Technical Session I

Introduction to Natural Language Processing

Dr. Ambika P
ambika2202@gmail.com
www.linkedin.com/in/drambika
+91 9590399036

Agenda

- What is NLP?
- NLP used for
- NLP Capabilities
- Components(NLG and NLU)
- History of NLP
- NLP Definitions
- Levels of NLP
- Applications of NLP
- NLP in Simple form
- Feature extraction and methods
- Pre-processing
- Similarity Measures

What is NLP?

- Branch of AI that deals with the interaction between computers and humans using the natural language.
- Objective - To read, decipher, understand, and make sense of the human languages in a manner that is valuable.
- NLP techniques rely on machine learning to derive meaning from human languages

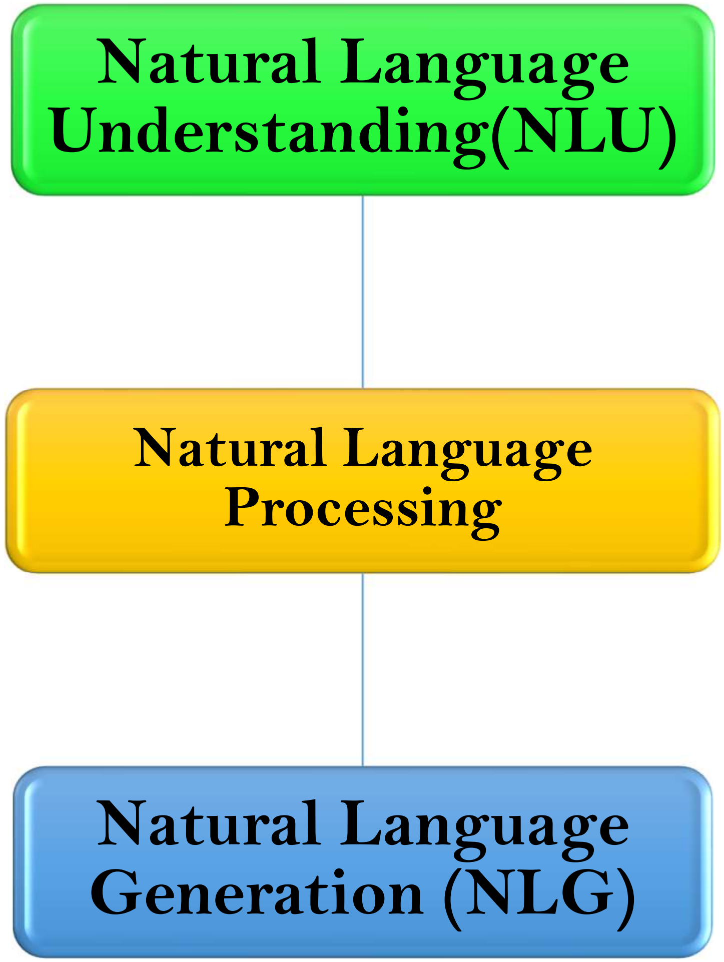
NLP used for

Google Translate	• **Language translation applications**
Word Processors	• **Microsoft Word and Grammarly**
Interactive Voice Response	• **Call centers to respond to certain users' requests.**
Personal assistant applications	• **OK Google, Siri, Cortana, and Alexa.**

Higher-level NLP capabilities

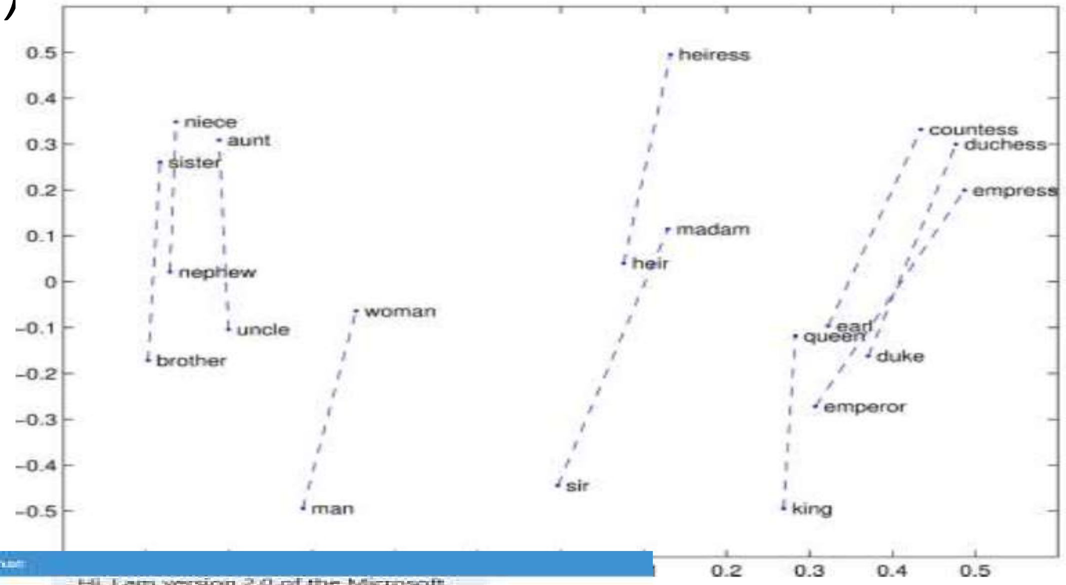
Content categorization	• A linguistic-based document summary, including search and indexing, content alerts and duplication detection.
Topic discovery and modeling.	• Accurately capture the meaning and themes in text collections, and apply advanced analytics to text, like optimization and forecasting
Contextual extraction.	• Automatically pull structured information from text-based sources
Sentiment analysis.	• Identifying the mood or subjective opinions within large amounts of text, including average sentiment and opinion mining
Speech-to-text and text-to-speech conversion.	• Transforming voice commands into written text, and vice versa.
Document summarization.	• Automatically generating synopses of large bodies of text
Machine translation.	• Automatic translation of text or speech from one language to another.

Two main Components


Natural Language Understanding(NLU)
Natural Language Processing
Natural Language Generation (NLG)

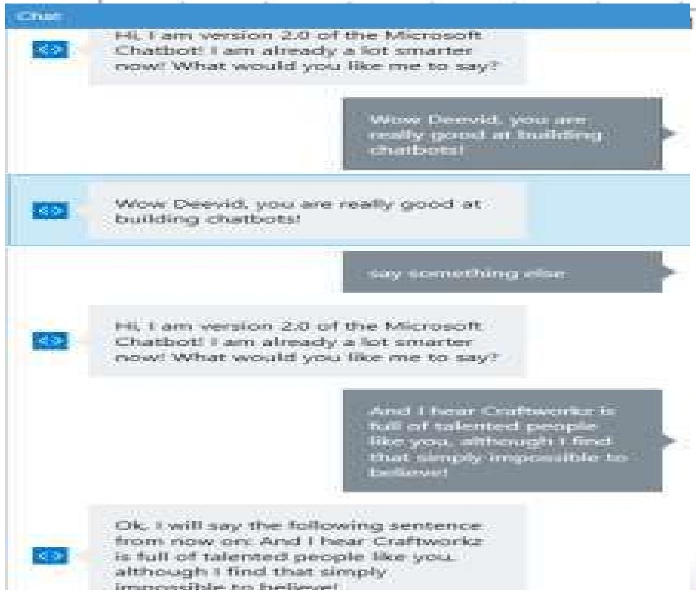
Natural Language Understanding(NLU)

- Deriving meaning from natural language
- Imagine a Concept (Semantic or Representation) space
- Any idea/word/concept has unique
- Computer representation
- Usually via a vector space
- NLU - Mapping language into this space



Natural Language Generation(NLG)

- Mapping from computer representation space to language space
- Opposite direction of NLU
- Usually need NLU to perform NLG!
- NLG is really hard!



Hi, I am version 2.0 of the Microsoft Chatbot! I am already a lot smarter now! What would you like me to say?

Wow Deevid, you are really good at building chatbots!

Wow Deevid, you are really good at building chatbots!

say something else

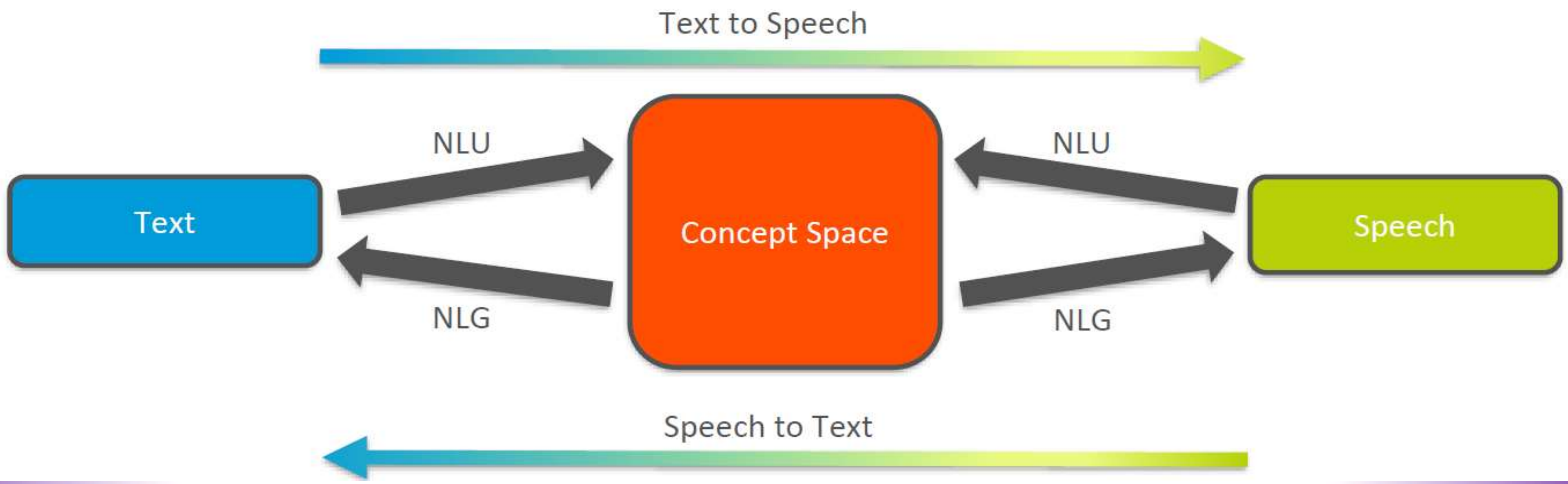
Hi, I am version 2.0 of the Microsoft Chatbot! I am already a lot smarter now! What would you like me to say?

