Natural Language Processing NLP

T5 Bootcamp by SDAIA

Objectives

By the end of this module, trainees will have a comprehensive understanding of:

- ✓ Introduction to Natural Language Processing
- ✓ Text Preprocessing
- ✓ Classical NLP Techniques
- ✓ Neural Networks in NLP

Agenda

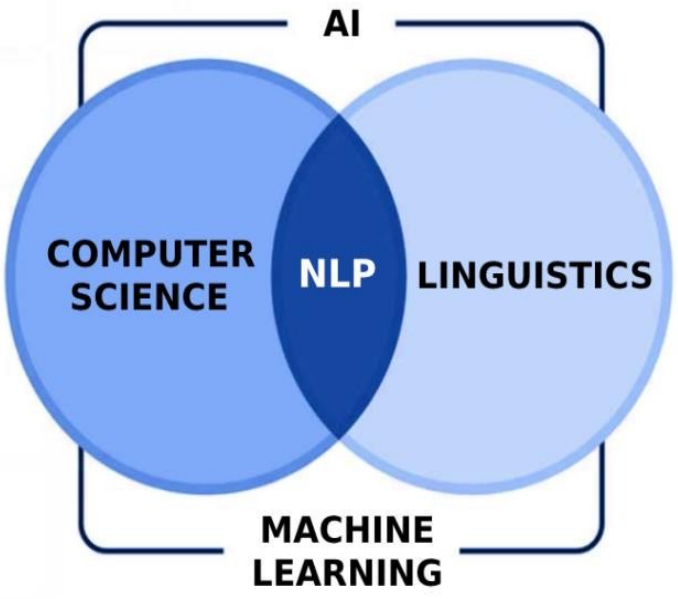
- Introduction to Natural Language Processing
- Text Preprocessing
- Word Embeddings
- Tradition Approach for Text Representation
- Neural Approach for Text Representation
- Pre-Trained Embeddings

Introduction to Natural Language Processing

What is Natural Language Processing?

Natural language processing (NLP) is the intersection of computer science, linguistics and machine learning.

- Natural language processing, or NLP, combines computational linguistics—rule-based modeling of human language—with statistical and machine learning models to enable computers and digital devices to recognize, understand and generate text and speech.
- Applications of NLP techniques include voice assistants like Amazon's Alexa and Apple's Siri, but also things like machine translation and text-filtering.
- The ultimate goal of NLP is all about making computers understand and generate human language.



Why NLP is difficult?

Understanding human language is considered a difficult task due to its complexity. Also, There are **more than 7,100 languages in the world today…!**

Spoken language is often riddled with slang, irony, homonyms, and different dialects/inflections, making it difficult to pin down exact meanings without human input.

For natural language processing to reach its potential, it needs to continuously process more and more data—something that can be costly and time-consuming for all but the largest organizations.

- ✓ **Language is evolutionary**
  - New words, new meanings each and everyday, ex: “Googling”
  - Different meaning in Different Context.
- ✓ **Language is subtle**
  - “book” → verb or noun
  - “bank” → financial institution or river-side
  - “People like ice-cream” →Defining scope
  - “make up a story” → word with multiple parts.

Language is COMPLEX..!

NLP Terminology

There are some key fundamentals of natural language:

1. **Syntax** - This refers to the rules and structures of the arrangement of words to create a sentence.
2. **Semantics** - This refers to the meaning behind words, phrases and sentences in language.
3. **Morphology** - This refers to the study of the actual structure of words and how they are formed from smaller units called morphemes.
4. **Pragmatics** - This is the study of how context plays a big role in the interpretation of language, for example, tone.
5. **Phonology/Phoneme** - This refers to the study of sounds in language, and how the distinct units are formed together to combine words.
6. **Ambiguity** - This refers to words or sentences with multiple interpretations.
7. **Polysemy** - This refers to words with multiple related meanings.

Technique to understand Natural Language

The way we understand what someone has said is an unconscious process relying on our intuition and knowledge about language itself.

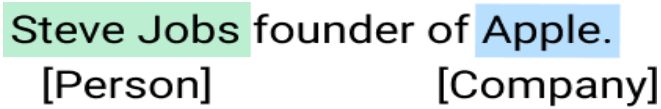
Syntactic analysis (syntax) and **Semantic** analysis (semantic) are the two primary techniques that lead to the understanding of natural language. Language is a set of valid sentences, but what makes a sentence valid? Syntax and semantics.

- Syntax is the grammatical structure of the text, whereas semantics is the meaning being conveyed.
- A sentence that is syntactically correct, however, is not always semantically correct.
- For example, “cows flow supremely” is grammatically valid (subject—verb—adverb) but it doesn’t make any sense.

Syntactic and Semantic Analysis

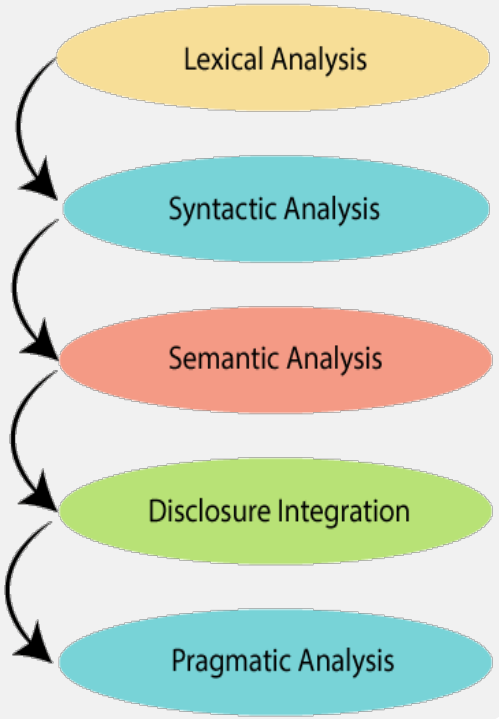
- **Syntactic** analysis is the process of analyzing natural language with the rules of a formal grammar.
  - Grammatical rules are applied to categories and groups of words, not individual words.
  - For example, a sentence “The dog (noun phrase) went away (verb phrase).” Note how we can combine every noun phrase with a verb phrase.
  - Again, it’s important to highlight that a sentence can be syntactically correct but not make sense.

- **Semantic** analysis is the process of understanding the meaning and interpretation of words, signs and sentence structure.
  - This allows computers to partly understand natural language the way humans do.
  - Semantic analysis is one of the toughest parts of natural language processing and it’s not fully solved yet.
  - Voice Recognition basically can understand what you have said but doesn’t understand the meaning behind it.



NLP Analysis Types

- **Lexical Analysis:** Involves identifying and analyzing the structure of words, dividing text into paragraphs, sentences, and words. Lexicon represents the collection of words and phrases in a language.
- **Syntactic Analysis:** Analyzes words in a sentence for grammar and arranges them to show relationships among words. Incorrect sentences like “The school goes to boy” are rejected by English syntactic analyzers.
- **Semantic Analysis:** Determines the exact meaning or dictionary meaning from the text, ensuring meaningfulness by mapping syntactic structures and objects in the task domain. Sentence examples like “hot ice-cream” are disregarded.
- **Discourse Integration:** Considers the meaning of each sentence in relation to the sentence before it and after it, ensuring coherence and continuity in the overall text.
- **Pragmatic Analysis:** Reinterprets what was said to understand what it actually meant, requiring real-world knowledge to derive aspects of language beyond literal meanings.



NLP Applications

Voicemail to text transcription: Most smartphones today offer this type of transcription, where NLP listens to and transcribes your voicemails.

Customer service chatbots: While many chatbots today operate off a pre-written script, AI-powered chatbots can help tackle more complex customer requests.

Seamless translation: From travelers to government officials, seamless voice to text translation (or vise versa) can always come in handy.

Voice-operated GPS systems: If you have a newer GPS system, you've likely input your destination via voice. NLP helps your GPS parse through your language.

Speech recognition, also called speech-to-text, is the task of reliably converting voice data into text data.

Common Approaches in NLP

Supervised NLP

Supervised NLP methods train the model with a set of labeled or known input and output.

For example, companies train NLP tools to categorize documents according to specific labels.

Unsupervised NLP

Unsupervised NLP uses language modes to predict the pattern that occurs when it is fed a non-labeled text.

For example, the autocomplete feature which suggests relevant words that make sense.

Natural Language Understanding (NLU)

Natural language understanding focuses on analyzing the meaning behind sentences.

For example, a virtual assistant might use NLU to understand a user's request to book a flight.

Natural Language Generation (NLG)

Natural language generation focuses on producing conversational text like humans do based on context.

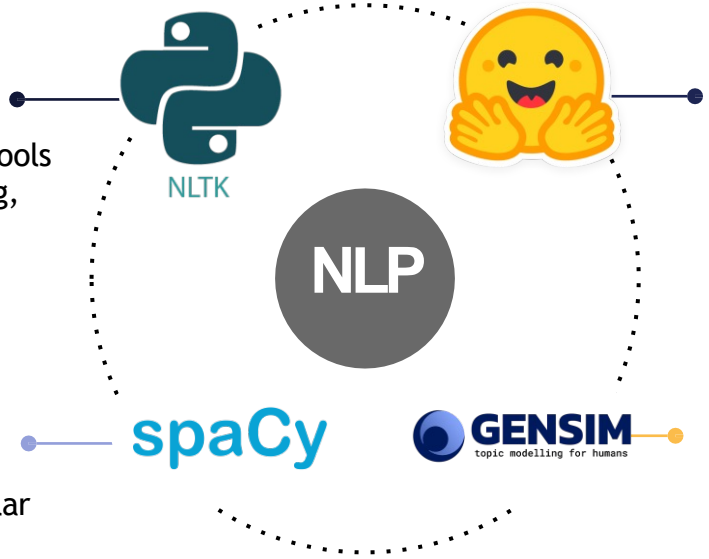
For example, chatbots with NLG capabilities can converse with customers.

NLP Libraries & Frameworks

Many languages and libraries support NLP. Below are some of most useful ones:

Natural Language Toolkit (NLTK)

One of the earliest NLP libraries in Python, It includes text-processing tools for classification, tagging, stemming, parsing, and semantic reasoning



Hugging Face

This platform provides open-source implementations and weights for over 135 state-of-the-art NLP models, offering easy customization and training capabilities.

spaCy

An NLP library supporting over 66 languages, spaCy offers pre-trained word vectors and implements popular models like BERT.

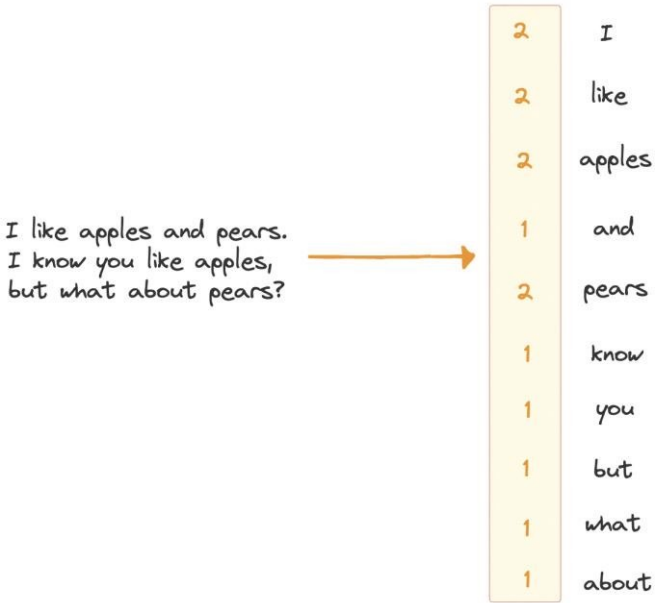
Gensim

Gensim specializes in vector space modeling and topic modeling algorithms, making it suitable for tasks like document similarity analysis and topic extraction.

How does NLP work?

Before you can ingest anything into your NLP model, you need to keep in mind that **computers only understand numbers**. Therefore, when you have text data, you will need to use text vectorization to transform the text into a format that the machine learning model can understand.

- Once the text data is vectorized in a format the machine can understand, the NLP machine learning algorithm is then fed training data.
- This training data helps the NLP model to understand the data, learn patterns, and make relationships about the input data.
- Once the model has gone through the training phase, it will then be put to the test through the testing phase to see how accurately the model can predict outcomes using unseen data.



Tokenization

Text Preprocessing Techniques: **Tokenization**

The text is split into smaller units. We can use either sentence tokenization or word tokenization based on our problem statement. You can easily tokenize the sentences and words of the text with the tokenize module of NLTK.

```
[17] import nltk
     nltk.download('punkt')
     from nltk.tokenize import word_tokenize, sent_tokenize

[18] ar_text= 'اهلا ومرحبا بكم في معسكر علم'
     text='The weather is warm today. It is a great day to go to the beach.'

[20] print(word_tokenize(ar_text))
     print(sent_tokenize(ar_text))

     ['اهلا', 'ومرحبا', 'بكم', 'في', 'معسكر', 'علم']
     ['اهلا ومرحبا بكم في معسكر علم']

[21] print(word_tokenize(text))
     print(sent_tokenize(text))

     ['The', 'weather', 'is', 'warm', 'today', '.', 'It', 'is', 'a', 'great', 'day', 'to', 'go', 'to', 'the', 'beach', '.']
     ['The weather is warm today.', 'It is a great day to go to the beach.']
```

Text Preprocessing Techniques: **Tokenization**

Types of Tokenization:

- **Word Tokenization**: Breaks text into individual words, suitable for languages with clear word boundaries like English.
  - Input: "Natural Language Processing is fascinating!"
  - Output: ["Natural", "Language", "Processing", "is", "fascinating!"]
- **Character Tokenization**: Segments text into individual characters, beneficial for languages without clear word boundaries or tasks like spelling correction.
  - Input: "Hello, world!"
  - Output: ['H', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!']
- **Sub-word Tokenization**: Divides text into units larger than a single character but smaller than a full word, striking a balance between word and character tokenization. Useful for languages forming meaning by combining smaller units or handling out-of-vocabulary words in NLP tasks.
  - Input: "Chatbots are becoming increasingly popular."
  - Output: ["Chat", "bots", "are", "becoming", "increasingly", "popular", "."]

Text Preprocessing Techniques: **Tokenization**

Challenges and Limitations of the tokenization task:

- In general, this task is used for text corpus written in English or French where these languages separate words by using white spaces, or punctuation marks to define the boundary of the sentences.
- In the tokenization of Arabic texts since Arabic has a complicated morphology as a language.

For example, a single Arabic word may contain up to six different tokens like the word (“دقّع”)

Programming Tutorial Module 1 (Tokenization)

Information Retrieval

IR terminology

- **Document** refers to any unit of text indexed in the system and available for retrieval. It can refer to any piece of text (e.g., a newspaper article, a report, a social media post) or even smaller textual units (e.g., paragraphs or sentence)
- **Collection** refers to a set of documents that may satisfy user requests (e.g., set of texts or webpages)
- **Term** refers an item (a word or a phrase) that occurs in a collection and helps the algorithm in finding relevant documents in the collection
- **Query** represents a user's information need expressed as a set of search terms

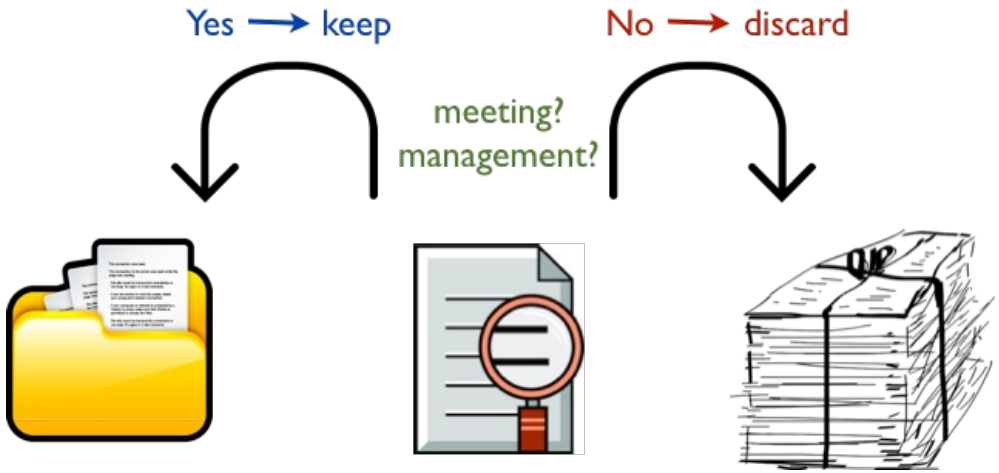
How does this apply to our scenario?

