stopwords might not add much value to the meaning of the document.
- Generally, the most common words used in a text are “the”, “is”, “in”, “for”, “where”, “when”, “to”, “at” etc.

# What are stopwords?

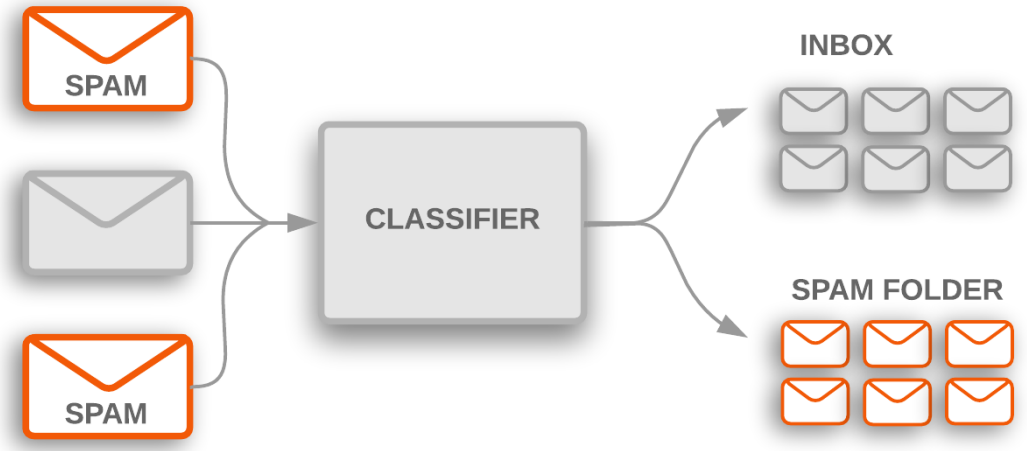
- Consider this text string – "There is a pen on the table".

- Now, the words "is", "a", "on", and "the" add no meaning to the statement while parsing it.
- Whereas words like "there", "book", and "table" are the keywords and tell us what the statement is all about.

- Now Consider the string – "There pen table"

# What are stopwords?

- Stop words are often removed from the text before training machine learning models since stop words occur in abundance, hence providing little to no unique information that can be used for classification or clustering.
- For tasks like text classification, where the text is to be classified into different categories, stopwords are removed or excluded from the given text so that more focus can be given to those words which define the meaning of the text.



# Removing Stopwords with NLTK

- NLTK, or the Natural Language Toolkit, is a python library for text preprocessing. It has a list of stopwords stored in 16 different languages.

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

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're"
, "you've", "you'll", "you'd", 'your', 'yours'…]
```

# Removing Stopwords with NLTK

- So to remove stopwords from a sentence, we can use the following code:

```
my_string = “there is a pen on the table”
my_string_no_stopwords = []
for s in my_string.split():
    if s not in stopwords.words('english'):
        my_string_no_stopwords.append(s)
print(my_string_no_stopwords)
```

```
['pen', 'table']
```

# Removing Stopwords with Spacy

- spaCy is one of the most versatile and widely used libraries in NLP.
- We can quickly and efficiently remove stopwords from the given text using SpaCy.
- It has a list of its own stopwords that can be imported as STOP_WORDS from the *spacy.lang.en.stop_words* class.

# Removing Stopwords with Spacy

- This process can be accomplished in 2 ways, either in a similar way to NLTK:

```
from spacy.lang.en.stop_words import STOP_WORDS
my_string = "there is a pen on the table"
my_string_no_stopwords = []
for s in my_string.split():
    if s not in STOP_WORDS:
        my_string_no_stopwords.append(s)
print(my_string_no_stopwords)
```

```
['pen', 'table']
```

- Or you can use some of spaCy's *nlp* functionalities

```
from spacy.lang.en import English
# Load English tokenizer, tagger, parser, NER and word vectors
nlp = English()
my_string = "there is a pen on the table"
#  "nlp" Object is used to create documents with linguistic annotations.
my_doc = nlp(my_string)
token_list = []
for token in my_doc:
    token_list.append(token.text)

from spacy.lang.en.stop_words import STOP_WORDS

# Create list of word tokens after removing stopwords
filtered_sentence =[]

for word in token_list:
    lexeme = nlp.vocab[word]
    if lexeme.is_stop == False:
        filtered_sentence.append(word)


print(filtered_sentence)
```

```
['pen', 'table']
```

# Removing Stopwords with Gensim

- Gensim is a pretty handy library to work with on NLP tasks.
- We can easily import the *remove_stopwords* method from the class *gensim.parsing.preprocessing*