And I hear Craftworkz is full of talented people like you, although I find that simply impossible to believe!

Ok, I will say the following sentence from now on: And I hear Craftworkz is full of talented people like you, although I find that simply impossible to believe!

NLP: Speech vs Text

- Natural Language can refer to Text or Speech
- Goal of both is the same:
- translate raw data (text or speech) into underlying concepts (NLU) then possibly into the other form (NLG)



History of NLP

- NLP has been through (at least) 3 major eras:
- 1950s-1980s: Linguistics Methods and Handwritten Rules
- 1980s-Now: Corpus/Statistical Methods
- Now-???: Deep Learning

NLP Definitions

Phonemes

- The smallest sound units in a language

Morphemes

- The smallest units of meaning in a language

Syntax

- How words and sentences are constructed from these two buildingblocks

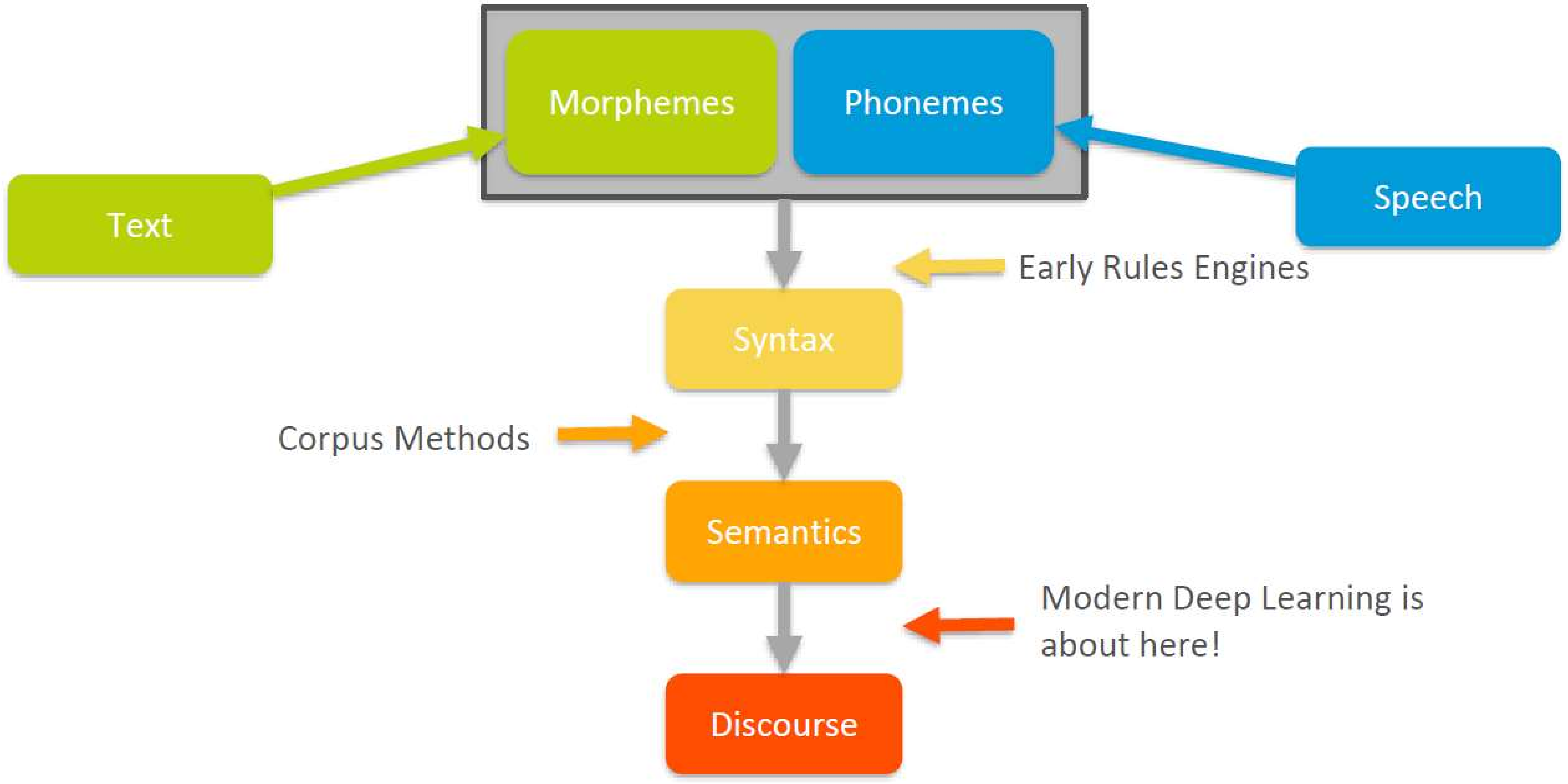
Semantics

- The meaning of those words and sentences

Discourse

- Semantics in context. Conversation, persuasive writing, etc.

Levels of NLP



NLU Applications

- ML on Text (Classification, Regression, Clustering)
- Document Recommendation
- Language Identification
- Natural Language Search
- Sentiment Analysis
- Text Summarization
- Extracting Word/Document Meaning (vectors)
- Relationship Extraction
- Topic Modeling …and more!

NLG Applications

- Image Captioning
- (Better) Text Summarization
- Machine Translation
- Question Answering/Chatbots
- …so much more
- Notice NLU is almost a prerequisite for NLG

NLP in simple example

What is a text ??

- ***Set of words sequentially written.***
- Each word in the text has a meaning where the text may or may not have a meaning.
- In machine leaning we take features right? so here each word is a feature(unique).

Example

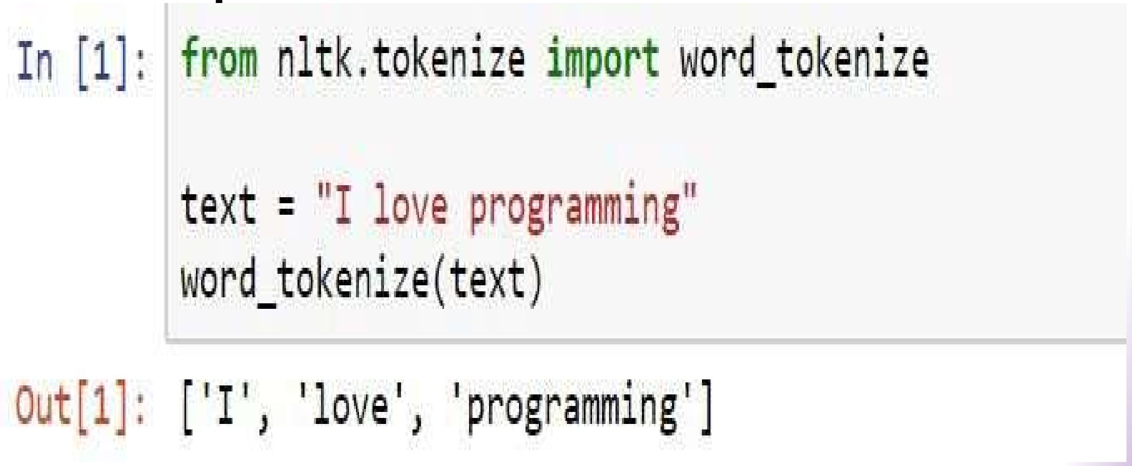
- Text : **I love programming →**
- **I** , **love, programming** are the features for this input.

How do we derive the features??

Tokenization

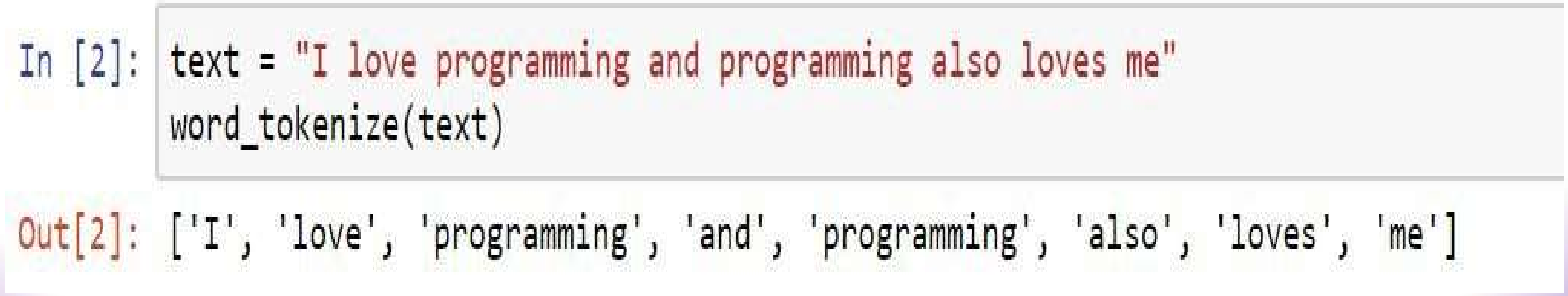
- A text is divided into token
- Open source tools like NLTK helps to get tokens from the text.