IR terminology

In our example:

- **Document** is a set of meeting notes or a report
- **Collection** contains all such meeting notes and reports on your computer
- **Terms** include “management” and “meeting(s)”
- **Query** contains these search terms

How to find *all* relevant documents?



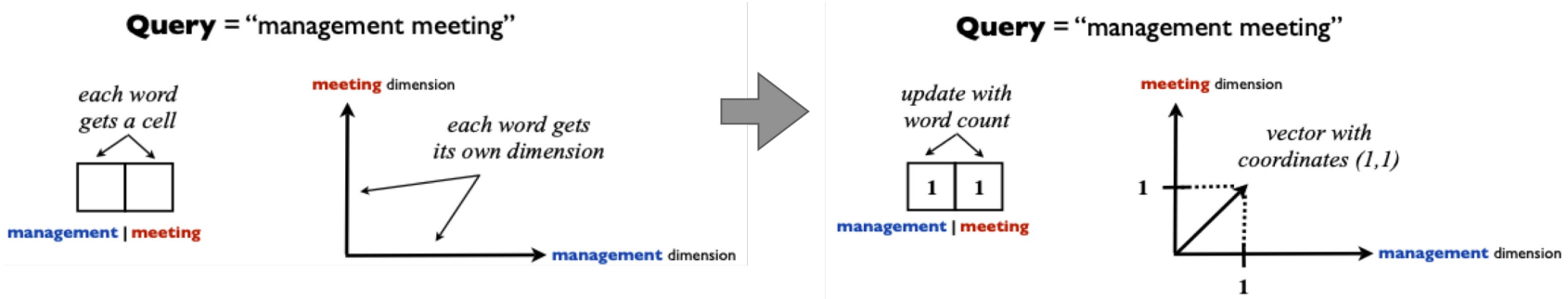
Idea: return all documents that contain both search terms “meeting” and “management” and discard all the rest. If there are multiple such documents, how should we choose the most relevant ones?

How can the algorithm identify relevant terms in text?

- First, recall the material from Week 1: the text comes in as a sequence of symbols ⇒
  tokenize
- Second, register all relevant words' occurrences in documents. This can be achieved with the use of such data structures as *lists* or *arrays*
- This brings us to **vector representations**: they allow us to represent both query terms and document terms in a shared *vector space* and measure relevance of documents to the query using simple methods from *Euclidean geometry*

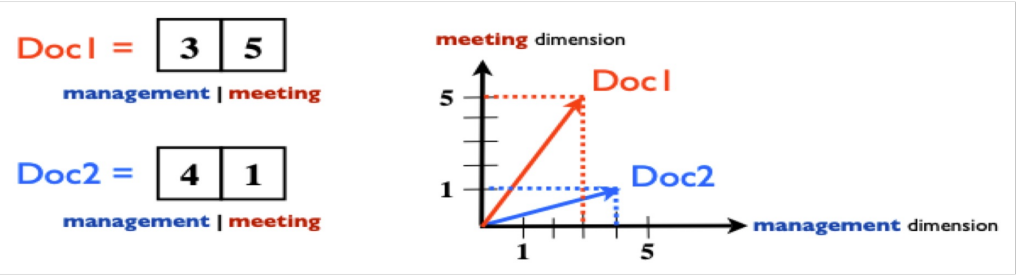
Vector-based representations for queries

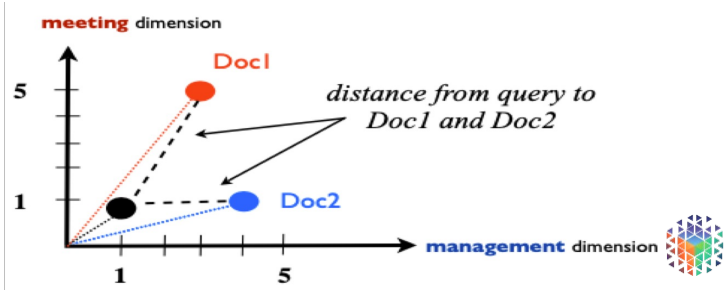
- Let's consider **"management meeting"**
- There are two search terms – **"management"** and **"meeting"**
- Let's give each of these terms their **own dimension** and represent this query in a 2- dimensional vector space (alternatively, think of this as an array with two cells)
- **Values** in each cell and coordinates in each dimension are defined by the number of each word occurrences



Vector-based representations for documents

- Suppose **doc1** contains 3 occurrences of “management” and 5 of “meeting” ⇒ (3, 5)
- Suppose **doc2** contains 4 occurrences of “management” and 1 of “meeting” ⇒ (4, 1)
- Let’s put them in the **same space** as the query
- We can interpret similarity as **Euclidean distance** in this space
- The *most relevant document* is the one with the *most similar content*, i.e., with similar occurrences of the overlapping terms ⇒ **closest in distance**





Similarity as Euclidean distance

Distance *d* between two points defined by the coordinates as (p_1, q_1) and (p_2, q_2):

$$d(p, q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}.$$

For more than two dimensions:

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_i - q_i)^2 + \cdots + (p_n - q_n)^2}.$$

What is *d*(query, doc1) and *d*(query, doc2) for our example of *query*=(1,1), *doc1*=(3, 5) and *doc2*=(4,1)

Which document is closer (more relevant) on this basis?

Similarity as Euclidean distance

Distance *d* between two points defined by the coordinates as (p_1, q_1) and (p_2, q_2):

$$d(p, q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}.$$

For more than two dimensions:

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_i - q_i)^2 + \cdots + (p_n - q_n)^2}.$$

For our example: ***d*(query, doc1)=4.47** and ***d*(query, doc2)=3**

d(query, doc2)=3 < *d*(query, doc1)=4.47 ⇒ **document2 is closer (more relevant)**

Similarity as Euclidean distance: caveat

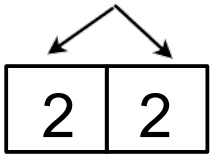
- If I repeated the words in the query several times (e.g., “management meeting management meeting”), will my information need be different?
- What will happen to the vector representation?

Similarity as Euclidean distance: caveat

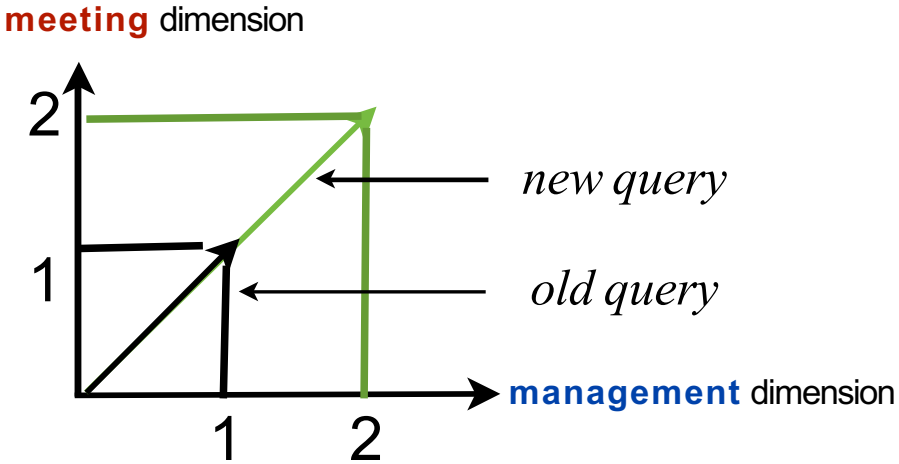
- If I repeated the words in the query several times (e.g., “management meeting management meeting”), will my information need be different?
- What will happen to the vector representation?

New Query = “management meeting management meeting”

counts increase



management | **meeting**



Cosine similarity

- Moreover, **longer documents** will have **longer vectors**
- Words in longer documents will have a **higher chance of occurrence** ⇒ higher raw counts ⇒ longer vectors
- This doesn't translate into higher importance **unless** you can **fairly compare** documents and their vectors; pure Euclidean distance doesn't account for that
- You can fairly compare documents / vectors if they are **length-normalized**
- **Cosine similarity** is such a metric: for vectors $A_i = (A_0, A_1, \ldots, A_N)$ and $B_i = (B_0, B_1, \ldots, B_N)$ it is defined as

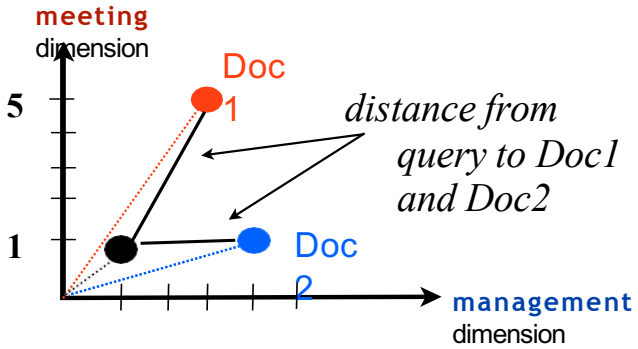
$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

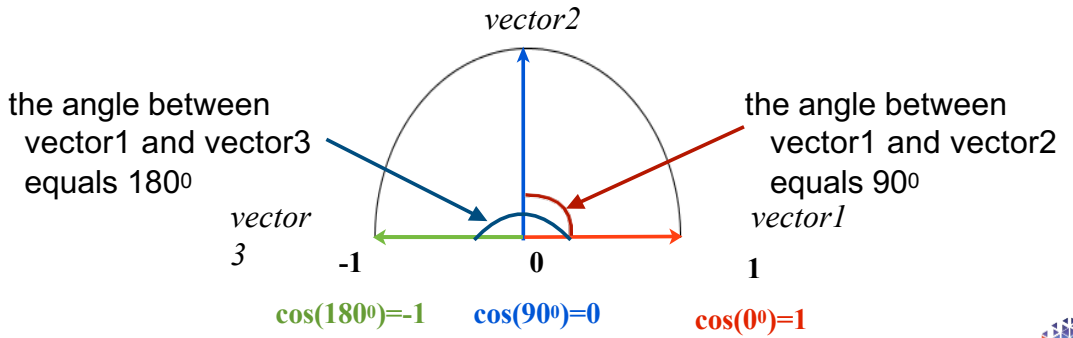
Cosine similarity

- **Cosine similarity** is such a metric: for vectors $A_i = (A_0, A_1, \ldots, A_N)$ and $B_i = (B_0, B_1, \ldots, B_N)$ it is defined as

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

- What is *cos*(q, d1) and *cos*(q, d2) for our example: q=(1,1), d1=(3,5), d2=(4,1)?
- Which document is more relevant on this basis (higher cosine)?





Cosine similarity

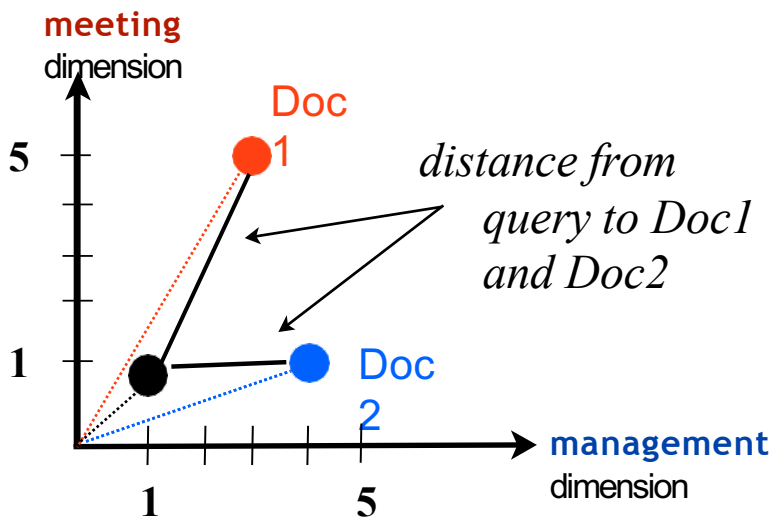
- **Cosine similarity** is such a metric: for vectors $A_i = (A_0, A_1, \ldots, A_N)$ and $B_i = (B_0, B_1, \ldots, B_N)$ it is defined as

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

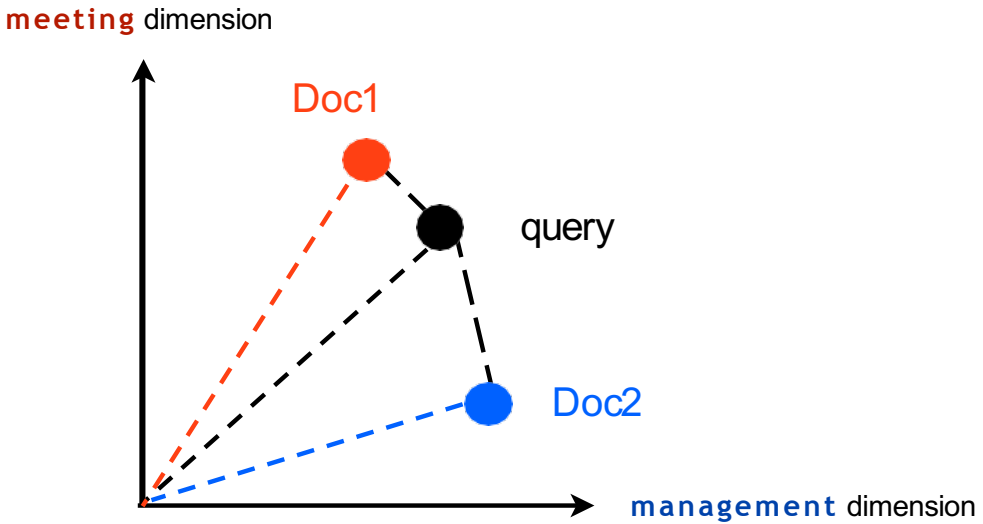
- For our example:
  - *len*(q) = sqrt(1^2+1^2) = 1.41; *len*(d1) = 5.83; *len*(d2) = 4.12
  - *dot_prod*(q, d1) = 8; *dot_prod*(q, d2) = 5
  - ***cos*(q, d1) = 0.97**; ***cos*(q, d2) = 0.86**
- *cos*(query, doc1)=0.97 > *cos*(query, doc2)=0.86 ⇒ document1 is more relevant

Cosine similarity

Before length normalization:



After length normalization:



Programming Tutorial Module 1: (Term weighting)

Programming Tutorial Module 1: (End to End IR)