```
from gensim.parsing.preprocessing import remove_stopwords
my_string = "there is a pen on the table"
filtered_sentence = remove_stopwords(my_string)
print(filtered_sentence)

['pen', 'table']
```

- And if you prefer, you can also do it in the same way done with NLTK, you can access the Gensim stopwords list from *gensim.parsing.preprocessing.STOPWORDS*

# Stopwords Removal is not Always a Good Idea

**Pros**

- Words are usually considered stopwords because they don't help us to find the context or the true meaning of a sentence.
- In addition, removing stopwords reduces the size of the data we have to process (vocabulary normalization), which improves performance.

**Cons**

- Depending on the NLP task at hand, and the type of model under training, removing stopwords might have a negative effect.
- For example, consider the sentiment of the 2 sentences: `"The boy was not happy"` and `"The boy happy"`

# Stopwords Removal is not Always a Good Idea

- So when to remove stopwords?
- We should remove these words only if they don't add any new information for the problem at hand;
  - Classification problems normally don't need stop words because it's possible to talk about the general idea of a text even if we remove stop words from it.
  - In addition, with new transformer models, the accuracy generally drops when stopwords are removed.

# Part 4: Vocabulary Normalization

# What is Vocabulary Normalization

- A vocabulary reduction technique is to normalize your vocabulary so that tokens that mean similar things are combined into a single, normalized form.
- Doing so accomplishes 2 things:

1. Reduces the number of tokens you need to retain in your vocabulary .
2. Improves the association of meaning across those different “spellings” of a token or ngram in your corpus.

# What is Vocabulary Normalization

- A vocabulary reduction technique is to normalize your vocabulary so that tokens that mean similar things are combined into a single, normalized form.
- Doing so accomplishes 2 things:

1. Reduces the number of tokens you need to retain in your vocabulary .
2. Improves the association of meaning across those different “spellings” of a token or ngram in your corpus.

# What is Vocabulary Normalization

- We'll talk about 3 techniques concerning vocabulary normalization:

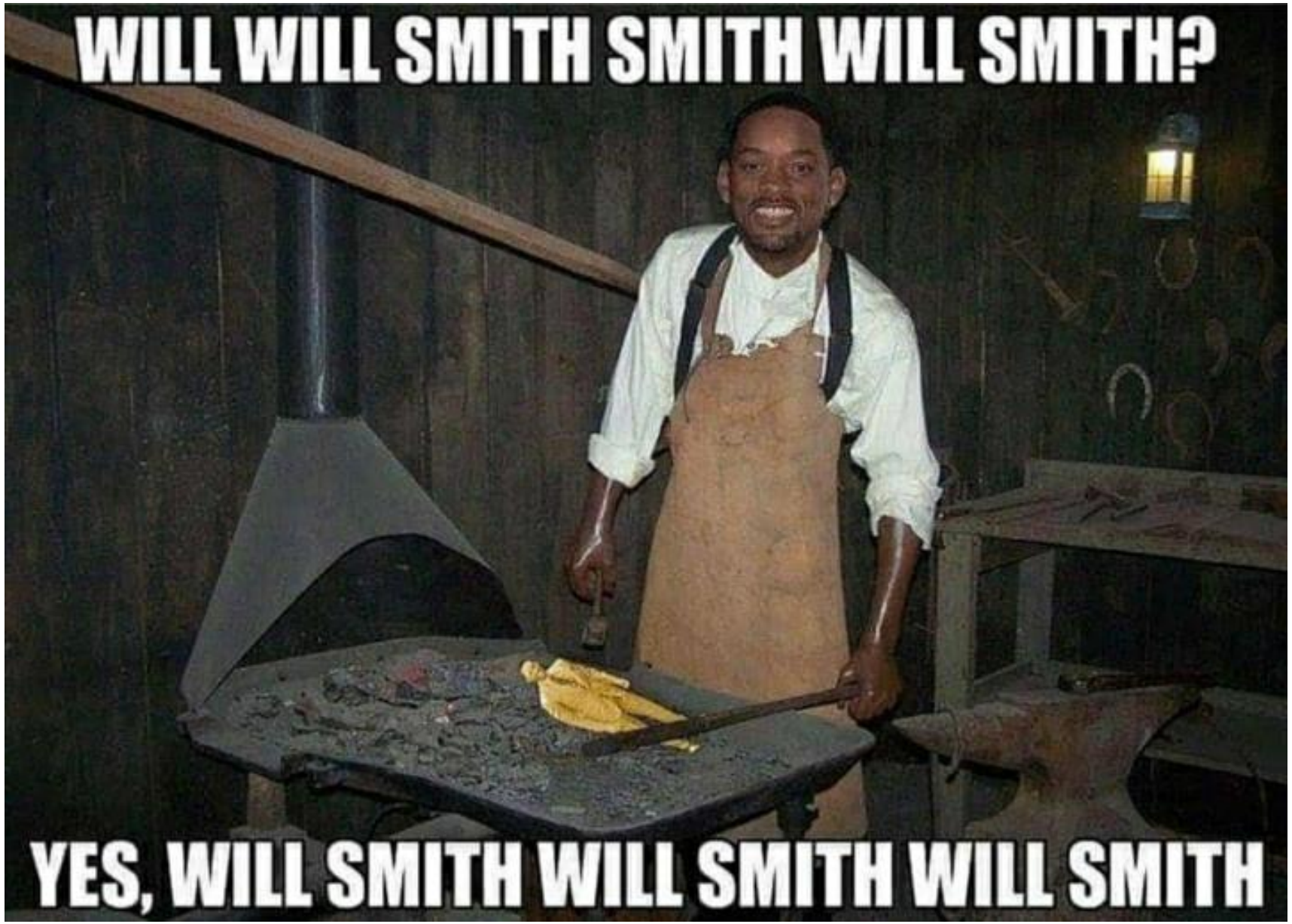
1. Case Folding (lowercasing)
2. Stemming
3. Lemmatization

# Case Folding

- Case folding is when you consolidate multiple "spellings" of a word that differ only in their capitalization.
- So why would we use case folding at all? Words can have different casing when they are capitalized because of their presence at the beginning of a sentence, or when they're written in ALL CAPS for emphasis.
- Undoing this denormalization is called case normalization, or more commonly, case folding.

# Case Folding

- However, some information is often communicated by capitalization of a word.
- Often capitalization is used to indicate that a word is a proper noun.
- You'll want to be able to recognize proper nouns as distinct from other words, if named entity recognition is important to your pipeline.


WILL WILL SMITH SMITH WILL SMITH?
YES, WILL SMITH WILL SMITH WILL SMITH

# Case Folding

In Python, you can easily normalize the capitalization of your tokens with a list comprehension