Example



```
In [1]: from nltk.tokenize import word_tokenize

        text = "I love programming"
        word_tokenize(text)

Out[1]: ['I', 'love', 'programming']
```

Lets say we have an example like below

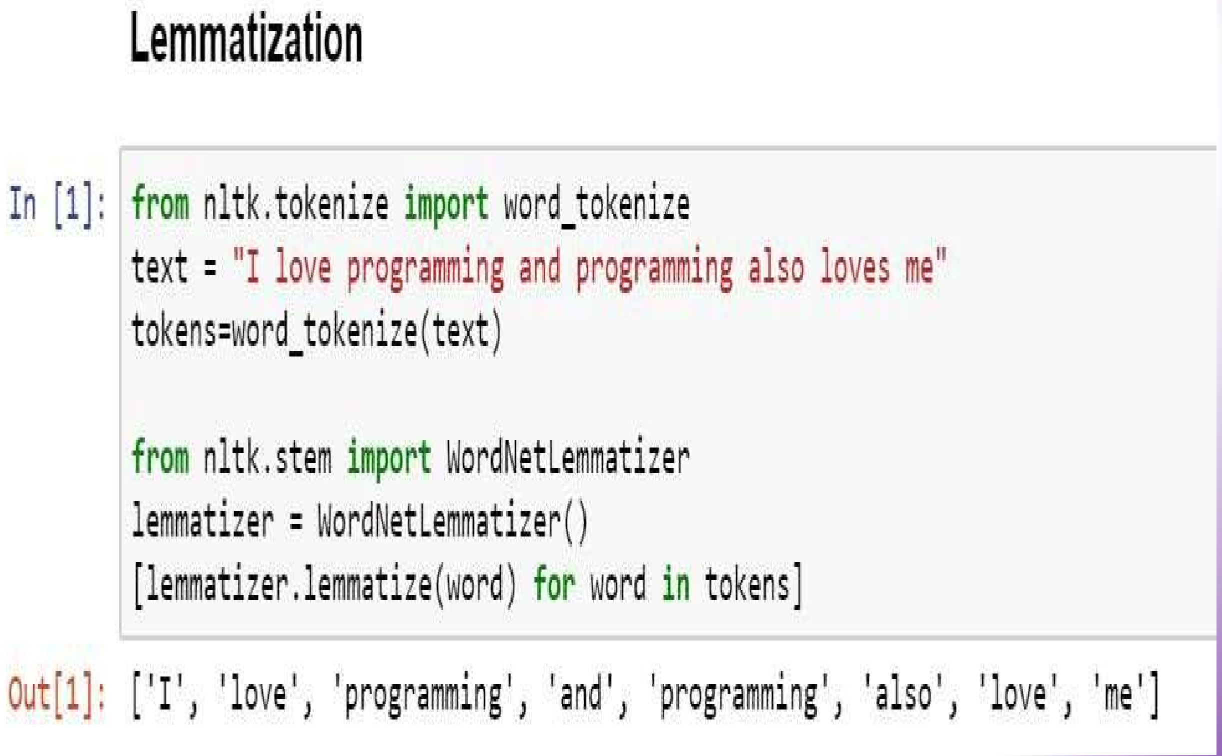


```
In [2]: text = "I love programming and programming also loves me"
        word_tokenize(text)

Out[2]: ['I', 'love', 'programming', 'and', 'programming', 'also', 'loves', 'me']
```

Lemmatization

- ***Programming*** repeated twice as tokens
- but we only take once so the features for this text are → ***I*** , ***love***, ***programming***, ***and***, ***also***, ***loves, me***.
- the words ***love*** and ***loves*** mean same , these are called **inflectional forms.** we need to remove these
- removing these inflectional endings is called **lemmatization**

Example



Lemmatization

```
In [1]: from nltk.tokenize import word_tokenize
        text = "I love programming and programming also loves me"
        tokens=word_tokenize(text)

        from nltk.stem import WordNetLemmatizer
        lemmatizer = WordNetLemmatizer()
        [lemmatizer.lemmatize(word) for word in tokens]

Out[1]: ['I', 'love', 'programming', 'and', 'programming', 'also', 'love', 'me']
```

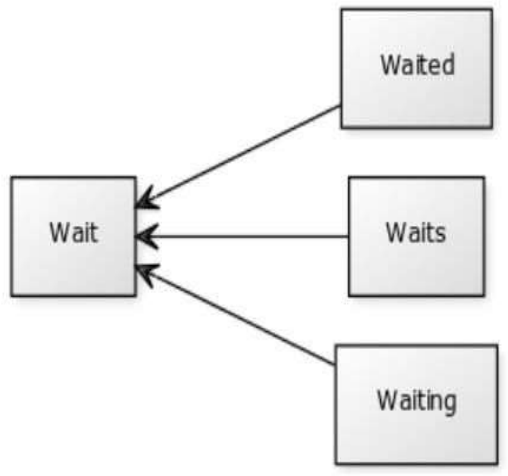
so now the features for this text are
→ ***I*** , ***love***, ***programming***, ***and***, ***also***, ***me***.

Stemming

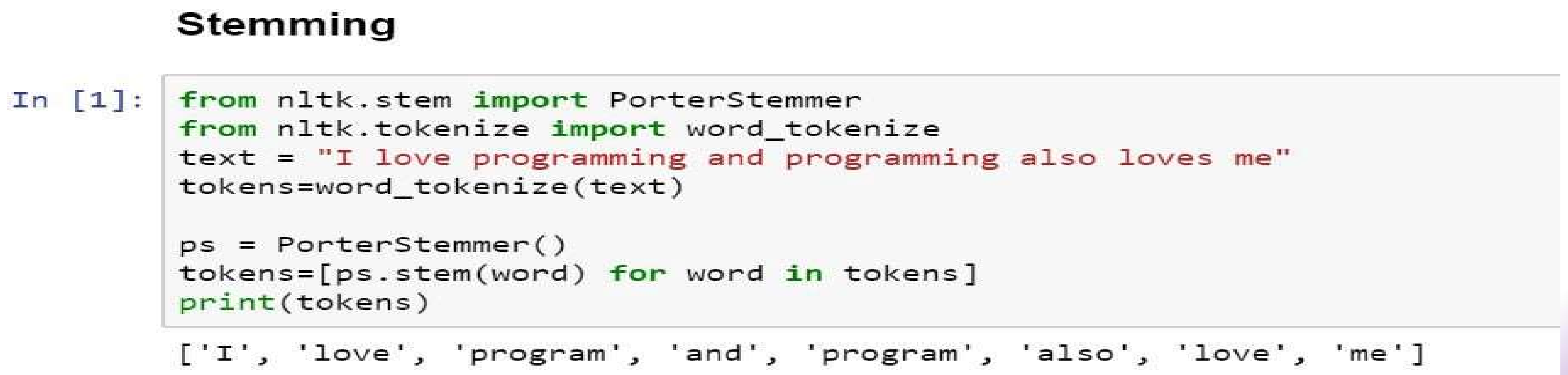
- The word **programming** is similar to the word **program**
- ***Finally features are I , love, program, and, also,me***

A word stem is part of a word. It is sort of a normalization idea, but linguistic.

For example, the stem of the word waiting is wait.



word stem

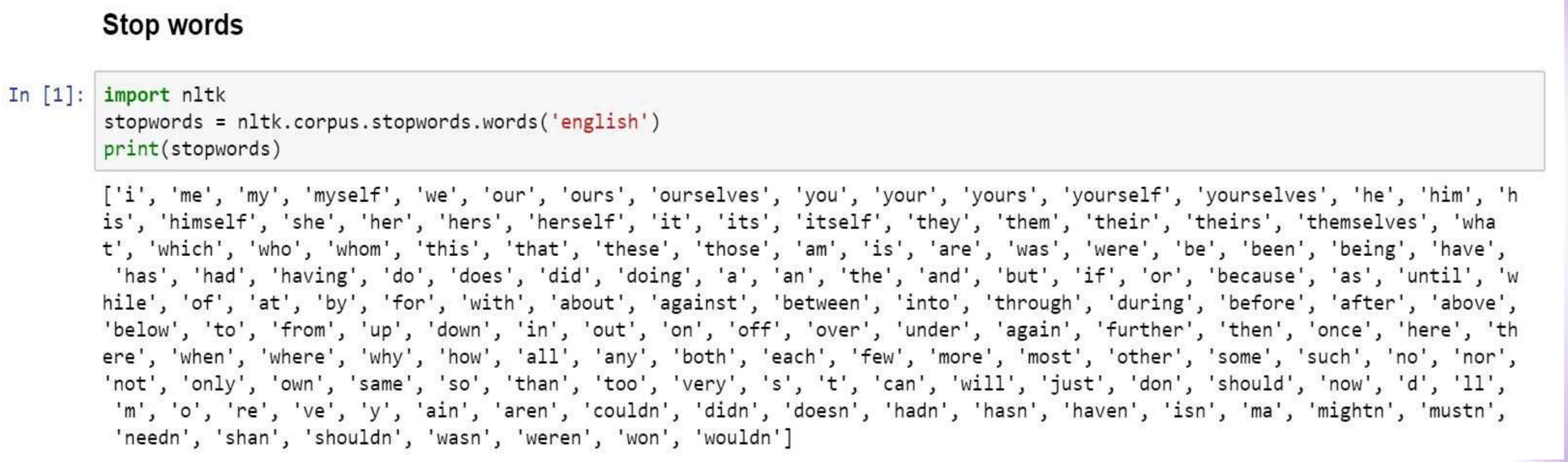


Stemming

```
In [1]: from nltk.stem import PorterStemmer
        from nltk.tokenize import word_tokenize
        text = "I love programming and programming also loves me"
        tokens=word_tokenize(text)

        ps = PorterStemmer()
        tokens=[ps.stem(word) for word in tokens]
        print(tokens)

        ['I', 'love', 'program', 'and', 'program', 'also', 'love', 'me']
```