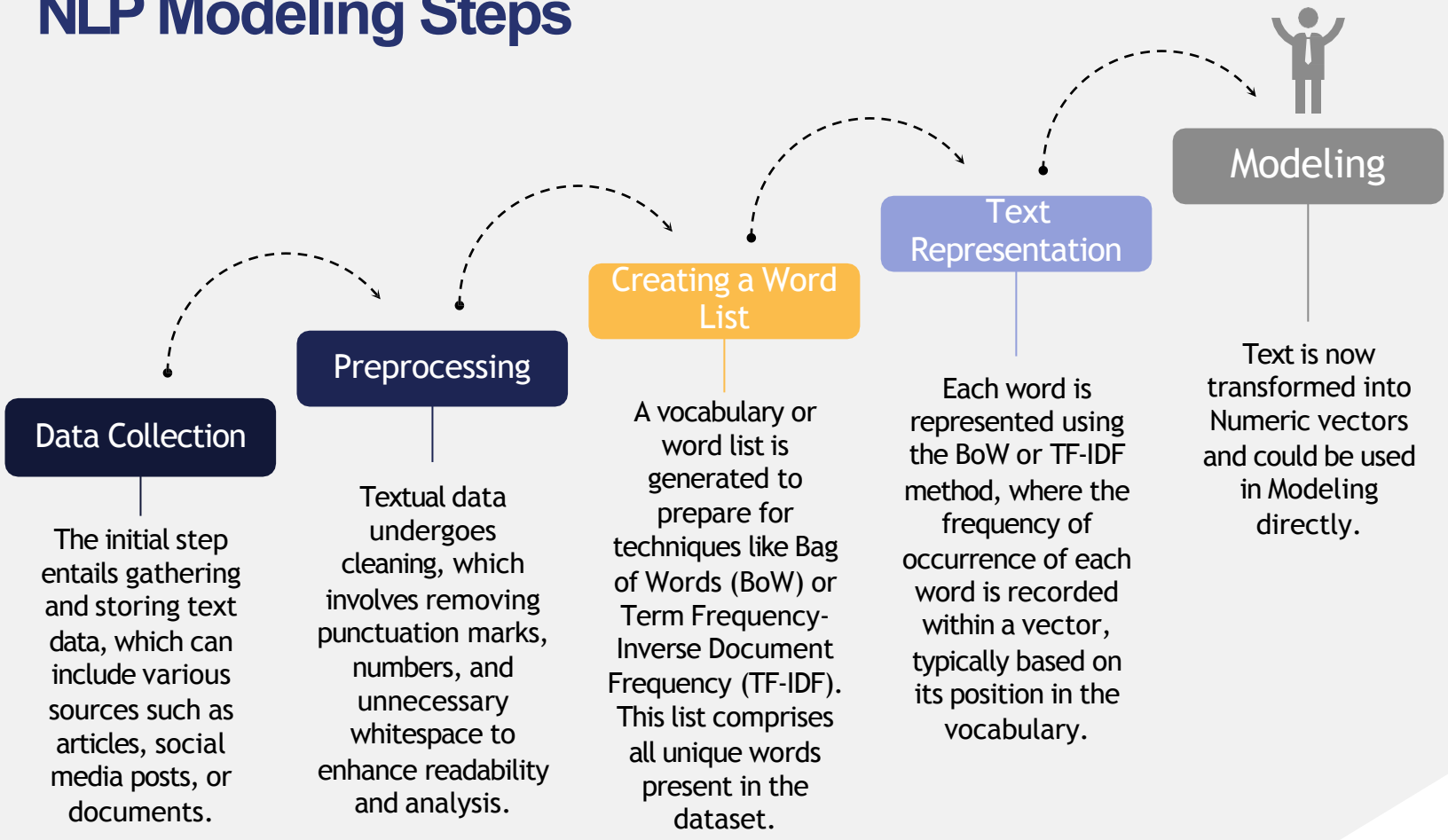
Text Preprocessing

Why is Text Preprocessing important?

Data quality significantly influences the performance of a machine-learning model. Inadequate or low-quality data can lead to lower accuracy and effectiveness of the model.

- In general, text data derived from natural language is unstructured and noisy.
- Text preprocessing is a critical step to transform messy, unstructured text data into a form that can be effectively used to train machine learning models, leading to better results and insights.
- The goal of text preprocessing is to enhance the quality and usability of the text data for subsequent analysis or modeling.

NLP Modeling Steps



Text Preprocessing Techniques

Text preprocessing refers to a series of techniques used to clean, transform and prepare raw textual data into a format that is suitable for NLP or ML or DL tasks

Text preprocessing typically involves the following steps:

- Lowercasing
- Removing Punctuation & Special Characters
- Stop-Words Removal
- Removal of URLs
- Removal of HTML Tags
- Stemming & Lemmatization
- Tokenization
- Text Normalization

The order in which these techniques are applied may vary depending on the needs of the project.

Text Preprocessing Techniques: **Lowercasing**

Lowercasing is a text preprocessing step where all letters in the text are converted to lowercase. This step is implemented so that the algorithm does not treat the same words differently in different situations.

```
text = "Hello T5 Students!"
lowercased_text = text.lower()

print(lowercased_text)
```

```
Output:

hello t5 students!
```


Text Preprocessing Techniques: **Removing Punctuation & Special Characters**

Punctuation removal is a text preprocessing step where you remove all punctuation marks *(such as periods, commas, exclamation marks, emojis etc.)* from the text to simplify it and focus on the words themselves.

```python
import re

text = "Hello, T5 Students! This is?*  💜an&/|~^+%'\" example- of text preprocessing."
punctuation_pattern = r'[^\w\s]'

text_cleaned = re.sub(punctuation_pattern, '', text)
print(text_cleaned)
```

```
Output:

Hello T5 Students This is an example of text preprocessing
```


Text Preprocessing Techniques: **Stop-Words Removal**

Stopwords are words that don't contribute to the meaning of a sentence. The NLTK library has a set of stopwords, and we can use these to remove stopwords from our text and return a list of word allowing the focus on the important words.

```
[27] from nltk.corpus import stopwords

     #remove english stopwords function
     def remove_stopwords(text, language):
       stop_words= set(stopwords.words(language))
       word_tokens = text.split()
       filtered_text= [word for word in word_tokens if word not in stop_words]
       print(language,filtered_text)

     remove_stopwords(ar_text,'arabic')
     remove_stopwords(text,'english')
```

```
arabic ['معسكر' ,'ومرحبا' ,'اهلا']
english ['The', 'weather', 'warm', 'today.', 'It', 'great', 'day', 'go', 'beach.']
```

Text Preprocessing Techniques: **Removal of URLs**

This preprocessing step is to remove any URLs present in the data.

```
import re
def remove_urls(text):
     url_pattern = re.compile(r'https?://\S+|www\.\S+')
     return url_pattern.sub(r'', text)
ar_text =  " https://sdaia.tuwaiq.edu.sa/: ركسهل طبارلا مولع يلالو ءاكذلا وه يعانطصلاا "
en_text = " The link for T5 Bootcamp:  https://sdaia.tuwaiq.edu.sa/"
print(remove_urls(ar_text))
print(remove_urls(en_text))
```

```
Output:

  وه يعانطصلاا ءاكذلا و يلالا مولع ركسهل طبارلا
 The link for T5 Bootcamp:
```

Text Preprocessing Techniques: **N-Grams**

N-grams are continuous sequences of words or symbols, or tokens in a document. In technical terms, they are a type of tokenization which can be defined as the neighboring sequences of items in a document.

```
[23] from nltk import ngrams
     ar_text= 'اهلا ومرحبا بكم في معسكر علم'
     n=2
     unigrams = ngrams(ar_text.split(),n)
     for grams in unigrams:
       print(grams)
```

```
('اهلا', 'ومرحبا')
('ومرحبا', 'بكم')
('بكم', 'في')
('في', 'معسكر')
('معسكر', 'علم')
```

```
[24] from nltk import ngrams
     ar_text= 'اهلا ومرحبا بكم في معسكر علم'
     n=3
     unigrams = ngrams(ar_text.split(),n)
     for grams in unigrams:
       print(grams)
```

```
('اهلا', 'ومرحبا', 'بكم')
('ومرحبا', 'بكم', 'في')
('بكم', 'في', 'معسكر')
('في', 'معسكر', 'علم')
```

Text Preprocessing Techniques: **N-Grams**

English Example:

```
from nltk import ngrams
sentence = "The weather is warm and
sunny today"
n = 2
unigrams = ngrams(sentence.split(), n)
for grams in unigrams:
    print (grams)
```

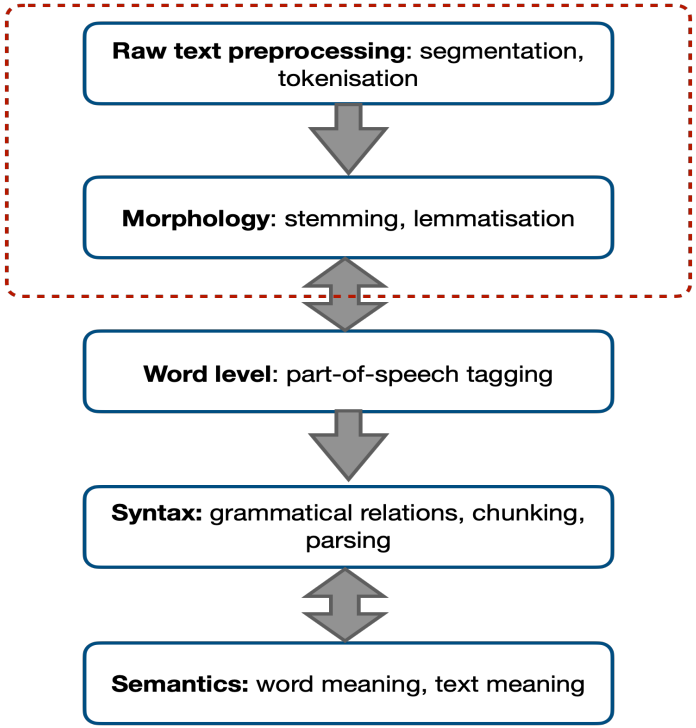
```
Output:

('The', 'weather')
('weather', 'is')
('is', 'warm')
('warm', 'and')
('and', 'sunny')
('sunny', 'today')
```

```
from nltk import ngrams
sentence = "The weather is warm and
sunny today"
n = 3
unigrams = ngrams(sentence.split(), n)
for grams in unigrams:
    print (grams)
```

```
Output:

('The', 'weather', 'is')
('weather', 'is', 'warm')
('is', 'warm', 'and')
('warm', 'and', 'sunny')
('and', 'sunny', 'today')
```

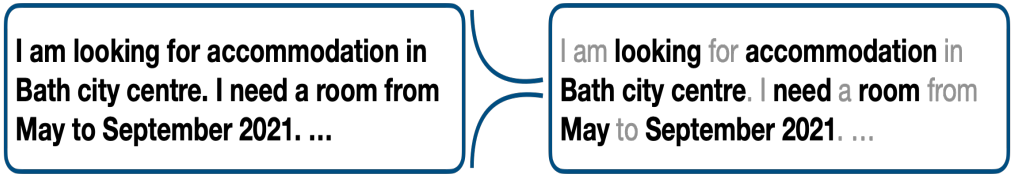

Reminder : levels of linguistic analysis

- Recall that raw text preprocessing (**tokenization**) is the first step in many NLP applications
- Last week we also discussed that not all words are equally important: some words are typically filtered out as they are considered **stopwords**
- **Implications for IR**: note that each word in the query or document adds another dimension to the shared vector space ⇒ adds complexity but doesn't always help to solve the task (think stopwords)



Defining key terms space

- It is safe to filter out stopwords and punctuation marks:



- Are there any other challenges? In the following example, filtered-out words are greyed out, matching words are highlighted in blue, and important words that will currently be missed by the algorithm are highlighted in red:

query15

How much do information retrieval and dissemination systems, as well as automated libraries, cost?

doc27

A computer program has been written and used which simulates the several-year operation of an information **system** and computes estimates of the expected operating **costs** as well as the amount of equipment and personnel required during that time period. [...]

Defining key terms space

query15

How much do information retrieval and dissemination systems, as well as automated libraries, cost?

doc27

A computer program has been written and used which simulates the several-year operation of an information **system** and computes estimates of the expected operating **costs** as well as the amount of equipment and personnel required during that time period. [...]

- **Implications for an IR system**: if the system cannot match *"system"* ↔ *"systems"* and *"cost"* ↔ *"costs"*, it will create two different dimensions for each pair ⇒ complexity of the algorithm will increase, but useful information will be lost
- **Solution**: match different forms of the same word to their *common core* or *shared base form*
- This is achieved by **stemming** and **lemmatization** algorithms

Text Preprocessing Techniques: **Stemming**

It is also known as the text standardization step where the words are stemmed or diminished to their root/base form. For example, words like ‘programmer’, ‘programming, ‘program’ will be stemmed to ‘program’.

- But the disadvantage of stemming is that it stems the words such that its root form loses the meaning, or it is not diminished to a proper English word.
- There are various algorithms that can be used for stemming,
    1. Porter Stemmer algorithm
    2. Snowball Stemmer algorithm
    3. Lovins Stemmer algorithm
    4. **ISRIStemmer algorithm for Arabic text**

Text Preprocessing Techniques: **Stemming**

Arabic Stemmer: ISRIStemmer

```
import nltk
from nltk.stem.isri import ISRIStemmer
st = ISRIStemmer()

# Define a function to perform stemming on the
'text' column
def stem_words(words):
    return [st.stem(word) for word in words]

stem_words(['البيانات','معسكر','مرحبا','اهلا '])
```

```
output:

[بين ، عسكر ، رحب ، اهل]
```


Morphology

- Different contexts require words in different forms:
    - single **system** vs. multiple **systems**
    - action taking place now (**watching** a video) vs. on a regular basis (**watch** in the evenings) vs. in the past (**watched** yesterday)
- Different aspects of the meaning highlighted: reference to time or continuity, singularity or multiplicity of objects, etc.
- Different forms of the same word are called **morphological forms**, and the branch of linguistics dealing with them is called **morphology**

English inflectional Morphology

For **nouns**:

- Base (singular) forms: *system*, *fox*
- Plural forms: *system**s***, *fox**es***

For **verbs**:

- Base (infinitive) forms: *watch*, *sing*
- 3rd person forms: *watch**es***, *sing**s***
- Past tense forms: *watch**ed***, *s**a**ng*
- Past participle forms: *watch**ed***, *s**u**ng*
- Progressive forms: *watch**ing***, *sing**ing***

- We can describe these word formations using rules: e.g., $BASE_{noun}$ + *s* and $BASE_{noun}$ + *es* for plural in nouns, or $BASE_{verb}$ + *ed* for past in verbs, etc.
- **BUT**: *mouse* ↔ *mice*, *man* ↔ *men*; *sing* ↔ *sang*, *be* ↔ *am* ↔ *is* ↔ *were*, etc.

Lemmatization

- **Base form** refers to the most basic form of the word, the starting point for any further formations. It is the form that you will find in the dictionary
- It is also referred to as **lemma**, and the process of converting any input word form to its lemma is called **lemmatization**
- In practice, lemmatization algorithms often combine look-up tables (especially for the irregular forms) with rule sets, which can be inferred using ML algorithms[1]
- **Pros**: lemmatization algorithms return base forms that are understandable to humans
- **Cons:** lemmatization is resource-intensive; look-up tables are needed for each language

Quiz

Lemmatize words in the following sentence:

```
Up to the 1980s, most natural language
processing systems were based on complex sets
of rules written by experts.
```


Quiz

Lemmatize words in the following sentence:

Up to the 1980s, most natural language processing systems were based on complex sets of rules written by experts.

Answer:

Up to the **1980**, most natural language processing **system be base** on complex **set** of **rule write** by **expert**.