```
tokens = ['House', 'Visitor', 'Center']
normalized_tokens = [x.lower() for x in tokens]
print(normalized_tokens)
```

```
['house', 'visitor', 'center']
```

By generalizing your model to work with text that has odd capitalization, **case normalization can reduce overfitting** for your machine learning pipeline.

# Case Folding

- Case normalization is particularly useful for a search engine;
- For search, normalization increases the number of matches found for a particular query.
- For a search engine without normalization, if you searched for “Age” you would get a different set of documents than if you searched for “age”.
- However, this additional accuracy comes at the cost of precision, returning many documents that the user may not be interested in.

# Stemming

- Another common vocabulary normalization technique is to eliminate the small meaning differences of pluralization or possessive endings of words, or even various verb forms, **Stemming**.
- For example, the words *housing* and *houses* share the same stem, *hous*.
- Stemming removes suffixes from words in an attempt to combine words with similar meanings together under their common stem.
- This stemming process reduces the size of our vocabulary while limiting the loss of information and meaning, as much as possible.
- This enables the model in training to behave identically for all the words included in a stem.

# Stemming

- Stemming is important for keyword search or information retrieval.

- It allows you to search for “developing houses in Portland” and get web pages or documents that use both the word “house” and “houses” and even the word “housing,” because these words are all stemmed to the “hous” token.

# Stemming

- We'll checkout 3 NLTK stemming algorithms, which are *Snowball Stemmer*, *Porter Stemmer* and *Lancaster Stemmer*
  - Porter Stemmer being the oldest one originally developed in 1979
  - Snowball stemmer is a slightly improved version of the Porter stemmer and is usually preferred over the latter
  - Lancaster Stemmer was developed in 1990 and uses a more aggressive approach than Porter Stemming Algorithm.

# Stemming

```
from nltk.stem import PorterStemmer
from nltk.stem import LancasterStemmer
from nltk.stem.snowball import SnowballStemmer