Stop words

Example stop words

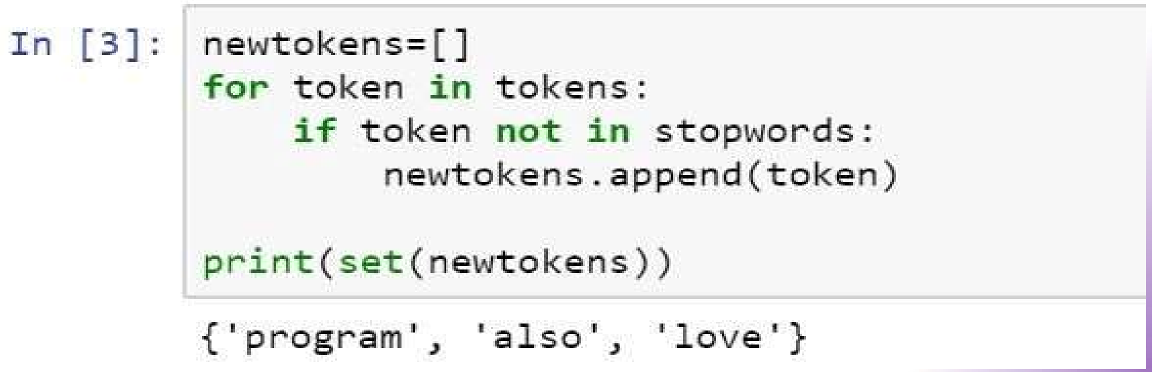


Stop words

```
In [1]: import nltk
        stopwords = nltk.corpus.stopwords.words('english')
        print(stopwords)
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', 'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'h
is', 'himself', 'she', 'her', 'hers', 'herself', 'it', 'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'wha
t', 'which', 'who', 'whom', 'this', 'that', 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have',
 'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'w
hile', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into', 'through', 'during', 'before', 'after', 'above',
'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then', 'once', 'here', 'th
ere', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more', 'most', 'other', 'some', 'such', 'no', 'nor',
'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's', 't', 'can', 'will', 'just', 'don', 'should', 'now', 'd', 'll',
 'm', 'o', 're', 've', 'y', 'ain', 'aren', 'couldn', 'didn', 'doesn', 'hadn', 'hasn', 'haven', 'isn', 'ma', 'mightn', 'mustn',
 'needn', 'shan', 'shouldn', 'wasn', 'weren', 'won', 'wouldn']
```

- There are couple of words which occur ve frequently in every language and don't hav much meaning , these words are called **Sto words.**
- These stop words need to be removed
- Note: we covert the text into lower case before tokenization to avoid duplicates.



```
In [3]: newtokens=[]
        for token in tokens:
            if token not in stopwords:
                newtokens.append(token)

        print(set(newtokens))
```

```
{'program', 'also', 'love'}
```

Example

- dataset which has 2 training examples

1 → I love programming
2 → Programming also loves me

- Step 1: After normalization, lemmatization, and stopwords,
- **Final Features : love**, **programming**, **also**
- **Final set = dictionary** or a **lexicon**
- we can not feed the words to the model / ml algorithm, The features values must be numbers.
- we take the count for every word in every document as a value.

Document Vectorization

- "I love programming" or "love programming" [After changing] , we feed [1 1 0] as a vector
- Process called as **document vectorization.**

Example

- X

x1	x2	x3	
1	1	0	---- 1
1	1	1	---- 2

Tools required

- Anaconda Navigator
- Python IDE (Spyder)

NLP Toolkits

NLTK (Natural Language Toolkit)

- most popular NLP library

TextBlob

- Wraps around NLTK and makes it easier to use

spaCy

- Built on Cython, so it's fast and powerful

gensim

- Great for topic modeling and document similarity

Regular Expressions

- Regular expression is a sequence of character(s) mainly used to find and replace patterns in a string or file
- It use two types of characters
- Meta characters: As the name suggests, these characters have a special meaning, similar to * in wild card. **[] . ^ $ * + ? {} () \ |**
- Literals (like a,b,1,2…)
- Module that helps to do regular expressions – "**re**"

<u>Common uses of regular expressions</u>

- Search a string (search and match)
- Finding a string (findall)
- Break string into a sub strings (split)
- Replace part of a string (sub)

Methods of Regular Expressions

- re.match()
- re.search()
- re.findall()
- re.split()
- re.sub()
- re.compile()

re.match(*pattern, string*):

- This method finds match if it occurs at start of the string.
- For example, calling match() on the string 'AV Analytics AV' and looking for a pattern 'AV' will match.
- However, if we look for only Analytics, the pattern will not match

re.search(*pattern, string*):

- It is similar to match() but it doesn't restrict us to find matches at the beginning of the string only.
- Unlike previous method, here searching for pattern 'Analytics' will return a match.
- search() method is able to find a pattern from **any position of the string** but it only **returns the first occurrence of the search pattern**

re.findall (*pattern, string*):

- It helps to get a list of all matching patterns.
- It has no constraints of searching from start or end.
- If we will use method findall to search 'AV' in given string it will return both occurrence of AV.
- While searching a string **re.findall() is better and** it works like re.search() and re.match() both.

re.split(*pattern, string*, [*maxsplit=0*]):

- This methods helps to split *string* by the occurrences of given *pattern.*
- argument "**maxsplit**". It has default value of zero.
- In this case it does the maximum splits that can be done, but if we give value to maxsplit, it will split the string.

re.compile(***pattern*, *repl*, *string***):

- We can combine a regular expression pattern into pattern objects, which can be used for pattern matching.
- It also helps to search a pattern again without rewriting it.

Operators

- Till now, we looked at various methods of regular expression using a constant pattern (fixed characters).
- But, what if we do not have a constant search pattern and we want to return specific set of characters from a string?

Answer : pattern operators

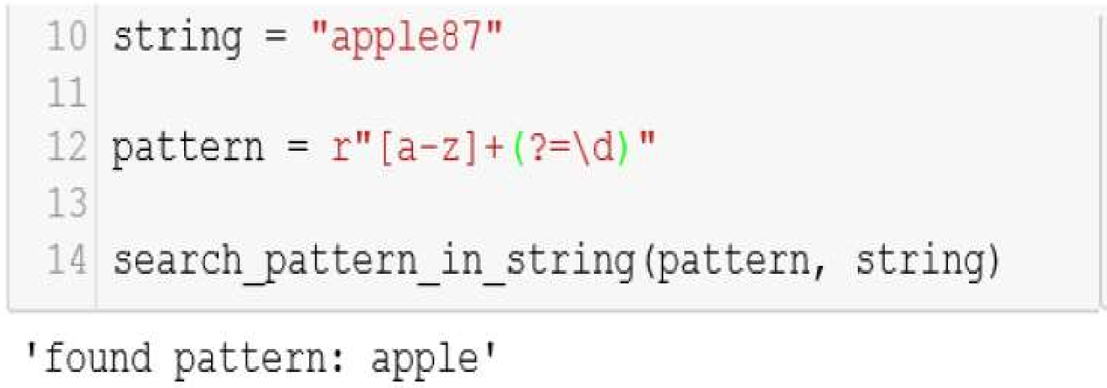
- Pattern operators -meta and literal characters
- It is commonly used in **web scrapping and text mining** to extract required information.