Stemming

- In contrast, **stemming** uses algorithms that rely on a set of language-specific morphological rules
    - productive rules can be extended from existing words to any new words: e.g., *photo* → *photos* ⇒ *selfie* → *selfies*
- In addition, it tries to map all related words to the common core – **stem**
- So far, we have looked into **inflectional morphology** – grammatical forms within the same type of word (e.g., noun or verb): *retrieve* → *retrieves* and *retrieve* → *retrieved*
- **Derivational morphology** deals with related forms derived from the common stem that may cross type-of-word boundaries (e.g., from verb to noun): *retrieve* → *retrieval*

Stemming vs lemmatization

- **Lemmatization** will return lemma "*retrieve*" for {*retrieve*, *retrieves*, *retrieved*, *retrieving*} and "*retrieval*" for {*retrieval*}
- **Stemming** will return the common stem "*retriev-*" for {*retrieve*, *retrieves*, *retrieved*, *retrieving, retrieval*}
- Productive pattern identified: VERB_STEM + *-al* → NOUN
    - *retriev-* + *-al* → *retrieval*
    - *approv-* + *-al* → *approval*
    - *den(i)-* + *-al* → *denial*

A glimpse into the porter stemmer (1980)

- The algorithm takes into account the form of the word (e.g., *v* denotes that the stem contains a vowel) and defines an interdependent set of steps
- Examples of the first step:

SSES	-> SS	caresses	-> caress
IES	-> I	ponies	-> poni
		ties	-> ti
SS	-> SS	caress	-> caress
S	->	cats	-> cat

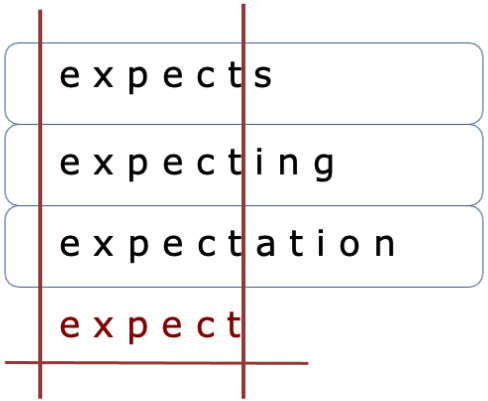
(*v*) ED	->	plastered	-> plaster
		bled	-> bled
(*v*) ING	->	motoring	-> motor
		sing	-> sing

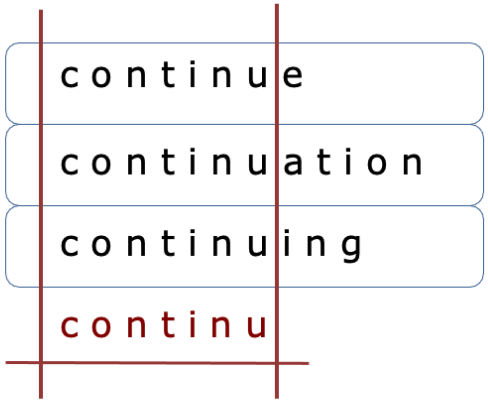
- Examples of the second step:

IZATION	-> IZE	vietnamization	-> vietnamize
ATION	-> ATE	predication	-> predicate
ATOR	-> ATE	operator	-> operate
ALISM	-> AL	feudalism	-> feudal
IVENESS	-> IVE	decisiveness	-> decisive

[1] See http://snowball.tartarus.org/algorithms/porter/stemmer.html for more details on the algorithm

Stemming examples





- **Pros:** stemmers are capable of establishing links between related words without the need for defining a set of dictionary base forms ⇒ efficient and scalable
- **Cons:** the results are not as “readable” as lemmas, and stemmers may be too “aggressive”

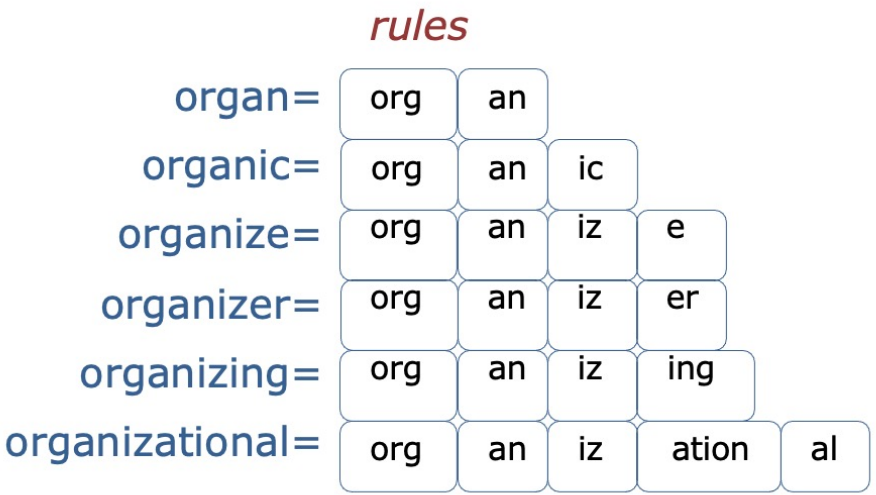
Quiz

Some stemmers will reduce both {*organize*, *organizing*, *organizational*, *organizer*} and {*organ*, *organic*} to the common core of ***org-***. Is this a desirable result (are the two groups of words related)? How does a stemmer arrive at this result?

	rules				
organ=	org	an			
organic=	org	an	ic		
organize=	org	an	iz	e	
organizer=	org	an	iz	er	
organizing=	org	an	iz	ing	
organizational=	org	an	iz	ation	al

Quiz

Some stemmers will reduce both {*organize*, *organizing*, *organizational*, *organizer*} and {*organ*, *organic*} to the common core of ***org-***. Is this a desirable result (are the two groups of words related)? How does a stemmer arrive at this result?



E.g., *Italy* → *Italian* and *history* → *historian*

E.g., *acid* → *acidic*

E.g., *modern* → *modernize*

E.g., *produce* → *producer*

E.g., *make* → *making*

E.g., *sense* → *sensational*

Text Preprocessing Techniques: **Lemmatization**

It stems the word but makes sure that it does not lose its meaning. Lemmatization has a pre-defined dictionary that stores the context of words and checks the word in the dictionary while diminishing.

- The difference between Stemming and Lemmatization can be understood with the example provided below:

Original Word	After Stemming	After Lemmatization
goose	goos	goose
geese	gees	goose
running	run	run
went	went	go

Text Preprocessing Techniques: **Lemmatization**

WordNetLemmatizer Lemmatization

```
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
lemmatizer = WordNetLemmatizer()

def lemma_words(text):
    word_tokens = word_tokenize(text)
    lemmas = [lemmatizer.lemmatize(word) for word in word_tokens]
    return lemmas

input_str = "data science uses scientific methods algorithms
            and many types of processes"
lemma_words(input_str)
```

```
Output:

['data','science','us’, 'scientific’, 'method’, 'algorithm',
 'and’, 'many’, 'type’, 'of’, 'process']
```


Text Preprocessing Techniques: **Lemmatization**

WordNetLemmatizer Lemmatization

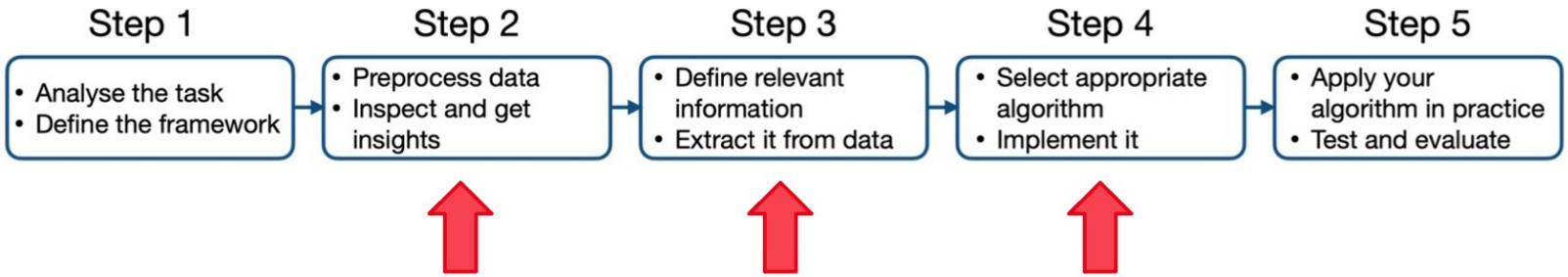
```
[13] import spacy
     nlp = spacy.load('en_core_web_sm')

[14] input_txt='data science uses scientific methods algorithms and many types of processes'

[17] doc1 = nlp(input_txt)

     token_text=[]
     token_lemma=[]

     for token in doc1:
         token_text.append(token.text)
         token_lemma.append(token.lemma_)

     print(token_text)
     print(token_lemma)

     ['data', 'science', 'uses', 'scientific', 'methods', 'algorithms', 'and', 'many', 'types', 'of', 'processes']
     ['datum', 'science', 'use', 'scientific', 'method', 'algorithm', 'and', 'many', 'type', 'of', 'process']
```


Conclusion: Stemming vs lemmatization

- Both have **pros** and **cons**
- The choice in practice **depends** on the available data and application
- For **IR**, since it can be very computationally expensive, the benefits in dimensionality reduction achieved by the use of stemming are typically higher than potential errors (they are not that prevalent) ⇒ **stemmer** is a reasonable choice for morphological preprocessing
- All popular NLP toolkits contain a suite of **stemmers** and **lemmatizers**. Even though none of these tools are perfect, they yield overall good results and are easy to deploy out-of-the-box. Still, it is good to understand what the output contains, how it is produced, and what can be improved

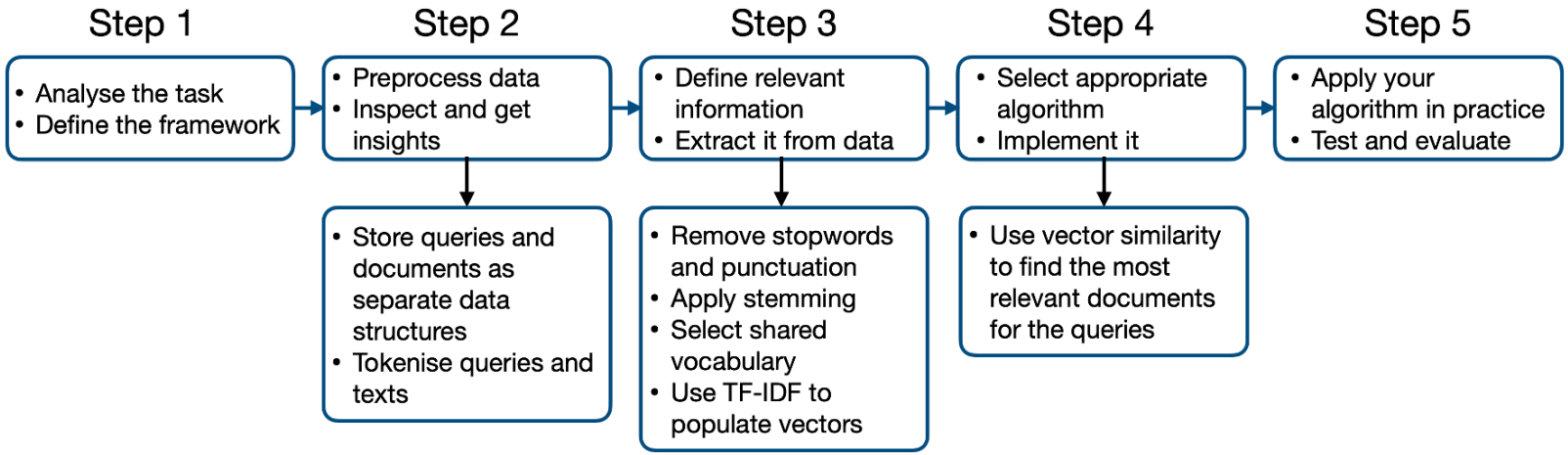
Developing an end-to-end IR application

NLP Pipeline



- Given what we've discussed so far in this lecture, how will you approach each step?
- What techniques and methods will you apply?

NLP Pipeline



Task Setting

- Suppose you are provided with a **collection** of documents and a **set** of queries
- Suppose you also have access to **labels** indicating which documents are relevant for which queries: *labels* = {$query_x$: doc_{x1}, doc_{x2}, …, doc_{xn}; $query_y$: doc_{y1}, doc_{y2}, … doc_{ym}; …}, i.e., each query is mapped to the set of relevant documents' ids
- The **goal** of your application is to identify ids of the most relevant documents to each query: e.g., it may return some *n* (3 or 10) most relevant documents

Step 2 : Preprocessing

- Store the list of queries and documents in **clearly defined data structures**, allowing the algorithm to identify each query and each document by their ids: e.g., you can represent a set of queries as a Python dictionary, mapping keys (each query id) to values (query's content); same applies to the collection of documents
- **Preprocess the raw content** of both documents and queries: for that, use tokenization from Week 1

Step 3 : Identification of relevant information

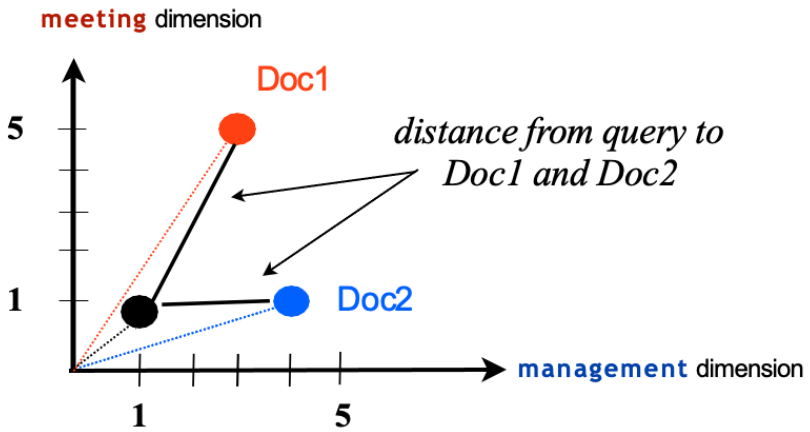
- **Select informative terms**:
    - remove stopwords (there are predefined lists in NLP toolkits)
    - remove punctuation marks (this can be done using Python functionality)
- **Build a shared vector space** with words defining the dimensions:
    - the dimensions should be shared across all documents and queries
    - it is expected that each vector with have some 0's in cells representing words that are not present in this query or document

Step 3 : Identification of relevant information

- Note that **dimensionality** is a problem so **reduce** it:
  - you have a choice between stemming and lemmatization
  - stemming may be more effective in this context
- **Fill in** the query and document **vectors** with appropriate values:
  - apply tf-idf technique to both queries and documents
  - note that it is also possible to apply different weighting schemes to separately queries and documents

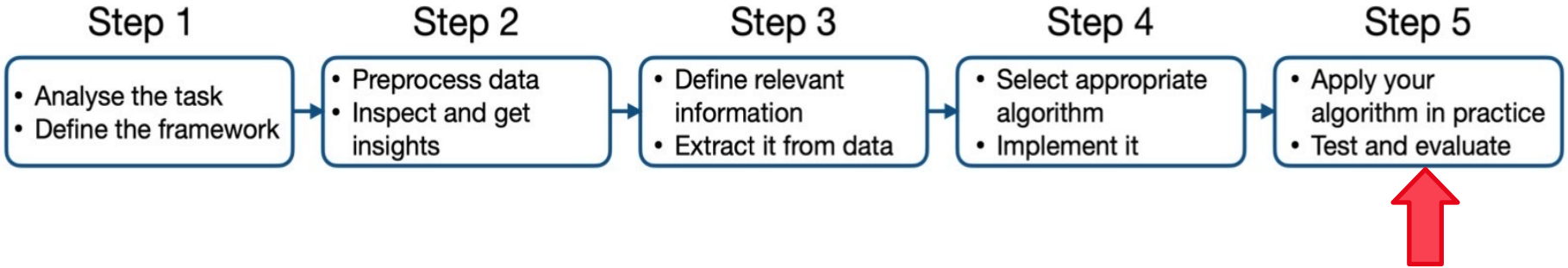
Step 4 : Application of the algorithm

- **Apply cosine similarity** to calculate the similarity between each query vector and each document vector
- **Select the top *n*** (e.g., Google Search shows 10 results per page) most relevant documents for each query



Evaluating your application

Let's consider the NLP pipeline once again



- Once all the steps are implemented, how do you evaluate your IR algorithm?