#initialize stemmers
porter = PorterStemmer()
lancaster=LancasterStemmer()
snowball = SnowballStemmer(language='english')

word_list =
["house","houses","housing","playful","playing","naming","friend",
"friendship", "friends", "friendships","aren't"]
print("{0:20}{1:20}{2:20}{3:20}".format("Word","Porter Stemmer","Lancaster
Stemmer","Snowball Stemmer"))
for word in word_list:
    print("{0:20}{1:20}{2:20}{3:20}".format(word,porter.stem(word),
lancaster.stem(word),snowball.stem(word)))
```

# Stemming

Output

| Word | Porter Stemmer | Lancaster Stemmer | Snowball Stemmer |
|---|---|---|---|
| house | hous | hous | hous |
| houses | hous | hous | hous |
| housing | hous | hous | hous |
| playful | play | play | play |
| playing | play | play | play |
| naming | name | nam | name |
| friend | friend | friend | friend |
| friendship | friendship | friend | friendship |
| friends | friend | friend | friend |
| friendships | friendship | friend | friendship |
| aren't | aren't | aren't | aren't |

# Lemmatization

- Lemmatization is the association between several words together having similar meaning even if their spelling is quite different.
- Example: runs, running, ran are all forms of the word run, therefore run is the lemma of all these words.
- Using lemmatization can make a model more general, but it can also make it less precise, because it will treat all spelling variations of a given root word the same.

# Lemmatization

- For example, “bank,” “banked,” and “banking” would all be treated the same in an NLP pipeline with lemmatization, despite the river meaning of “bank,” the raceway meaning of “banked,” and the finance meaning of “banking.”


# Lemmatization

- Lemmatization is a potentially more accurate way to normalize a word than stemming or case normalization because it takes into account a word's meaning.
- A lemmatizer uses a knowledge base of word synonyms and word endings to ensure that only words that mean similar things are consolidated into a single token.
- Some lemmatizers use the word's part of speech (POS) tag in addition to its spelling to help improve accuracy.

# Lemmatization

- Lemmatizers are better than stemmers for most applications.
- Stemmers are only really used in large-scale information retrieval applications (keyword search).
- Because the lemma of a word is a valid language word, stemmers work well on the output of a lemmatizer.
- This trick will reduce our dimensionality and increase our information retrieval recall even more than a stemmer alone.

- Python NLTK provides WordNet Lemmatizer that uses the WordNet Database to lookup lemmas of words.

```
import nltk
# download the wordnet corpus
nltk.download('wordnet')

from nltk.stem import WordNetLemmatizer
wordnet_lemmatizer = WordNetLemmatizer()

sentence = "An infestation is only realized when it's too late."
punctuations="?:!.,;"
sentence_words = nltk.word_tokenize(sentence)
for word in sentence_words:
    if word in punctuations:
        sentence_words.remove(word)

print("{0:20}{1:20}".format("Word","Lemma"))
for word in sentence_words:
    print ("{0:20}{1:20}".format(word,wordnet_lemmatizer.lemmatize(word)))
```

- Output

| Word | Lemma |
|---|---|
| An | An |
| infestation | infestation |
| is | is |
| only | only |
| realized | realized |
| when | when |
| it | it |
| 's | 's |
| too | too |
| late | late |

- As we can notice, no root form has been given for any word, this is because they are given without context.
- We need to provide the context in which we want to lemmatize; that is the parts-of-speech (POS).
- This is done by giving the value for pos parameter in *wordnet_lemmatizer.lemmatize.*

# Lemmatization

```
print("{0:20}{1:20}".format("Word","Lemma"))
for word in sentence_words:
    print ("{0:20}{1:20}".format(word,wordnet_lemmatizer.lemmatize(word,
pos="v")))
```

| Word | Lemma |
|---|---|
| An | An |
| infestation | infestation |
| is | be |
| only | only |
| realized | realize |
| when | when |
| it | it |
| 's | 's |
| too | too |
| late | late |

# Lemmatizers or Stemmers?

- Stemmers are generally faster to compute and require less-complex code and datasets.
- But stemmers will make more errors and stem a far greater number of words.
- Both stemmers and lemmatizers will reduce your vocabulary size and increase the ambiguity of the text. But lemmatizers do a better job retaining as much of the information content as possible based on how the word was used within the text and its intended meaning.

# Lemmatizers or Stemmers?

- If your application involves search, stemming and lemmatization will improve the recall of your searches by associating more documents with the same query words. However, stemming, lemmatization, and even case folding will significantly reduce the precision and accuracy of your search results.
- This is different, however, to a chatbot application, where accuracy is more important.
- As a result, a chatbot should first search for the closest match using unstemmed, unnormalized words before falling back to stemmed or filtered token matches to find matches.

# Lemmatizers or Stemmers?

- Bottom line, try to avoid stemming and lemmatization.
- The Stanford information retrieval course dismisses stemming and lemmatization entirely, due to the negligible recall accuracy improvement and the significant reduction in precision.