Operators

Operators	Description
.	Matches with any single character except newline '\n'.
?	match 0 or 1 occurrence of the pattern to its left
+	1 or more occurrences of the pattern to its left
*	0 or more occurrences of the pattern to its left
\w	Matches with a alphanumeric character whereas \W (upper case W) matches non alphanumeric character.
\d	Matches with digits [0-9] and /D (upper case D) matches with non-digits.
\s	Matches with a single white space character (space, newline, return, tab, form) and \S (upper case S) matches any non-white space character.
\b	boundary between word and non-word and /B is opposite of /b
[..]	Matches any single character in a square bracket and [^..] matches any single character not in square bracket
\	It is used for special meaning characters like \. to match a period or \+ for plus sign.
^ and $	^ and $ match the start or end of the string respectively
{n,m}	Matches at least n and at most m occurrences of preceding expression if we write it as {,m} then it will return at least any minimum occurrence to max m preceding expression.
a | b	Matches either a or b
( )	Groups regular expressions and returns matched text
\t, \n, \r	Matches tab, newline, return

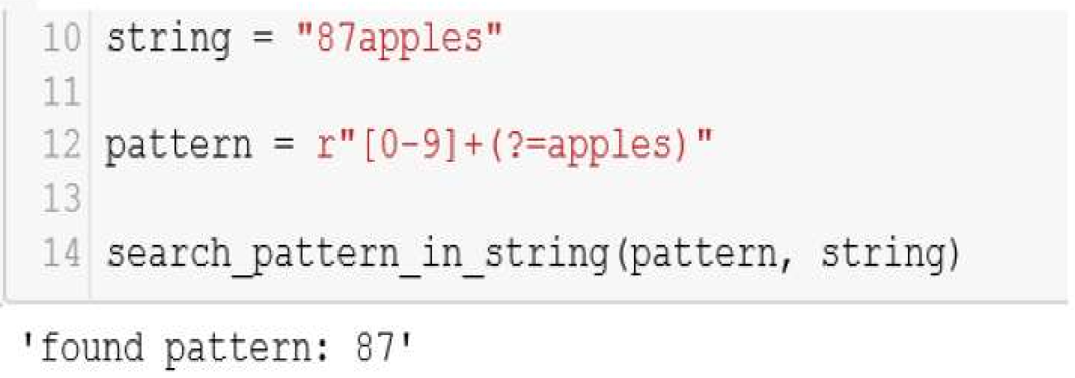
Lookaheads/Lookbehinds

- Allow you to keep the cursor in place, but still check to see if certain conditions are met before or after that character
- Look ahead: (?=\d) checks to see if the character is followed by a number
- Look Behind: (?<=[aeiou]) checks to see if the character is preceded by a vowel
- Example



```
string = "apple87"

pattern = r"[a-z]+(?=\d)"

search_pattern_in_string(pattern, string)
```

```
'found pattern: apple'
```



```
string = "87apples"

pattern = r"[0-9]+(?=apples)"

search_pattern_in_string(pattern, string)
```

```
'found pattern: 87'
```

Pre-processing Techniques

1. **Tokenization** - Turn text into a meaningful format for analysis
2. **Clean the data**
    - Remove: capital letters, punctuation, numbers, stop words
    - Stemming
    - Parts of speech tagging
    - Correct misspellings
    - Chunking (named entity recognition, compound term extraction)
    - Package needed – nltk

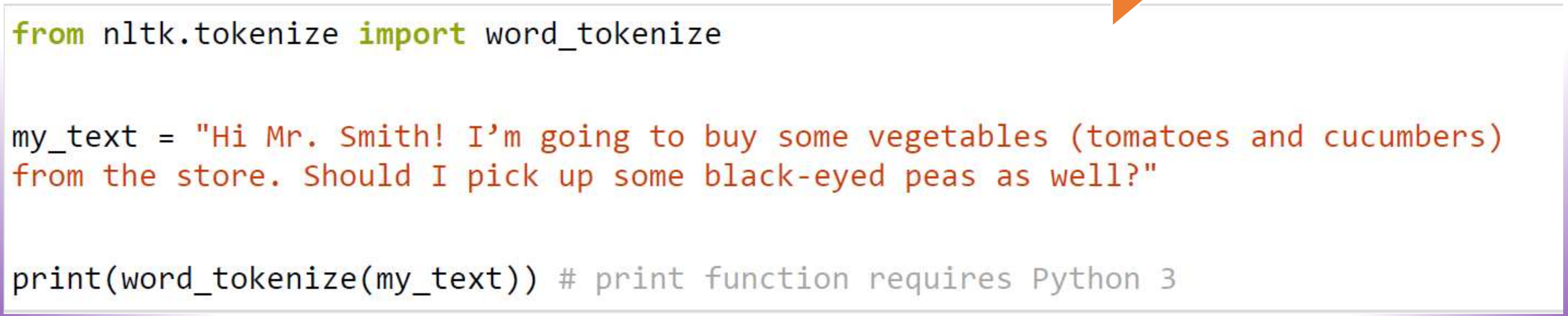
- Anaconda Prompt : To install C:\Users\Ambika\.anaconda\navigator\scripts
- > import nltk
- >nltk.download()

Tokenization

Tokenization = splitting raw text into small, indivisible units for processing

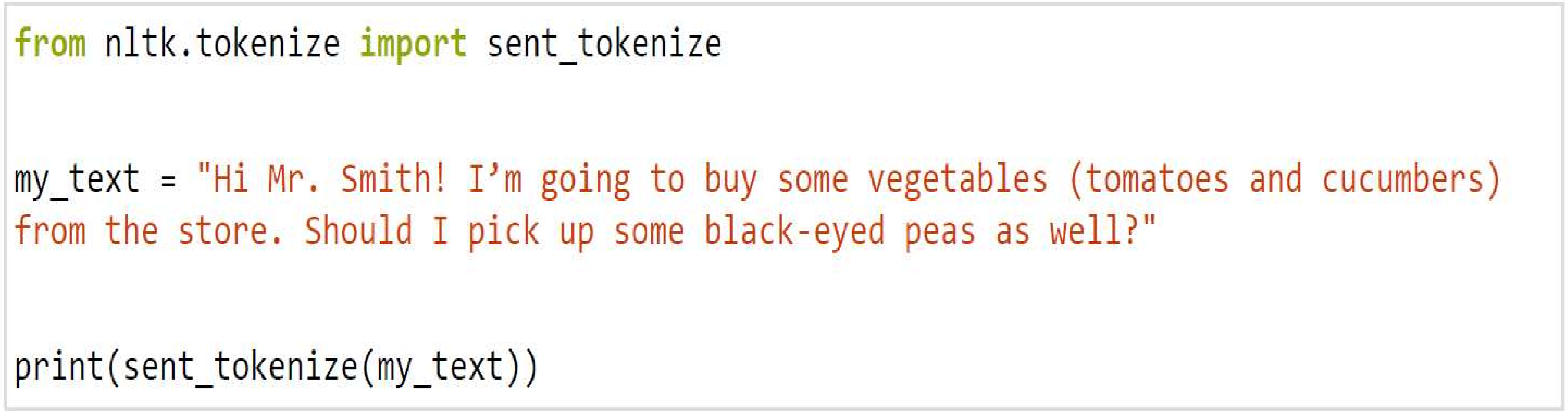
These units can be:

- **Words**
- **Sentences**
- **N-grams**
- **Other characters defined by regular expressions**



```
from nltk.tokenize import word_tokenize

my_text = "Hi Mr. Smith! I’m going to buy some vegetables (tomatoes and cucumbers)
from the store. Should I pick up some black-eyed peas as well?"

print(word_tokenize(my_text)) # print function requires Python 3
```

Sentence Tokenization



```
from nltk.tokenize import sent_tokenize


my_text = "Hi Mr. Smith! I’m going to buy some vegetables (tomatoes and cucumbers)
from the store. Should I pick up some black-eyed peas as well?"


print(sent_tokenize(my_text))
```

Tokenization - N Grams

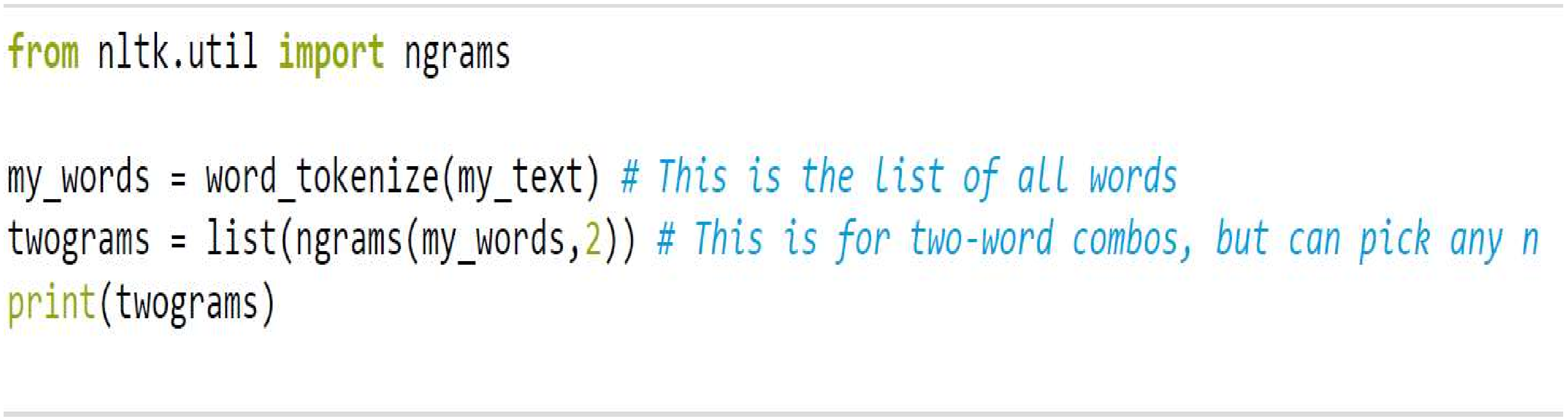
- an N-gram is simply a sequence of N words

San Francisco (is a 2-gram)

The Three Musketeers (is a 3-gram)

She stood up slowly (is a 4-gram)