Before length normalization:

Let’s define the priorties

- Some queries are **very general** (e.g., “What is information retrieval?”) and result in many hits
- Other queries are **quite specific** (e.g., “What is the TF-IDF technique?”) – they will result in fewer hits
- Does the **total number** of potentially relevant documents matter?
- Is your goal to find **all** relevant documents? (Which ML metric is this?)
- What matters in practice is whether you **present the most relevant results first** (how often does the user need to go to the second page of the search results)

Precision @k

- Recall that the algorithm measures cosine similarity
- This means that you can **order (or rank)** the documents by their similarity and introduce the **cut-off point** (e.g., the *top-3* or *top-10* documents)
- Let's call this cut-off point k: e.g., *k=3* or *k=10*
- If you have access to the **gold standard** (list of all documents relevant to the query), you can measure how many of the relevant documents are included in the top-k documents returned by the algorithm – these are called **true positives**

Precision @k

- The measure that tells you what proportion of true positives is contained in your system's output is called **precision**, and for the cut-off of top-k it's called **precision@k** (for example, *P@3* or *P@10*):

  $$P@k = (TP \text{ among top } k) / k$$

  where

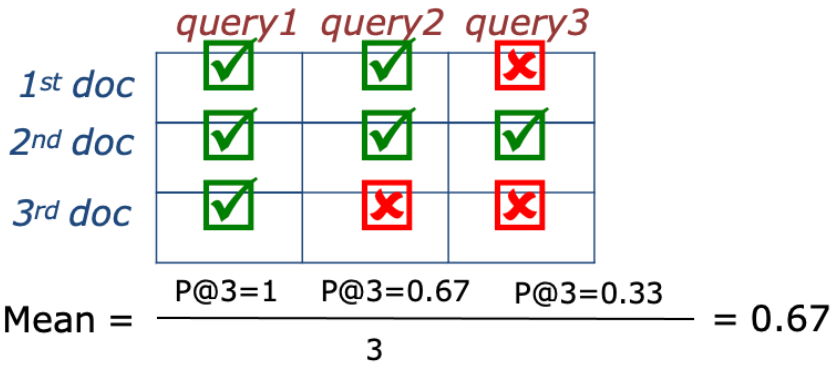
  TP among top k = number of documents among top k that are actually relevant

- For example, if 8 documents in the top-10 ones returned by your algorithm are relevant, P@10=0.8

Mean Precision @k

- Note that precision takes values between 0 (no relevant documents among the returned top-k) to 1 (all of them are relevant)
- You are not interested in the results of your system on a single query, rather across all queries, so report **mean P@k**:

$$\text{Mean P@k} = \Sigma_i (P_i@k) \,/\, (\text{number of queries})$$



Mean Reciprocal Rank (MRR)

- **Mean Reciprocal Rank (MRR)** measures how high, on the average, the algorithm places the first relevant document that it returns
- I.e., if you were looking for the information on Google, roughly how often will you be satisfied with the very 1st webpage returned? How often will you need to check the 2nd one?
- The **rank** of the document is its place in the ordered list of results: i.e., the first one on the sorted list has the rank of 1

Mean Reciprocal Rank (MRR)

- **Reciprocal rank (RR)** is defined as:

  RR = 1 / rank of the first relevant document in the ordered list

- Then, the **mean reciprocal rank (MRR)** is:

$$MRR = \sum_i (RR_i) / (\text{number of queries})$$

$$MRR = \frac{1 + 1 + 1/2}{3} = 0.83$$

Part of Speech (POS)

Text Preprocessing Techniques: **Part-of-speech tagging**

Part-of-speech tagging is used to identify the part of speech for each word in a sentence, such as nouns, verbs, adjectives, adverbs, and more.

```
# Sample text
text = "NLTK is a powerful library for natural language
          processing."
# Performing PoS tagging
pos_tags = pos_tag(words)
# Displaying the PoS tagged result in separate lines
for word, pos_tag in pos_tags:
    print(f"{word}: {pos_tag}")
```

```
Output:

PoS Tagging Result:
    NLTK: NNP
    is: VBZ
    a: DT
    powerful: JJ
    library: NN
    for: IN
    natural: JJ
    language: NN
    processing: NN
```

Currently, NLTK pos_tag only supports English and Russian (i.e. lang='eng' or lang='rus')

Named Entity Recognition (NER)

Text Preprocessing Techniques: **Named entity recognition (NER)**

Named Entity Recognition (NER) identifies and classifies named entities in text, such as people, organizations, lowomenions, etc..

```
from nltk import ne_chunk
from nltk.tokenize import word_tokenize
# Sample text
text = "Barack Obama was the 44th President of the United States."
words = word_tokenize(text)

# Performing Named Entity Recognition (NER)
ner_tags = ne_chunk(nltk.pos_tag(words))

# Displaying the NER tagged result in separate lines
for chunk in ner_tags:
    if hasattr(chunk, 'label'):
        print(f"{chunk.label()}: {' '.join(c[0] for c in chunk)}")
```

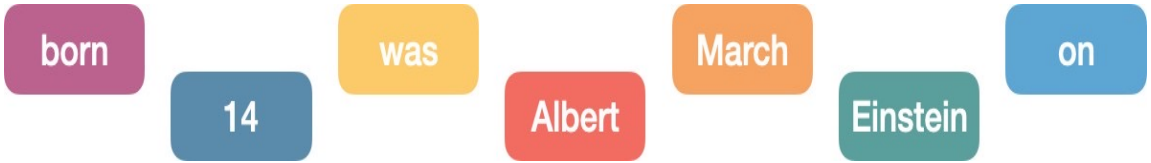
```
Output:

PERSON: Barack
PERSON: Obama
GPE: United States
```

Sequence modeling and labelling

When the order matters

- However, texts are never just random collections of disconnected words
- Language has a clear inherent structure, with language rules governing the way words are put together in sentences and sentences are put together in larger units

Who did what [to whom] [when] […]?



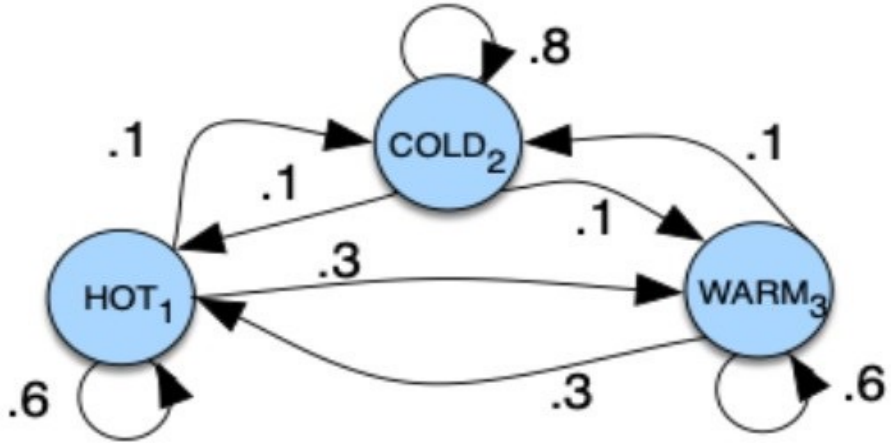
When the order matters

- However, texts are never just random collections of disconnected words
- Language has a clear inherent structure, with language rules governing the way words are put together in sentences and sentences are put together in larger units

Who did what [to whom] [when] […]?

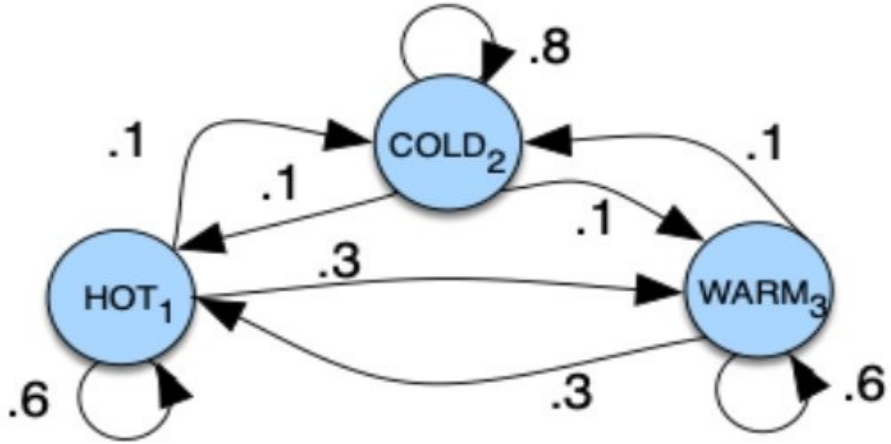
Sequence modeling: Weather example

- Weather changes can be modelled **probabilistically**: e.g., it is more likely that a cold day is followed by another cold day than immediately by a hot one
- Let's represent this with a **directed graph**
- Vertices are **states**, and edges with associated probabilities are **transitions**



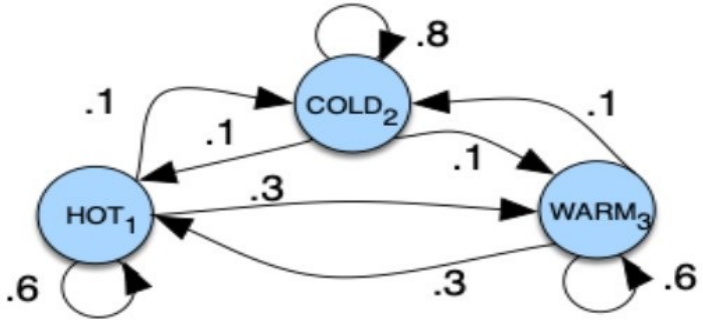
Markov chain weather example

- The graph visualises a **Markov chain**
- A Markov chain is a model that defines the probabilities of sequences of random variables (states), which come from a predefined set
- E.g., $s_{weather}$ = [s_1=*hot*, s_2=*cold*, s_3=*warm*]



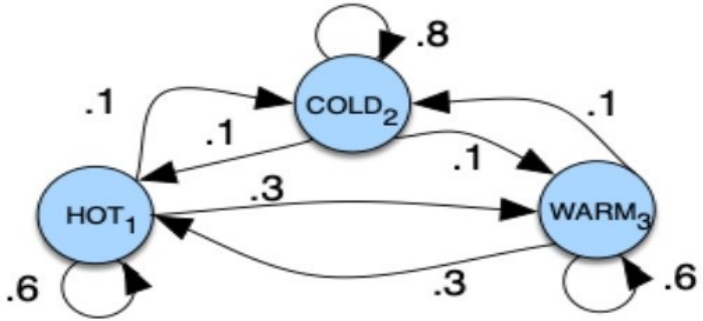
Markov chain weather example

- The chain also defines **transition probabilities** a_i for each state *i*
- E.g., a_1 = [0.6, 0.1, 0.3]
- Interpretation:
  - P(s_1=*hot* → s_1=*hot*) = 0.6
  - P(s_1=*hot* → s_2=*cold*) = 0.1
  - P(s_1=*hot* → s_3=*warm*) = 0.3



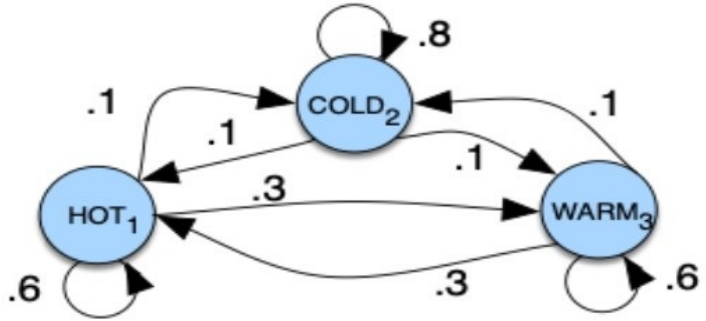
Markov chain weather example

- Finally, let's define a **start distribution** π
- E.g., π = [0.1, 0.7, 0.2] (note that $\sum_i \pi_i = 1$)
- Interpretation:
  - P(starting at s_1=*hot*) = 0.1
  - P(starting at s_2=*cold*) = 0.7
  - P(starting at s_3=*warm*) = 0.2



Markov (Independence) assumption

- **First-order** Markov chain: when predicting the future, the past does not matter, only the present
    - $P(s_i=a \mid s_1 \ldots s_{i-1}) = P(s_i=a \mid s_{i-1})$
- **Interpretation**: to predict the weather for tomorrow, you only need to consider the weather for today
- The order of the model defines the length of the context

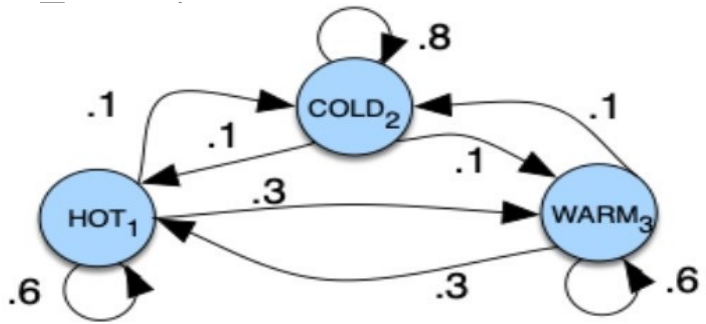


Markov model components

- $S=s_1\ s_2\ \ldots\ s_N$ – a set of *N* **states**
- $A= a_{11}\ a_{12}\ \ldots\ a_{N1}\ \ldots\ a_{NN}$ – a **transition probability matrix** *A* with a_{ij} representing the probability of moving from state s_i to state s_j:

$$\sum_j a_{ij} = 1\ \forall i$$

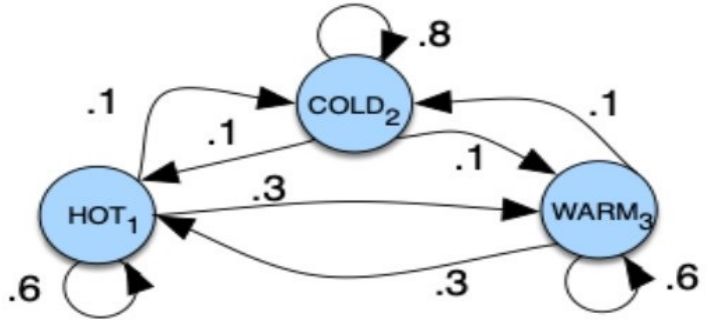
- $\pi = \pi_1, \pi_2, \ldots, \pi_N$ – an **initial probability distribution** over states



Markov model components

- π = [0.1, 0.7, 0.2]
- A:

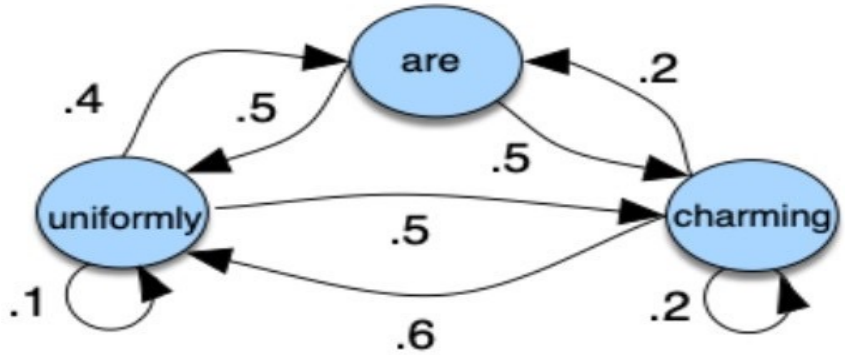
	s_1 = hot	s_2 = cold	s_3 = warm
s_1 = hot	0.6	0.1	0.3
s_2 = cold	0.1	0.8	0.1
s_3 = warm	0.3	0.1	0.6



How does this apply to language?

- π = [0.1, 0.7, 0.2]
- A:

	uniformly	are	charming
uniformly	0.1	0.4	0.5
are	0.5	0.0	0.5
charming	0.6	0.2	0.2



How does this apply to language?

- π = [0.1, **0.7**, 0.2]
- A:

	uniformly	are	charming
uniformly	0.1	0.4	**0.5**
are	**0.5**	0.0	0.5
charming	0.6	0.2	0.2

P("are uniformly charming")? = P(start with "are")
× P("are" → "uniformly")
× P("uniformly" → "charming")
= 0.7 × 0.5 × 0.5 = 0.175

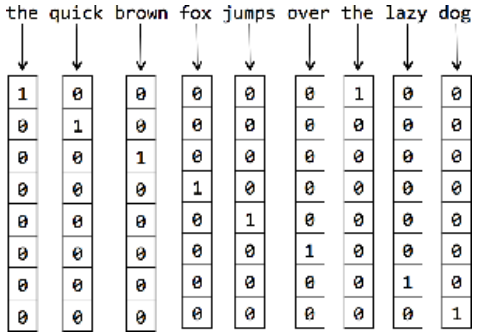
Word Embeddings

Word Embedding in NLP

Word Embeddings are numeric representations of words in a lower-dimensional space, capturing semantic and syntactic information. They play a vital role in Natural Language Processing (NLP) tasks.

- Word Embedding is an approach for representing words and documents. Word Embedding or Word Vector is a numeric vector input that represents a word in a lower-dimensional space. It allows words with similar meanings to have a similar representation.
- Word Embeddings are a method of extracting features out of text so that we can input those features into a machine learning model to work with text data. They try to preserve syntactical and semantic information.
- Approaches for Text Representation

1. **Traditional Approach**
2. **Neural Approach**
3. **Pretrained Word-Embedding**



Text Representation Problem

The Problem

- Given a supervised learning task to predict which tweets are about real disasters and which ones are not (classification). The independent variable would be the tweets (text) and the target variable would be the binary values (1: Real Disaster, 0: Not real Disaster).
- Machine Learning and Deep Learning algorithms only take numeric input. **So, how do we convert tweets to their numeric values?**

The Solution

- Word Embeddings in NLP is a technique where individual words are represented as real-valued vectors in a lower-dimensional space and captures inter-word semantics.
- Each word is represented by a real-valued vector with tens or hundreds of dimensions.

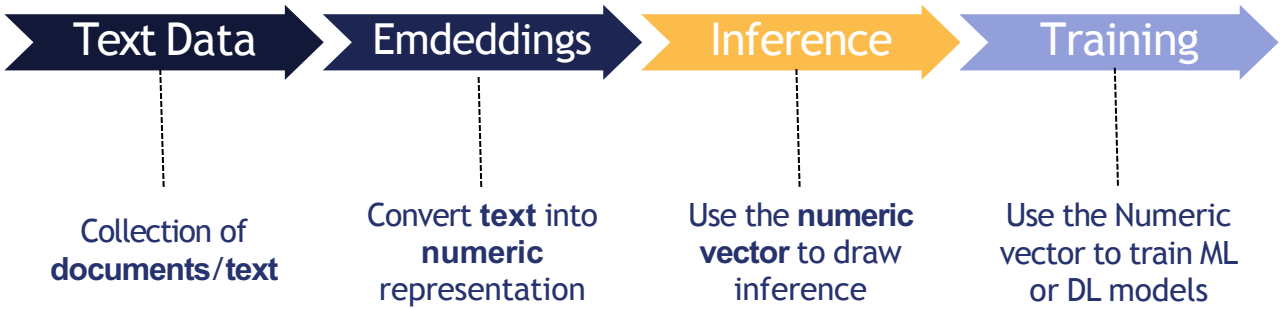
Why & How to use Word Embeddings

Why are Word Embedding needed?

- Dimensionality reduction
- Predictive modeling of surrounding words
- Capturing inter-word semantics.
- Facilitating efficient computational processing
- Enhancing language modeling and text generation capabilities

How are Word Embeddings used?

- Word Embeddings are used as input to machine learning models.



Traditional Approach for Text Representation

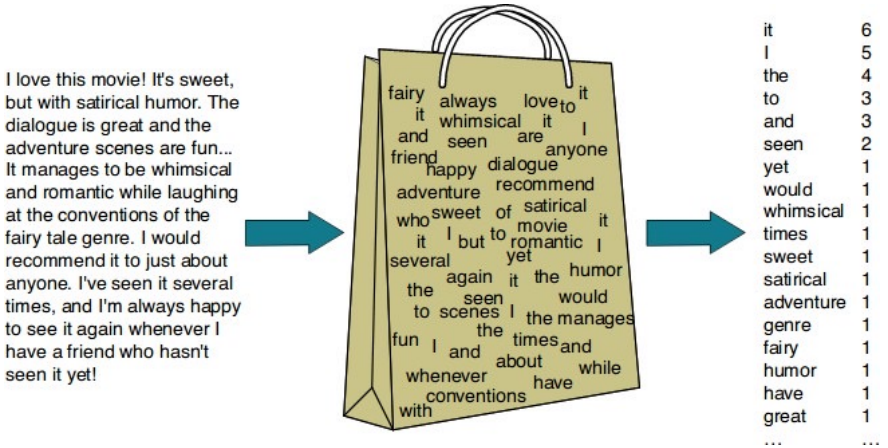
Text Data Conversion: From Words to Numbers

After all the text processing steps are performed, the final acquired data is converted into the numeric form using Bag of words or TF-IDF.

Machine learning algorithms cannot work with the raw text directly; the **text** must be converted into **numbers**. Specifically, vectors of numbers.

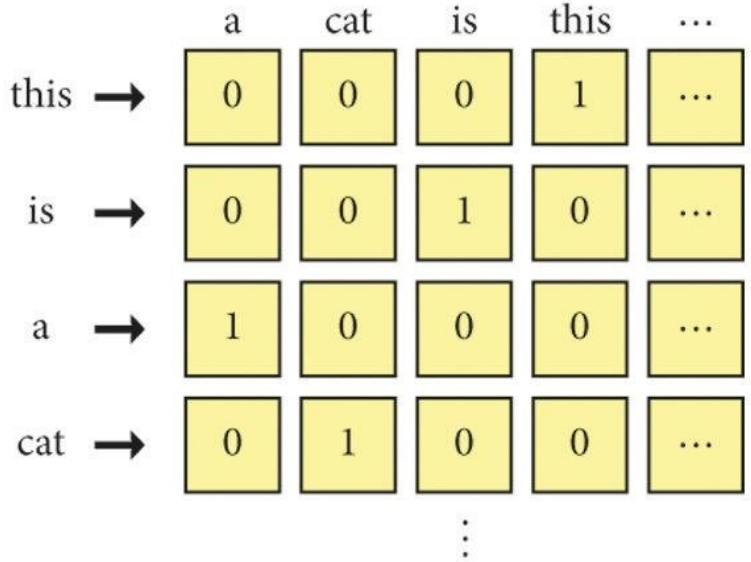
Popular and simple method of feature extraction with text data which are currently used are:

1. **One-Hot Encoding**
2. **Bag-of-Words**
3. **TF-IDF**



One-Hot Encoding

- One-hot encoding is a simple method for representing words in natural language processing (NLP).
- In this encoding scheme, each word in the vocabulary is represented as a unique vector, where the dimensionality of the vector is equal to the size of the vocabulary.
- The vector has all elements set to 0, except for the element corresponding to the index of the word in the vocabulary, which is set to 1.

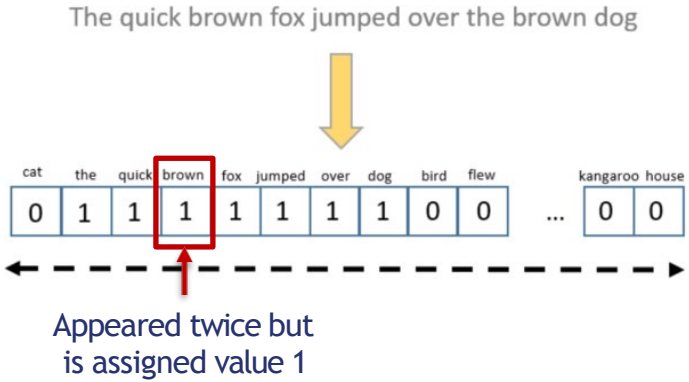


Disadvantaged of One-Hot Encoding

One-hot encoding is a simple and intuitive method for representing words in NLP, it has several disadvantages, which may limit its effectiveness in certain appliwomenions.

- One-hot encoding results in high-dimensional vectors, making it computationally expensive and memory-intensive, especially with large vocabularies.
- It does not capture semantic relationships between words; each word is treated as an isolated entity without considering its meaning or context.
- It is restricted to the vocabulary seen during training, making it unsuitable for handling out-of-vocabulary words.
- It is not suitable for capturing word frequency or importance within the text.

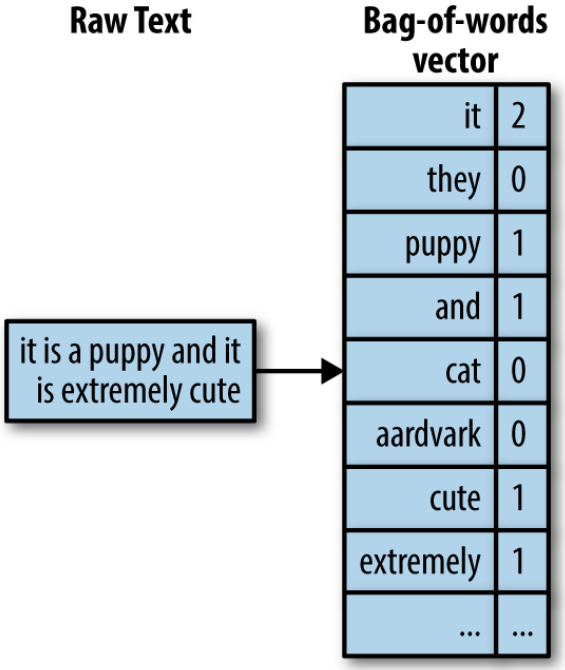
To overcome these shortcomings, Bag-of-Words can be used.



Bag-of-Words

The **Bag-of-Words** model serves as a simplified representation applied in natural language processing (NLP) and information retrieval (IR).

- In this approach, a text, whether it's a sentence or a document, is portrayed as a collection of its words, irrespective of grammar or word sequence, but taking into account word frequency.
- Bag-of-words is widespread in document classification methods, where the frequency of each word serves as a feature for training classifiers.
- Known for its simplicity in comprehension and implementation, the bag-of-words model has proven highly effective in tasks like language modeling and document classification.



Bag-of-Words *(cont'd)*

To create a Bag-of-Words It involves two things:

1. A vocabulary of known words known as Corpus:
    - This step revolves around constructing a document corpus which consists of all the unique words in the whole of the text present in the data provided. It is sort of like a dictionary where each index will correspond to one word and each word is a different dimension.
2. A measure of the presence of known words:
    - After the creation of Corpus, it is used to create sparse vector of d-unique words and for each document, we will fill it will number of times the corresponding word occurs in a document.

Corpus

Sentence	0 (fake)	1 (news)	2 (audience)	3 (encourage)	4 (false)	5 (mentioned)	6 (misleading)
news mentioned fake	1	1	0	0	0	1	0
audience encourage fake news	1	1	1	1	0	0	0
fake news false misleading	1	1	0	0	1	0	1

Occurrence

Bag-of-Words Implementation

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

corpus = [
    "Tune a hyperparameter.",
    "You can tune a piano but you can't tune a fish.",
    "Fish who eat fish, womench fish.",
    "People can tune a fish or a hyperparameter.",
    "It is hard to womench fish and tune it.",]

vectorizer = CountVectorizer(stop_words='english')
X = vectorizer.fit_transform(corpus)
pd.DataFrame(X.A, columns=vectorizer.get_feature_names_out())
```

	catch	eat	fish	hard	hyperparameter	people	piano	tune
0	0	0	0	0	1	0	0	1
1	0	0	1	0	0	0	1	2
2	1	1	3	0	0	0	0	0
3	0	0	1	0	1	1	0	1
4	1	0	1	1	0	0	0	1

Bag-of-Words Drawbacks

Bag of words do have few shortcomings:

1. **No Word Order**: It doesn't care about the order of words, missing out on how words work together.
2. **Ignores Context**: It doesn't understand the meaning of words based on the words around them.
3. **Always Same Length**: It always represents text in the same way, which can be limiting for different types of text.
4. **Lots of Words**: It needs to know every word in a language, which can be a huge list to handle.
5. **No Meanings**: It doesn't understand what words mean, only how often they appear, so it can't grasp synonyms or different word forms.

To overcome these shortcomings, TF-IDF can be used

Term Frequency - Inverse Document Frequency (TF-IDF)

TF-IDF, short for term frequency-inverse document frequency, is a numerical statistic that is intended to reflect **how important a word is to a document in a collection or corpus**.

The TF-IDF value increases proportionally to the number of times a word appears in the Document and is offset by the number of documents in the corpus that contain the word, which helps to adjust for the fact that some words appear more frequently in general.

A **"Document"** is a distinct text. This generally means that each article, book, or so on is its own document.

This concept includes:

1. **Counts**: Count the number of times each word appears in a document.
2. **Frequencies**: Calculate the frequency that each word appears in a document out of all the words in the document.

Text1: Basic Linux Commands for Data Science
Text2: Essential DVC Commands for Data Science

	basic	commands	data	dvc	essential	for	linux	science
Text 1	0.5	0.35	0.35	0.0	0.0	0.35	0.5	0.35
Text 2	0.0	0.35	0.35	0.5	0.5	0.35	0.0	0.35

Term Frequency - Inverse Document Frequency (TF-IDF)

Term Frequency:

- Term Frequent (TF) is a measure of how frequently a term, t, appears in a document, d.
- TF can be said as what is the probability of finding a word in a document.

$$tf_{t,d} = \frac{n_{t,d}}{Number\ of\ terms\ in\ the\ document}$$

- The numerator, n is the number of times the term "t" appears in the document "d". Thus, each **document** and **term** would have its own TF value.
- Let's take an example to get a clearer understanding:
  - **Sentence 1: The car is driven on the road.**
  - **Sentence 2: The truck is driven on the highway.**

Step 1: TF Calculation

$tf_{Car,Sent\ 1} = \frac{1}{7}$ $tf_{road,Sent\ 2} = \frac{1}{7}$

Term Frequency - Inverse Document Frequency (TF-IDF)

Inverse Document frequency:

- The inverse document frequency is a measure of how much information the word provides, i.e., if it's common or rare across all documents.
- We need the IDF value because computing just the TF alone is not sufficient to understand the importance of words.

$$idf_t = \log \frac{number\ of\ documents}{number\ of\ documents\ containing\ the\ term\ 't'}$$

- In inverse document frequency (IDF), if a word is observed in all documents *(a.k.a a common word)*, its **idf-score is zero** as the logarithm of 1 is zero.
- Let's complete the previous example:

Step 2: IDF Calculation

$idf_{Car,Sent\ 1} = log\frac{2}{1} = 0.3$ $\quad$ $idf_{road,Sent\ 2} = log\frac{2}{1} = 0.3$

Term Frequency - Inverse Document Frequency (TF-IDF)

Term Frequency – Inverse Document frequency:

- A high TF-IDF weight is achieved when a term has a high frequency within a specific document and a low frequency across all documents, effectively filtering out common terms.
- The IDF's log function yields a value greater than or equal to 0, as the ratio inside it is always greater than or equal to 1; as a term appears in more documents, the IDF and TF-IDF values approach 0.
- TF-IDF assigns larger values to less frequent words in the document corpus, with high values resulting from both high IDF and TF, indicating rare occurrence across the entire document corpus but frequent presence within a specific document.

$$tf_{t,d} = \frac{n_{t,d}}{Number\ of\ terms\ in\ the\ document} \qquad idf_t = \log \frac{number\ of\ documents}{number\ of\ documents\ containing\ the\ term\ 't'}$$

$$tf\_idf(t,d,D) = tf(t,d) \cdot idf(t,D)$$

Term Frequency - Inverse Document Frequency (TF-IDF)

Full example:

The previous Example which has the consists of the following sentences:

- **Sentence 1: The car is driven on the road.**
- **Sentence 2: The truck is driven on the highway.**

In this example, each sentence is a separate document. To calculate TF-IDF for the above two documents, which represent the corpus.

- **TF** : *number of times the term appears in the doc/total number of words in the doc*
- **IDF** : *ln(number of docs/number docs the term appears in)*
- **TFIDF** is the product of the TF and IDF scores of the term

Higher the TFIDF score, the rarer the term is and vice-versa.

Word	TF		IDF	TF*IDF	
	A	B		A	B
The	1/7	1/7	log(2/2) = 0	0	0
Car	1/7	0	log(2/1) = 0.3	0.043	0
Truck	0	1/7	log(2/1) = 0.3	0	0.043
Is	1/7	1/7	log(2/2) = 0	0	0
Driven	1/7	1/7	log(2/2) = 0	0	0
On	1/7	1/7	log(2/2) = 0	0	0
The	1/7	1/7	log(2/2) = 0	0	0
Road	1/7	0	log(2/1) = 0.3	0.043	0
Highway	0	1/7	log(2/1) = 0.3	0	0.043

TF-IDF Implementation