Tokenization – Ngram example

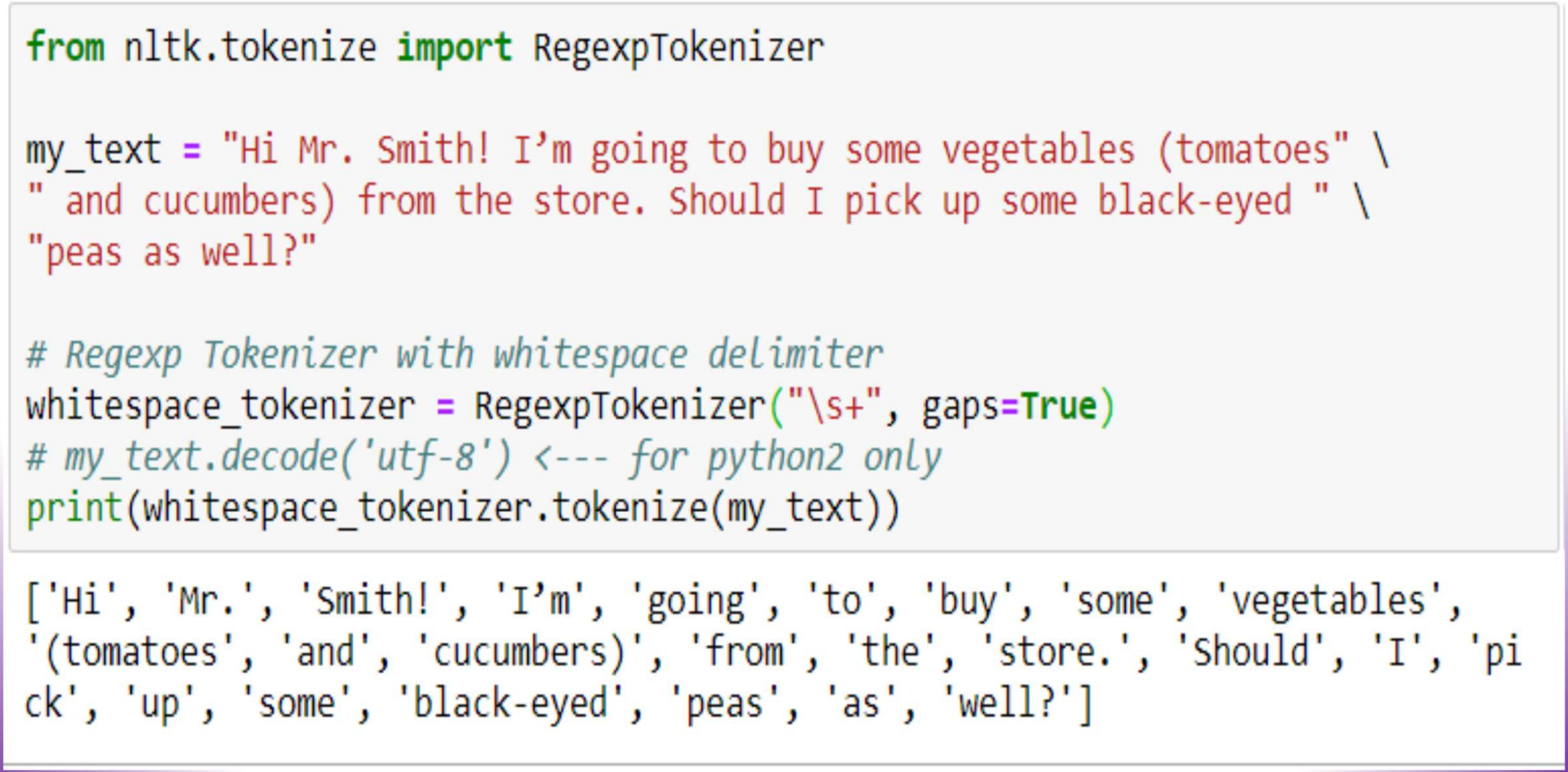


```python
from nltk.util import ngrams

my_words = word_tokenize(my_text) # This is the list of all words
twograms = list(ngrams(my_words,2)) # This is for two-word combos, but can pick any n
print(twograms)
```

Tokenization – Regular Expression

- If anubo want to tokenize by some other type of grouping or pattern.
- Regular expressions (regex) allows you to do so.
- Some examples of regular expressions:
- Find white spaces: \s+
- Find words starting with capital letters: [A-Z]['\w]+

Tokenization – Regular expression



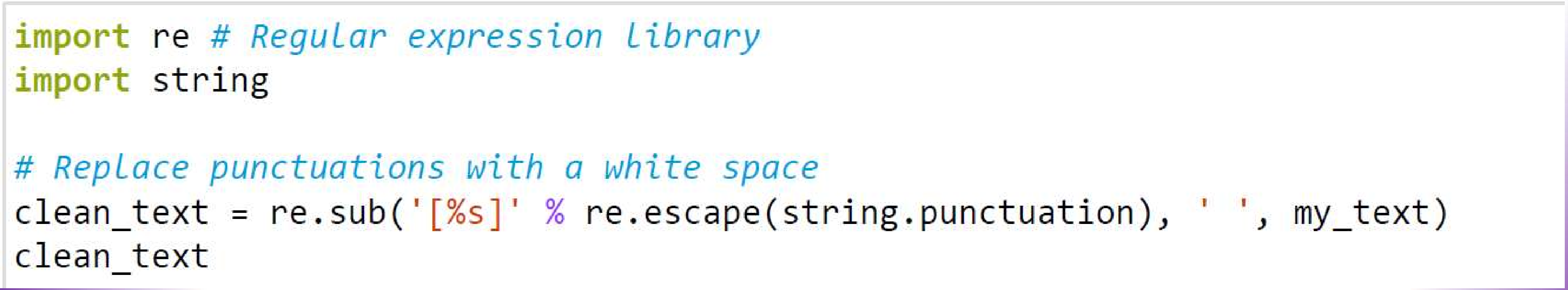
```
from nltk.tokenize import RegexpTokenizer

my_text = "Hi Mr. Smith! I’m going to buy some vegetables (tomatoes" \
" and cucumbers) from the store. Should I pick up some black-eyed " \
"peas as well?"

# Regexp Tokenizer with whitespace delimiter
whitespace_tokenizer = RegexpTokenizer("\s+", gaps=True)
# my_text.decode('utf-8') <--- for python2 only
print(whitespace_tokenizer.tokenize(my_text))
```

```
['Hi', 'Mr.', 'Smith!', 'I’m', 'going', 'to', 'buy', 'some', 'vegetables',
'(tomatoes', 'and', 'cucumbers)', 'from', 'the', 'store.', 'Should', 'I', 'pi
ck', 'up', 'some', 'black-eyed', 'peas', 'as', 'well?']
```

Data Clean

- Normalize the text – removal of punctuation, capital to small letter conversion, removal of numbers.

- Mytext = “Hi Mr. Smith! I’m going to buy some vegetables (tomatoes and cucumbers) from the store. Should I pick up 2lbs of black-eyed peas as well?”
- Removal of punctuation Example



```
import re # Regular expression library
import string

# Replace punctuations with a white space
clean_text = re.sub('[%s]' % re.escape(string.punctuation), ' ', my_text)
clean_text
```

- **Convert to lower case**

```
clean_text = clean_text.lower()
clean_text
```

- **Remove numbers**

```
# Removes all words containing digits
clean_text = re.sub('\w*\d\w*', ' ', clean_text)
clean_text
```

- **Stop Words**

  Hi Mr. Smith! I'm going to buy some vegetables (tomatoes and cucumbers) from the store. Should I pick up some black-eyed peas as well?

- What is the most frequent term in the text above? Is that information meaningful?
- Stop words are words that have very **little semantic value.**
- There are language and context-specific stop word lists online that you can use.

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

Stop word removal

```
input_str = “NLTK is a leading platform for building Python
programs to work with human language data.”
stop_words = set(stopwords.words(‘english’))
from nltk.tokenize import word_tokenize
tokens = word_tokenize(input_str)
result = [i for i in tokens if  not i in stop_words]
print (result)
```