```
from sklearn.feature_extraction.text import
TfidfVectorizer
corpus = [
    "Tune a hyperparameter.",
    "You can tune a piano but you can't tune a fish.",
    "Fish who eat fish, womench fish.",
    "People can tune a fish or a hyperparameter.",
    "It is hard to womench fish and tune it.",]

vectorizer = TfidfVectorizer(stop_words='english')
X = vectorizer.fit_transform(corpus)
df = pd.DataFrame(np.round(X.A,3),
columns=vectorizer.get_feature_names_out())
df
```

	catch	eat	fish	hard	hyperparameter	people	piano	tune
0	0.000	0.000	0.000	0.000	0.820	0.000	0.000	0.573
1	0.000	0.000	0.350	0.000	0.000	0.000	0.622	0.701
2	0.380	0.471	0.796	0.000	0.000	0.000	0.000	0.000
3	0.000	0.000	0.373	0.000	0.534	0.661	0.000	0.373
4	0.534	0.000	0.373	0.661	0.000	0.000	0.000	0.373

TF-IDF Drawbacks

TF-IDF, while commonly employed in information retrieval and text mining, has certain limitations, particularly in tasks requiring nuanced language comprehension:

1. **Lack of Semantic Understanding:** TF-IDF treats words independently, neglecting their semantic relationships and context, which limits its ability to grasp deeper language semantics.
2. **Sensitivity to Document Length:** Longer documents may have higher term frequencies, potentially biasing TF-IDF towards longer texts and impacting its effectiveness in capturing term importance.
3. **Difficulty with Rare Terms:** TF-IDF may struggle to accurately assess the importance of rare terms due to their low frequency in the corpus, affecting the representation of less common but significant words.
4. **Vulnerability to Noise:** Noise or irrelevant terms in the document can distort TF-IDF scores, leading to less accurate representations of document content and potentially affecting downstream tasks.
5. **Lack of Inter-document Relationships:** TF-IDF does not consider relationships between documents, which can be crucial for tasks such as document clustering or topic modeling, where understanding document similarity is essential.

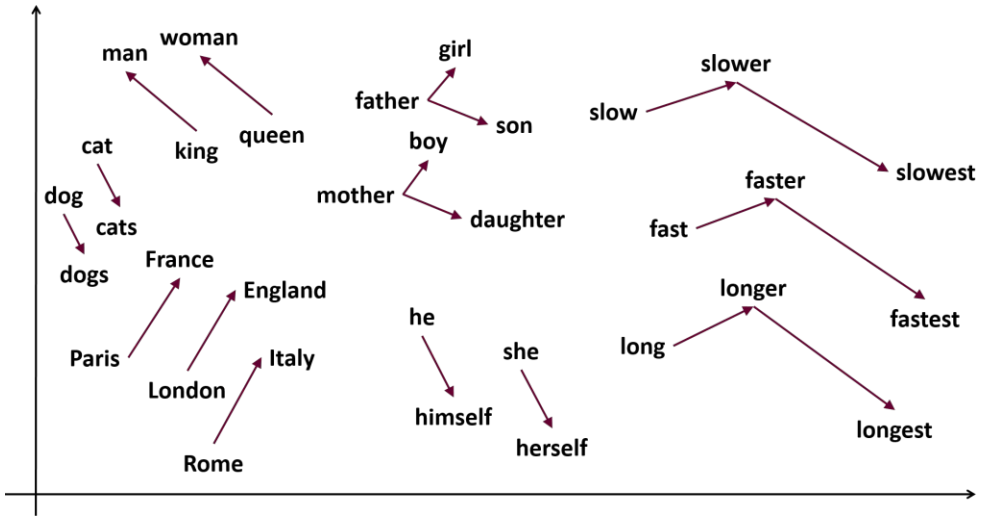
To overcome these shortcomings, Word2Vec can be used

Neural Approach for Text Representation

Word Embedding

Word Embedding is a language modeling technique for mapping words to vectors of real numbers. It represents words or phrases in vector space with several dimensions.

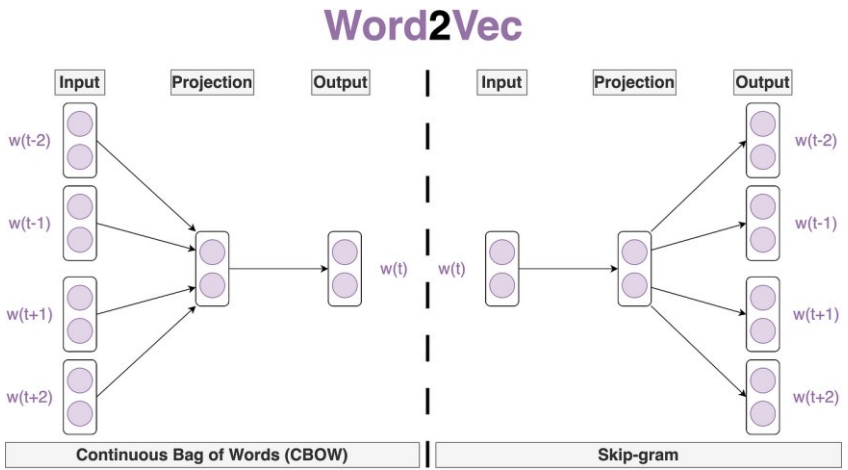
- Word embeddings can be generated using various methods like neural networks, co-occurrence matrices, probabilistic models, etc.
- Word2Vec consists of models for generating word embedding. These models are shallow two-layer neural networks having one input layer, one hidden layer, and one output layer.
- **Word2vec was created, patented, and published in 2013 by a team of researchers led by Tomas Mikolov at Google.**



Word2Vec

Word2Vec is a widely used method in natural language processing (NLP) that allows words to be represented as vectors in a continuous vector space.

- Word2Vec is an effort to map words to high-dimensional vectors to capture the semantic relationships between words, developed by researchers at Google.
- For example, it can identify relations like country-capital over larger datasets showing us how powerful word embeddings can be.
- Words with similar meanings should have similar vector representations, according to the main principle of Word2Vec.
- There are two neural embedding methods for Word2Vec:
  1. **CBOW (Continuous Bag of Words)**
  2. **Skip Gram**

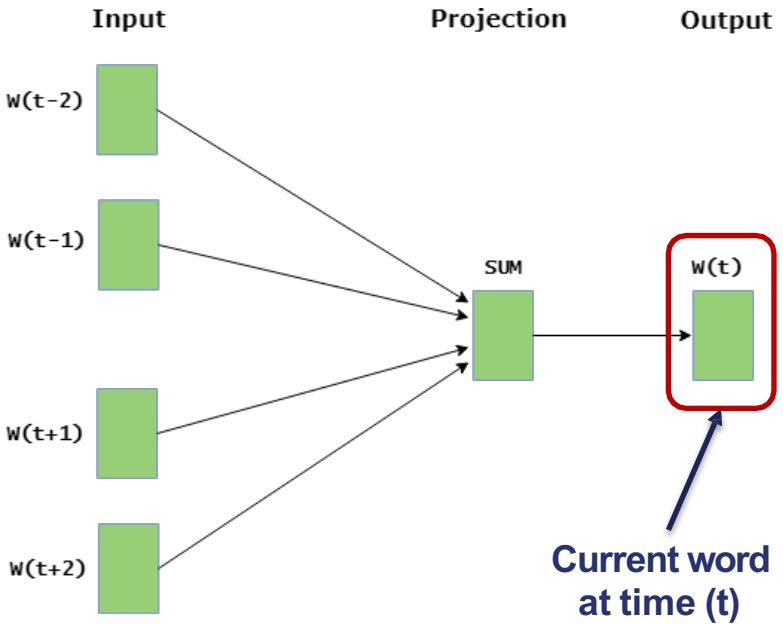


Word2Vec: **CBOW (Continuous Bag of Words)**

1. **CBOW (Continuous Bag of Words)**

The CBOW model predicts the current word given context words within a specific window.

- The **input layer** contains the context words.
- The **output layer** contains the current word.
- The **hidden layer** contains the dimensions we want to represent the current word present at the output layer.

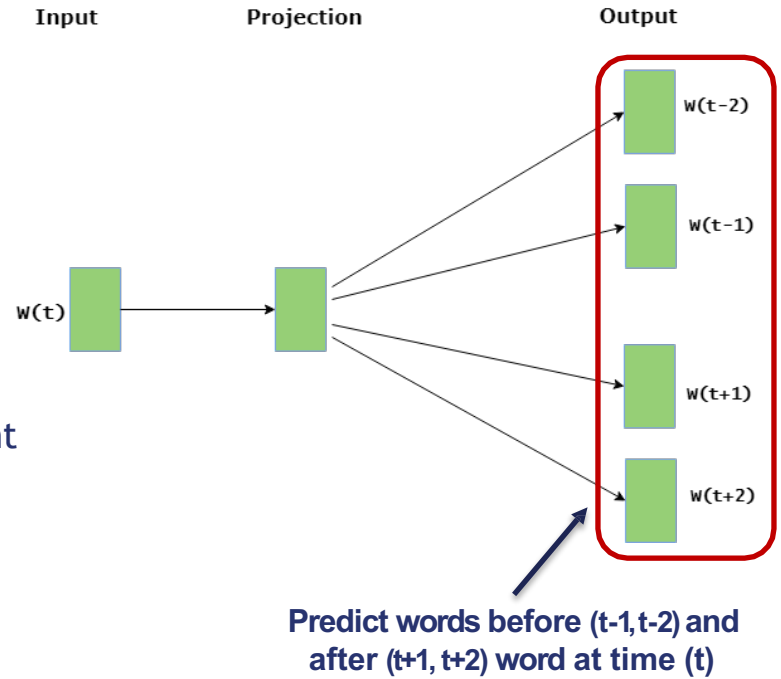


Word2Vec: **Skip Gram**

2. **Skip Gram**

Skip gram predicts the surrounding context words within specific window given current word.

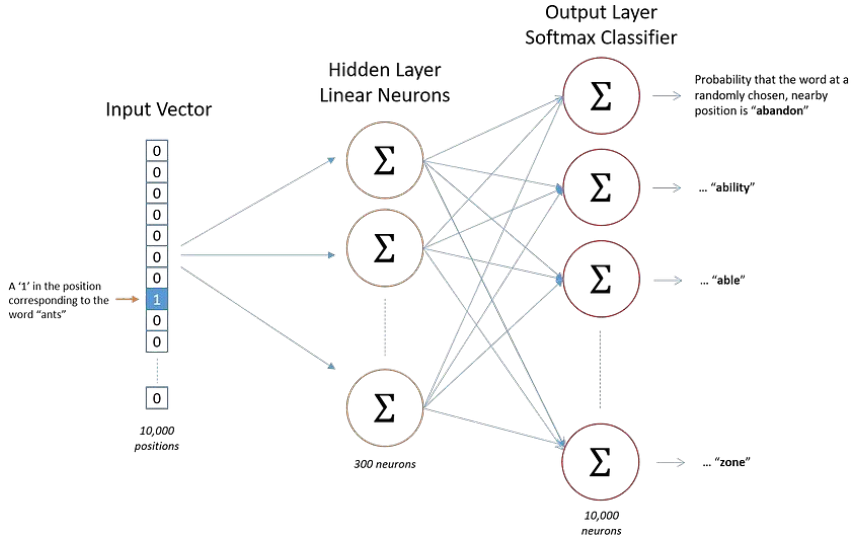
- The **input layer** contains the current word.
- The **output layer** contains the context words.
- The **hidden layer** contains the number of dimensions in which we want to represent current word present at the input layer.



Word2Vec Model Architecture

Word2Vec essentially is a shallow 2-layer neural network trained.

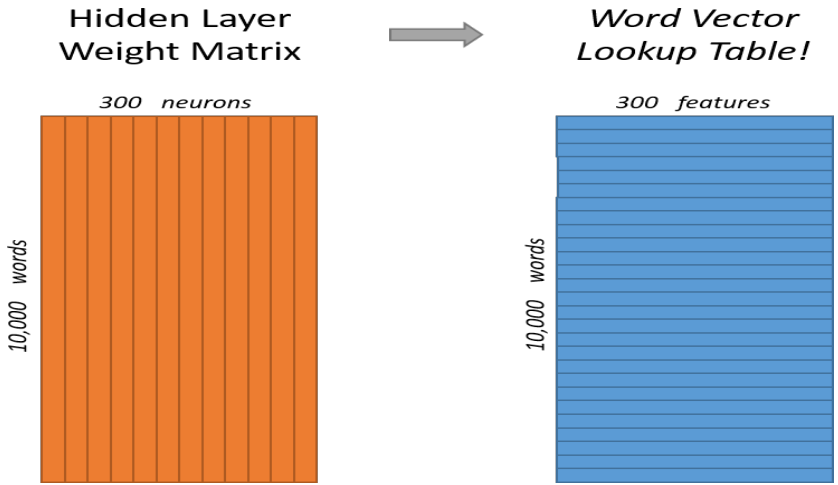
- The input contains all the documents/texts in our training set.
- For the network to process these texts, they are represented in a 1-hot encoding of the words.
- The number of neurons present in the hidden layer is equal to the length of the embedding you want.
- That is, if we want all our words to be vectors of length 300, then the hidden layer will contain 300 neurons.



Word2Vec Model Architecture *(cont'd)*

The output layer contains probabilities for a target word (given an input to the model, what word is expected) given a particular input.

- At the end of the training process, the hidden weights are treated as the word embedding.
- Intuitively, this can be thought of as each word having a set of n weights *(300 considering the example)* "weighing" their different characteristics.



Implementing Word2Vec

1. **Training the embeddings:**

- Word2Vec is imported from the gensim.model library. Each input to the model must be a list of phrases, so the input to the model is generated by using the split() function on each line in the corpus of texts.
- The model is then set up with various parameters, with a brief explanation of their meanings:

    1. **Size** refers to the size of the word embedding that it would output.
    2. **Window** refers to the maximum distance between the current and predicted word within a sentence.
    3. The **min_count** parameter is used to set a minimum frequency for the words to be a part of the model, i.e. it ignores all words with count less than **min_count**.
    4. **iter** refers to the number of iterations for training the model.

Implementing Word2Vec *(cont'd)*

1. **Training the embeddings:**