Pre-processing: Stemming

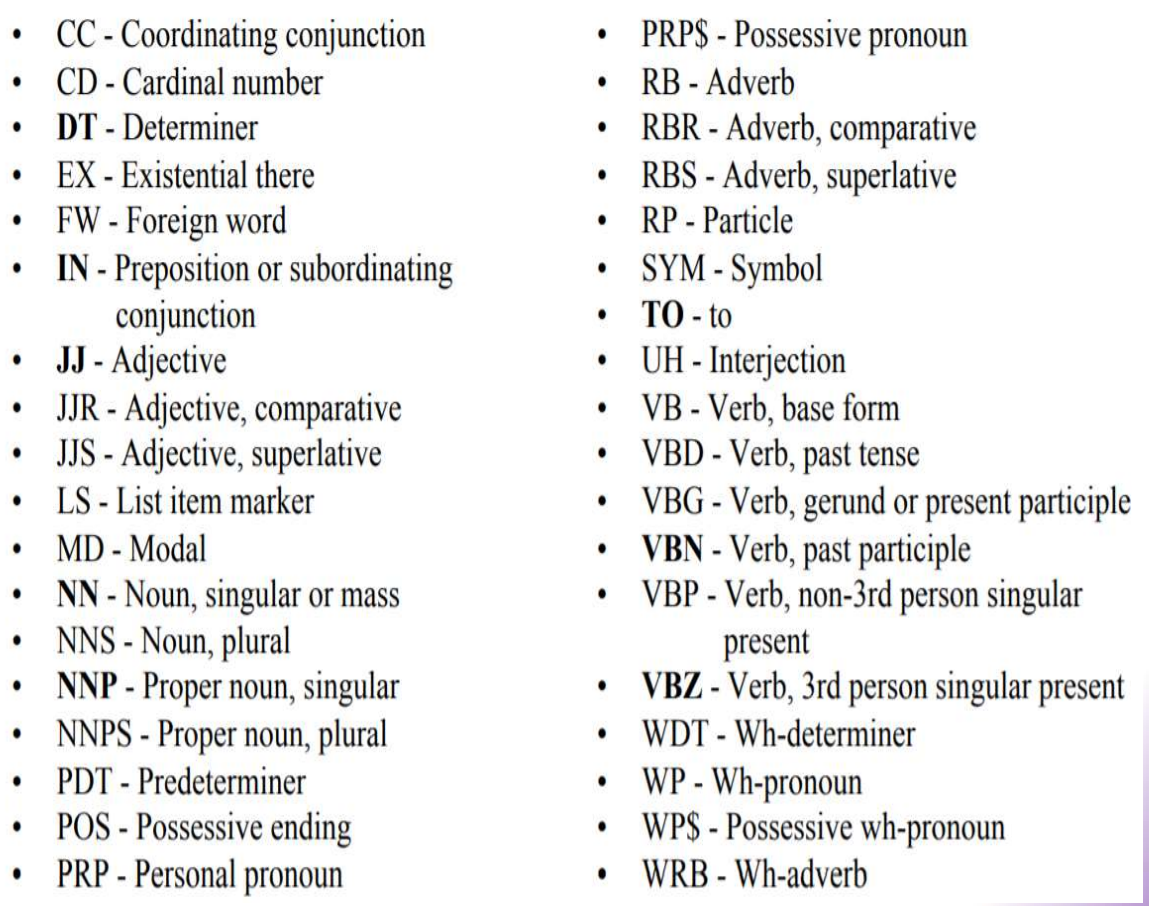
- **Stemming & Lemmatization** = Cut word down to base form
- **Stemming:** Uses rough heuristics to reduce words to base
- **Lemmatization:** Uses vocabulary and morphological analysis
- Makes the meaning of run, runs, running, ran all the same
- Cuts down on complexity by reducing the number of unique words
- Multiple stemmers available in NLTK
  - PorterStemmer(old), LancasterStemmer (new one added in 1990), SnowballStemmer (Non-English Stemmers)
  - WordNetLemmatizer(work based on wordnet lexical DB)
  - Demo available : https://text-processing.com/demo/stem/

Porter Stemmer Vs Lancaster Stemmer

Lancaster Stemmer produces an even shorter stem than porter because of iterations and over-stemming is occurred

Pre processing - Parts of Speech

- Nouns, verbs, adjectives, etc.
- Parts of speech tagging labels each word as a part of speech
- Available POS tags
- How to check
- **nltk.help.upenn_tagset()**



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

Pre- processing – Named Entity Recognition

- Referred as entities
- Identifies and tags named entities in text (people, places, organizations, phone numbers, emails, etc.)
- Can be tremendously valuable for further NLP tasks
- For example: “United States” --> “United_States”

Pre-processing Recap

Tokenization

Sentence
Word
N-Gram
Regex

Remove

Punctuation
Capital Letters
Numbers
Stop Words

Chunking

Named Entity Recognition

Compound Term Extraction

More

Stemming
Parts of Speech
Misspellings
Diff Languages

Text Similarity Measures

- What are Text Similarity Measures?
- Text Similarity Measures are metrics that measure the similarity or distance

between two text strings.

- It can be done on surface closeness (**lexical similarity**) of the text strings or meaning closeness (**semantic similarity**)
- lexical **word similarities** and lexical **documents similarities**.
- Many applications - Measuring similarity between documents
- Example :information retrieval, text classification, document clustering, topic modeling, topic tracking, matrix decomposition

Text Similarity Measures

Word Similarity

- Levenshtein distance- known as edit distance
- Applications: Spell Check, Speech Recognition, Plagiarism detection

Document Similarity

- Count vectorizer and the document-term matrix
- Bag of words
- Cosine similarity
- Term frequency-inverse document frequency (TF-IDF)

Word Similarity

- Levenshtein distance: Minimum number of operations to get from one word to another.
- Levenshtein operations are
  - Deletions: Delete a character
  - Insertions: Insert a character
  - Mutations: Change a character

Example: kitten —> sitting

- kitten —> sitten (1 letter change)
- sitten —> sittin (1 letter change)
- sittin —> sitting (1 letter insertion)

Levenshtein distance = 3

Word Similarity example

How similar are the following pairs of words?

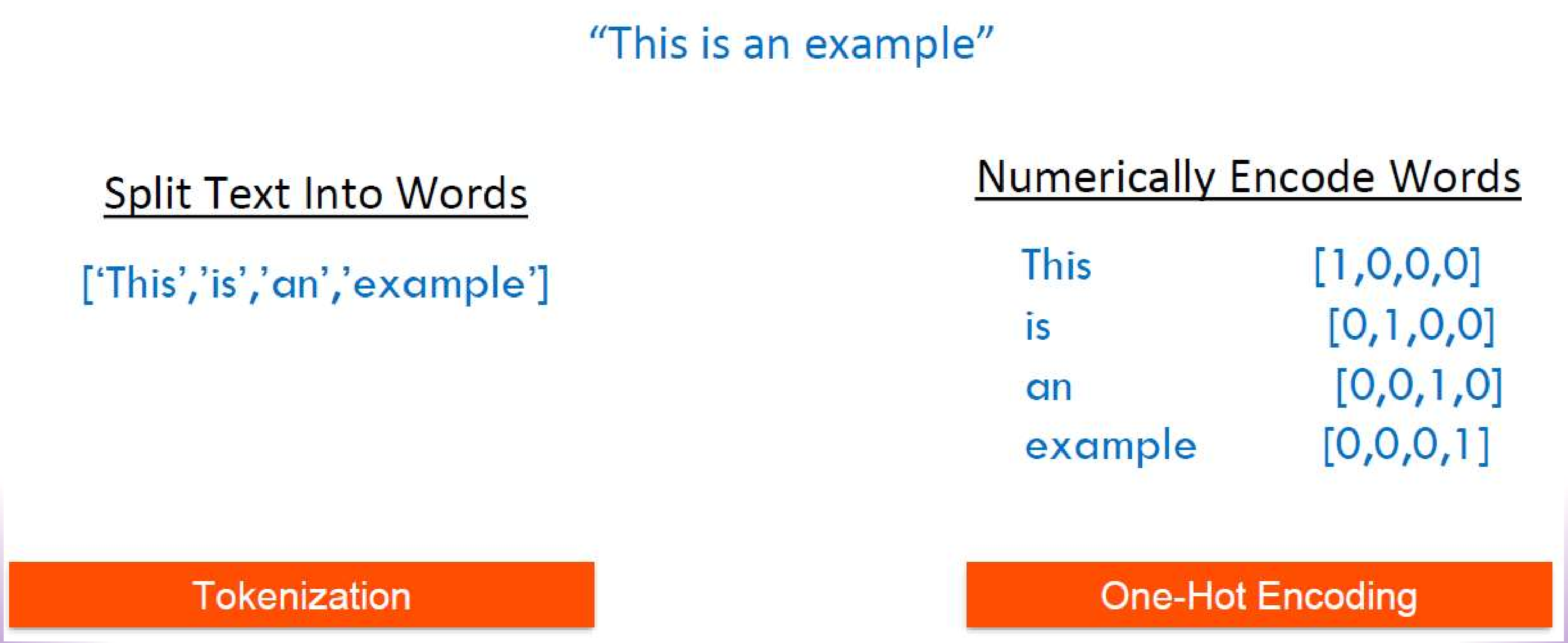
MATH	MATH	Levenshtein distance = 0
MATH	BATH	Levenshtein distance = 1
MATH	BAT	Levenshtein distance = 2
MATH	SMASH	Levenshtein distance = 2

Document Similarity

- Used in large number of documents and trying to find similar ones
- Group, or cluster, together similar documents – Document Summarization
- Comparing Documents needs the following
  - Tokenization
  - Count vectorizer and the document-term matrix

Text format for analysis - Example

- Few ways that text data can be put into a standard format for analysis



"This is an example"

Split Text Into Words

['This','is','an','example']

Tokenization

Numerically Encode Words

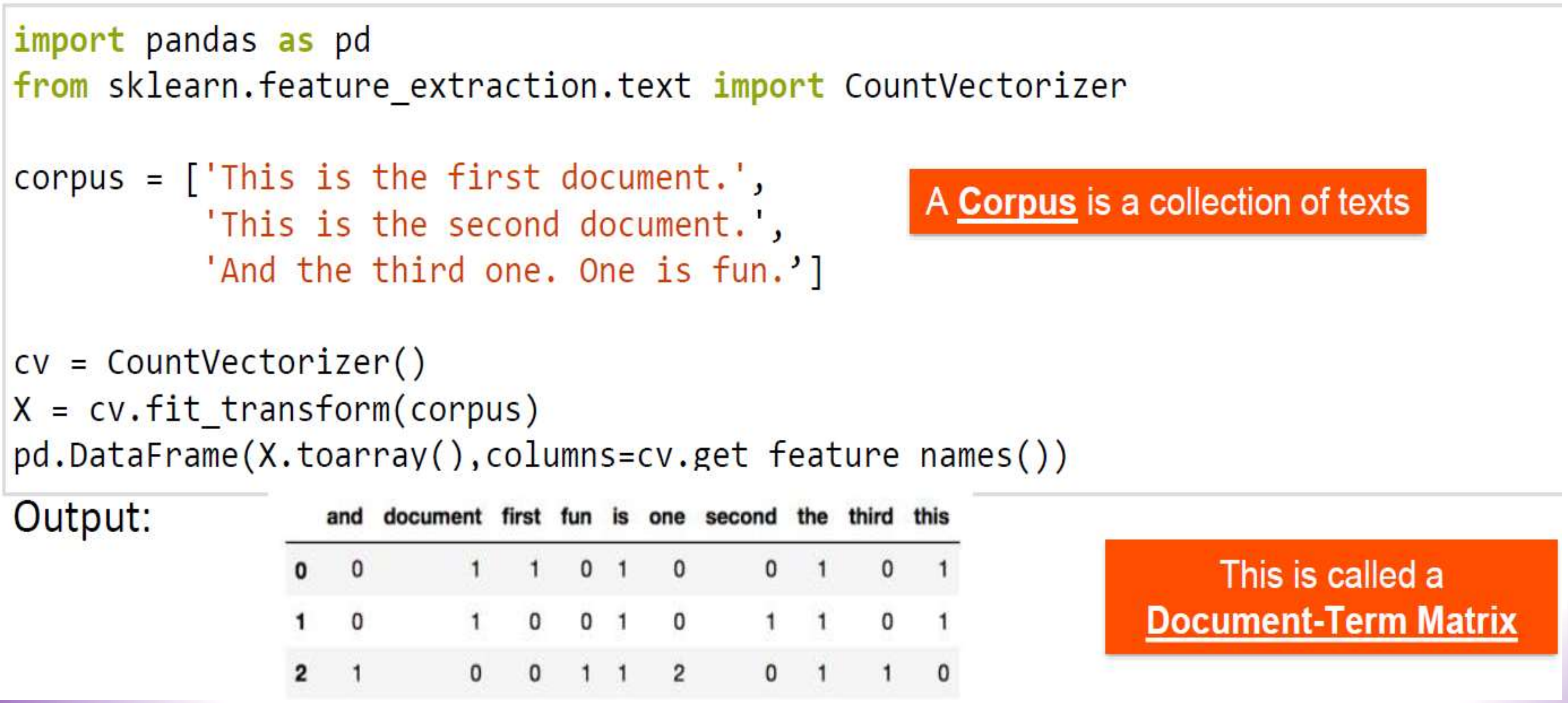
This	[1,0,0,0]
is	[0,1,0,0]
an	[0,0,1,0]
example	[0,0,0,1]

One-Hot Encoding

Text format - Count Vectorizer

The Count Vectorizer helps us create a Document-Term Matrix

- Rows = documents
- Columns = terms



```python
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer

corpus = ['This is the first document.',
          'This is the second document.',
          'And the third one. One is fun.’]

cv = CountVectorizer()
X = cv.fit_transform(corpus)
pd.DataFrame(X.toarray(),columns=cv.get feature names())
```

A **Corpus** is a collection of texts

Output:

```
   and  document  first  fun  is  one  second  the  third  this
0    0         1      1    0   1    0       0    1      0     1
1    0         1      0    0   1    0       1    1      0     1
2    1         0      0    1   1    2       0    1      1     0
```

This is called a **Document-Term Matrix**

Text format- Key Concepts

- Bag of Words Model
  - Simplified representation of text, where each document is recognized as a bag of its words
  - Grammar and word order are disregarded, but multiplicity is kept

Document Similarity: Cosine Similarity

- Cosine Similarity is a way to quantify the similarity between documents
- Step 1: Put each document in vector format
- Step 2: Find the cosine of the angle between the documents
- **Doc – 1 "I love you"**
- **Doc 2 - "I love NLP"**
- **i love you nlp**

	i	love	you	nlp
Doc 1	1	1	1	0
Doc 2	1	1	0	1

$$\mathbf{A} \cdot \mathbf{B} = \|\mathbf{A}\| \, \|\mathbf{B}\| \cos\theta$$

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|_2 \|\mathbf{B}\|_2}$$

a = [1, 1, 1, 0]
b = [1, 1, 0, 1]

= 0.667

- **Cosine similarity measures the similarity between two non-zero vectors with the cosine of the angle between them.**

Document Similarity: Example

- Here are five documents. Which ones seem most similar to you?

"The weather is hot under the sun"

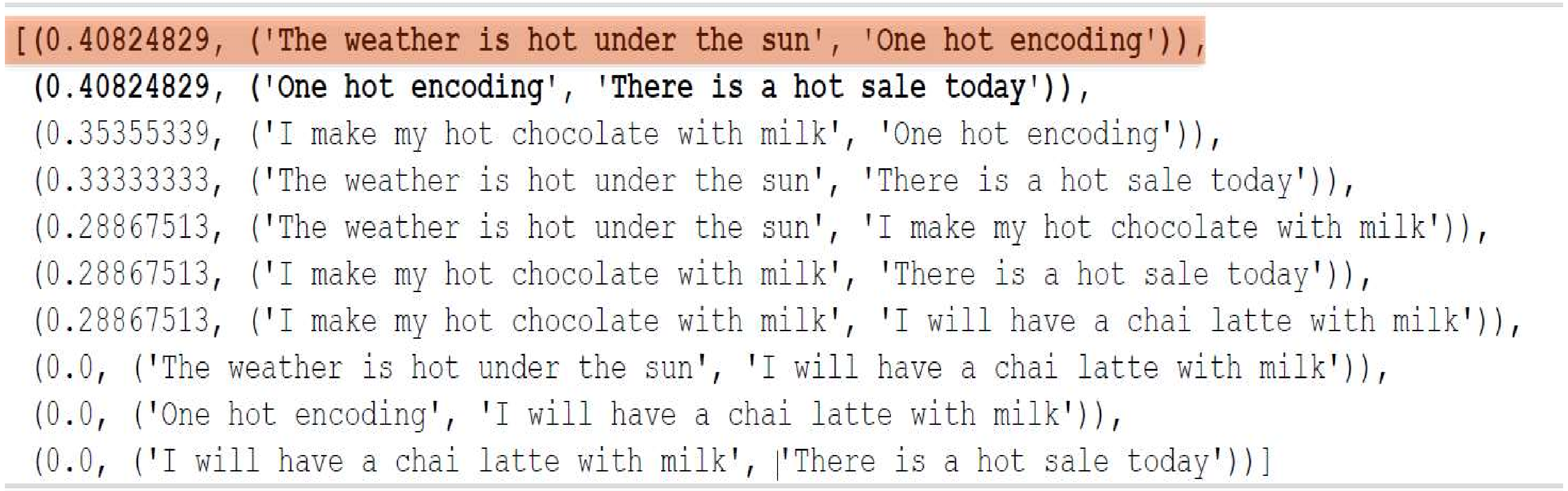
"I make my hot chocolate with milk"

"One hot encoding"

"I will have a chai latte with milk"

"There is a hot sale today"

Why Term Frequency-Inverse Document Frequency (TF-IDF)



```
[(0.40824829, ('The weather is hot under the sun', 'One hot encoding')),
 (0.40824829, ('One hot encoding', 'There is a hot sale today')),
 (0.35355339, ('I make my hot chocolate with milk', 'One hot encoding')),
 (0.33333333, ('The weather is hot under the sun', 'There is a hot sale today')),
 (0.28867513, ('The weather is hot under the sun', 'I make my hot chocolate with milk')),
 (0.28867513, ('I make my hot chocolate with milk', 'There is a hot sale today')),
 (0.28867513, ('I make my hot chocolate with milk', 'I will have a chai latte with milk')),
 (0.0, ('The weather is hot under the sun', 'I will have a chai latte with milk')),
 (0.0, ('One hot encoding', 'I will have a chai latte with milk')),
 (0.0, ('I will have a chai latte with milk', 'There is a hot sale today'))]
```

These two documents are most similar, but it's just because the term "hot" is a popular word

"Milk" seems to be a better differentiator, so how we can mathematically highlight that?

TF-IDF

TF-IDF = (Term Frequency) * (Inverse Document Frequency)

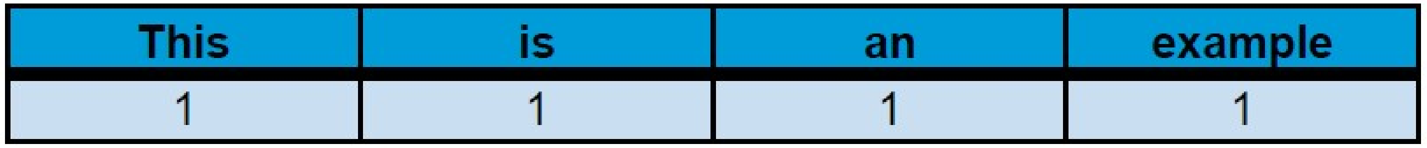
Different value for every document / term combination

$$\frac{\text{Term Count in Document}}{\text{Total Terms in Document}}$$

$$\log\left(\frac{\text{Total Documents} + 1}{\text{Documents Containing the Term} + 1}\right)$$

TF Vs IDF

- Term Frequency - recording the term (word) count
- “This is an example”



This	is	an	example
1	1	1	1

- However, if there were two documents, one very long and one very short, it wouldn’t be fair to compare them by word count alone
- A better way to compare them is by a normalized term frequency, which is **(term count) / (total terms).**

Term Frequency-Inverse Document Frequency advantages

- Assigns more weight to rare words and less weight to commonly occurring words.
- Tells us how frequent a word is in a document relative to its frequency in the entire corpus.
- Tells us that two documents are similar when they have more rare words in common.