```
# Suppose we have a paragraph of text related to Saudi Vision 2030
# We tokenize the paragraph into sentences and store them in a list called 'sentences'
paragraph = "Saudi Vision 2030 is a strategic framework aimed at reducing Saudi Arabia's dependence on oil, \
             diversifying its economy and developing public service sectors. It encompasses various initiatives \
             to achieve long-term goals such as promoting tourism, enhancing eduwomenion, and fostering innovation."
sentences = nltk.sent_tokenize(paragraph)

# Tokenize each sentence into words and store them in a list of lists called 'tokenized_sentences'
tokenized_sentences = [nltk.word_tokenize(sentence) for sentence in sentences]

# We train a Word2Vec model on the 'sentences' data
# The model has parameters like embedding size of 100, window size of 5, and is trained for 10 iterations
from gensim.models import Word2Vec

# Create CBOW model
w2v_CBOW = Word2Vec(tokenized_sentences, size= 20, min_count=1, window=5, iter=10)

# Create Skip Gram model
w2v_SG = Word2Vec(tokenized_sentences, min_count=1, size=20, window=5, sg=1)
```


Implementing Word2Vec *(cont'd)*

2. **Using the word2vec model:**

- Finding the vocabulary of the model can be useful in several general applications, and in this case, provides us with a list of words we can try and use other functions.
- Finding the embedding of a given word can be useful when we're trying to represent sentences as a collection of word embeddings, like when we're trying to make a weight matrix for the embedding layer of a network. I included this so that it can help your intuition of what the word vector looks like.
- We can also find out the similarity between given words *(the cosine distance between their vectors)*. Here we have tried '**goals**' and '**eduwomenion'** and compared it with CBOW and Skip-Gram seeing how a stark distinction exists.

Implementing Word2Vec *(cont'd)*

2. **Using the word2vec model:**

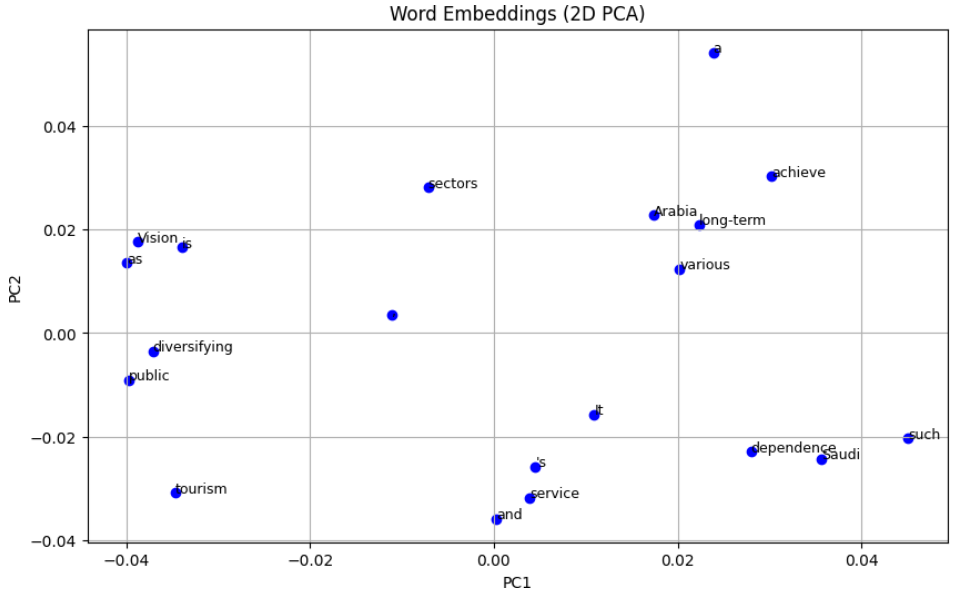
```
print("Cosine similarity between goals' " +
      "and eduwomenion' - CBOW : ",
      w2v_CBOW.wv.similarity('goals', 'eduwomenion'))

print("Cosine similarity between goals' " +
      "and 'eduwomenion' - Skip Gram : ",
      w2v_SG.wv.similarity('goals', 'eduwomenion'))
```

```
Output:

Cosine similarity between 'goals ' and 'eduwomenion ' - CBOW :   0.12904271
Cosine similarity between 'goals ' and 'eduwomenion ' - Skip Gram :   0.12125311
```


Implementing Word2Vec *(cont'd)*

3. **Visualizing word embeddings:**

- Word2Vec word embedding can usually be of sizes 100 or 300, and it is practically not possible to visualize a 300- or 100-dimensional space with meaningful outputs.
- In either case, it uses PCA to reduce the dimensionality and represent the word through their vectors on a 2-dimensional plane.
- **The actual values of the axis are not of concern as they do not hold any significance, rather we can use it to perceive similar vectors with respect to each other.**



Implementing Word2Vec *(cont'd)*

3. Visualizing word embeddings:

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

def plot_word_embeddings(model, words):
    word_vectors = np.array([model.wv[word] for word in words])
    pca = PCA(n_components=2)
    word_embeddings_2d = pca.fit_transform(word_vectors)

    plt.figure(figsize=(10, 6))
    for i, word in enumerate(words):
        x, y = word_embeddings_2d[i]
        plt.swomenter(x, y, marker='o', color='blue')
        plt.text(x, y, word, fontsize=9)

    plt.title('Word Embeddings (2D PCA)')
    plt.xlabel('PC1')
    plt.ylabel('PC2')
# Let's plot the embeddings of 20 random words from the model
random_words = np.random.choice(list(w2v_CBOW.wv.vocab.keys()), size=20, replace=False)
plot_word_embeddings(w2v_CBOW, random_words)
```


Implementing Word2Vec *(cont'd)*

3. Word Embeddings DataFrame

```
vector = pd.DataFrame(w2v_SG.wv.vectors)
vector.columns = list(w2v_SG.wv.vocab.keys())
vector.index = list(w2v_SG.wv.vocab.keys())
vector.head()
```

	Saudi	Vision	2030	is	a	strategic	framework	aimed	at	reducing	...
Saudi	-0.010134	-0.005619	-0.000848	-0.005767	-0.005605	-0.011439	-0.006764	0.005265	-0.008947	0.003225	...
Vision	-0.009512	0.002805	-0.002119	-0.006504	-0.008407	-0.001648	-0.000456	-0.004755	-0.002717	-0.002775	...
2030	0.011752	0.008253	-0.003374	0.006506	-0.007851	-0.010551	-0.011443	0.012018	-0.002569	-0.009787	...
is	-0.004154	-0.000653	-0.012131	-0.001399	-0.001966	0.004273	0.001662	0.003115	-0.011905	-0.006971	...
a	0.000704	0.007078	-0.012078	-0.00483	0.000472	0.002312	-0.010609	0.005597	0.001601	-0.011495	...

Word2Vec: **Drawbacks**

1. **Challenges with Phrase Representations:**
   - Combining word vector representations to obtain phrases like "hot potato" or "Boston Globe" is problematic.
   - Longer phrases and sentences pose even greater complexity.
2. **Limitations in Window Sizes:**
   - Smaller window sizes can lead to similar embeddings for contrasting words (e.g., "good" and "bad").
   - This similarity is undesirable, particularly in tasks like sentiment analysis where differentiation is crucial.
3. **Task Dependency:**
   - Word embeddings' effectiveness is dependent on the specific application.
   - Re-training embeddings for every new task is computationally expensive.
4. **Neglecting Polysemy and Biases:**
   - Word2Vec models may not adequately account for polysemy (multiple meanings of a word) and other biases present in the training data.
   - This can lead to skewed representations and biased outputs in downstream applications.

Pretrained Word-Embedding

Challenges with Word Embeddings from scratch

Training word embeddings from scratch is possible but it is quite challenging due to large trainable parameters and sparsity of training data. Some challenges in Building Word Embeddings from Scratch:

- **Large Trainable Parameters**: Training word embeddings from scratch involves dealing with a large number of trainable parameters, which can lead to computational challenges and increased memory usage.
- **Sparse Training Data:** Building word embeddings requires extensive training data with a rich vocabulary. However, obtaining such datasets can be difficult, particularly for niche domains or languages with limited resources.
- **Training Time**: Due to the vast number of parameters involved, training word embedding models from scratch can be time-consuming. The large-scale computations required often result in slower training processes.
- **Data Diversity**: Word embeddings should ideally capture the semantic relationships and subtleties of language. Achieving this necessitates training on diverse datasets to ensure robustness and generalization, adding to the complexity of the training process.

Pretrained Word-Embedding

Pre-trained word embeddings are representations of words that are learned from large corpora and are made available for reuse in various natural language processing (NLP) tasks.

- These embeddings capture semantic relationships between words, allowing the model to understand similarities and relationships between different words in a meaningful way.
- Pre-trained word embeddings are trained on large datasets and capture the syntactic as well as semantic meaning of the words. This technique is known as transfer learning in which you take a model which is trained on large datasets and use that model on your own similar tasks.
- There are two broad classification methods of pre trained word embeddings – word-level and character-level. We'll be looking into two types of word-level embeddings:

1. **Word2Vec**
2. **GloVe**

Using Pre-trained Models

Gensim comes with several already pre-trained models, in the Gensim-data repository.

- The downloader can be imported from the Gensim library.
- The Gensim downloader includes several pre-trained models like GloVe and Fastext in addition to Word2Vec explained in previous section.

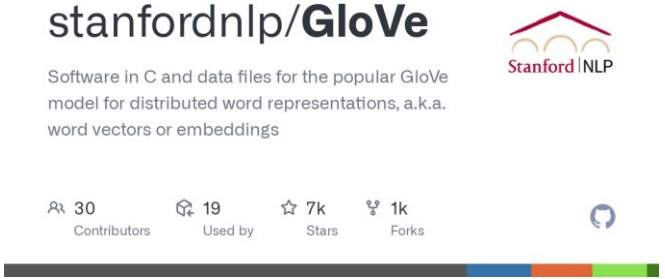
```
import gensim.downloader
print(list(gensim.downloader.info()['models'].keys()))
```

```
Output:

['fasttext-wiki-news-subwords-300', 'conceptnet-numberbatch-17-06-300',
'word2vec-ruscorpora-300', 'word2vec-google-news-300', 'glove-wiki-gigaword-50',
'glove-wiki-gigaword-100', 'glove-wiki-gigaword-200', 'glove-wiki-gigaword-300',
'glove-twitter-25','glove-twitter-50', 'glove-twitter-100', 'glove-twitter-200',
    '__testing_word2vec-matrix-synopsis']
```


GloVe: Global Vector for Text Representation

GloVe stands for Global Vectors for Word Representation. GloVe method of word embedding was developed at Stanford by Pennington, et al first presented it in 2014.

- It is a popular word embedding model which works on the basic idea of deriving the relationship between words using statistics.
- It is a count-based model that employs co-occurrence matrix. A co-occurrence matrix tells how often two words are occurring globally. Each value is a count of a pair of words occurring together.
- Glove basically deals with the spaces where the distance between words is linked to their semantic similarity.
- It has properties of the global matrix factorization and the local context window technique. Training of the model is based on the global word-word co-occurrence data from a corpse, and the resultant representations results into linear substructure of the vector space



GloVe: **Co-occurrence Probabilities**

GloVe calculates the co-occurrence probabilities for each word pair. It divides the co-occurrence counts by the total number of co-occurrences for each word:

For example, the co-occurrence probability of “women” and “empowerment” is calculated as:

$$Co-occurrence\ Probability(\text{“}cat\text{”}, \text{“}mouse\text{”}) = \frac{Count(\text{“}cat\text{”}\ and\ \text{“}mouse\text{”})}{Total\ \ Co{-}occurrences(\text{“}cat\text{”})}$$

In this case:

Count("women" and "empowerment") = 1
Total Co-occurrences("women") = 2 (with “equality" and "empowerment")
Co-occurrence Probability("women", "empowerment") = 1 / 2 = 0.5

GloVe: **Data**

The GloVe has pre-defined dense vectors for around every 6 billion words of English literature along with many other general-use characters like commas, braces, and semicolons.

Users can select a pre-trained GloVe embedding in a dimension (e.g., 50-d, 100-d, etc.) that best fits their needs in terms of computational resources and task specificity. Where d stands for dimension. 100d means, in this file each word has an equivalent vector of size 100.

GloVe Pretrained Model Training Data:

1. Wikipedia 2014 + Gigaword 5 (6B tokens, 400K vocab, uncased, 50d, 100d, 200d, & 300d vectors, 822 MB download)
2. Common Crawl (42B tokens, 1.9M vocab, uncased, 300d vectors, 1.75 GB download)
3. Common Crawl (840B tokens, 2.2M vocab, cased, 300d vectors, 2.03 GB download)
4. Twitter (2B tweets, 27B tokens, 1.2M vocab, uncased, 25d, 50d, 100d, & 200d vectors, 1.42 GB download)

Implementing GloVe

Basic GloVe implementation using Gensim

```
from gensim.models import KeyedVectors
from gensim.downloader import load

# glove_model = load('glove-wiki-gigaword-50')
word_pairs = [('women', 'empowerment'), ('men', 'leaders'), ('science', 'data')]

# Compute similarity for each pair of words
for pair in word_pairs:
    similarity = glove_model.similarity(pair[0], pair[1])
    print(f"Similarity between '{pair[0]}' and '{pair[1]}' using GloVe: {similarity:.3f}")
```

```
Output:

Similarity between 'women' and 'empowerment' using GloVe: 0.374
Similarity between 'men' and 'leaders' using GloVe: 0.577
Similarity between 'science' and 'data' using GloVe: 0.526
```


GloVe: **Applications**

- **Text Classification**: Enhancing machine learning models for sentiment analysis, topic classification, and spam detection.
- **Named Entity Recognition (NER):** Improving NER systems by capturing semantic relationships between words.
- **Machine Translation:** Representing words in source and target languages to enhance translation quality.
- **Question Answering Systems:** Assisting models in understanding context for accurate answers.
- **Document Similarity and Clustering**: Measuring semantic similarity for information retrieval and organization.
- **Word Analogy Tasks**: Successfully completing word analogy tasks by recognizing semantic relationships.

GloVe: **Drawbacks**

- **Fixed Vocabulary:** GloVe embeddings are trained on a fixed vocabulary, limiting their ability to handle out-of-vocabulary words effectively.
- **Lack of Flexibility:** Pre-trained GloVe embeddings might not suit specific domain-specific or task-specific requirements, necessitating additional fine-tuning.
- **Dependency on Corpus Quality:** The quality of GloVe embeddings heavily depends on the quality and size of the training corpus, which may vary across different datasets and languages.

```python
from gensim.models import KeyedVectors
from gensim.downloader import load

# glove_model = load('glove-wiki-gigaword-50')
word_pairs = [('obama', 'elections'), ('bush', 'elections'), ( 'biden', 'elections')]

# Compute similarity for each pair of words
for pair in word_pairs:
    similarity = glove_model.similarity(pair[0], pair[1])
    print(f"Similarity between '{pair[0]}' and '{pair[1]}' using GloVe: {similarity:.3f}")
```

✓ 0.0s Python

```
Similarity between 'obama' and 'elections' using GloVe: 0.584
Similarity between 'bush' and 'elections' using GloVe: 0.612
Similarity between 'biden' and 'elections' using GloVe: 0.396
```

Thank You


SDAIA
الهيئة السعودية للبيانات
والذكاء الاصطناعي
Saudi Data & AI Authority