CS2V14 – TEXT AND SPEECH ANALYSIS (PE) 2024-2025 ODD SEM
UNIT 1 – BASICS OF NATURAL LANGUAGE PROCESSING

Foundations of natural language processing – Language Syntax and Structure- Text Preprocessing and Wrangling – Text tokenization – Stemming – Lemmatization – Removing stop-words – Feature Engineering for Text representation – Bag of Words model- Bag of N-Grams model – TF-IDF model

INTRODUCTION

Artificial intelligence (AI) integration has revolutionized various industries, and now it is transforming the realm of human behaviour research. This integration marks a significant milestone in the data collection and analysis endeavors, enabling users to unlock deeper insights from spoken language and empower researchers and analysts with enhanced capabilities for understanding and interpreting human communication. Human interactions are a critical part of many organizations. Many organizations analyze speech or text via natural language processing (NLP) and link them to insights and automation such as text categorization, text classification, information extraction, etc.

In business intelligence, speech and text analytics enable us to gain insights into customer-agent conversations through sentiment analysis, and topic trends. These insights highlight areas of improvement, recognition, and concern, to better understand and serve customers and employees. Speech and text analytics features provide automated speech and text analytics capabilities on 100% of interactions to provide deep insight into customer-agent conversations. Speech and text analytics is a set of features that uses natural language processing (NLP) to provide an automated analysis of an interaction's content and insight into customer-agent conversations. Speech and text analytics includes transcribing voice interactions, analysis for customer sentiment and topic spotting, and creating meaning from otherwise unstructured data.

FOUNDATIONS OF NATURAL LANGUAGE PROCESSING

Natural Language Processing (NLP) is the process of producing meaningful phrases and sentences in the form of natural language. Natural Language Processing precludes Natural Language Understanding (NLU) and Natural Language

Generation (NLG). NLU takes the data input and maps it into natural language. NLG conducts information extraction and retrieval, sentiment analysis, and more. NLP can be thought of as an intersection of Linguistics, Computer Science and Artificial Intelligence that helps computers understand, interpret and manipulate human language.

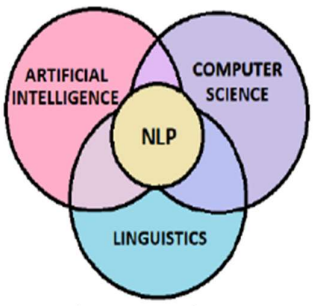

Fig. NLP Overview

Ever since then, there has been an immense amount of study and development in the field of Natural Language Processing.

Today NLP is one of the most in-demand and promising fields of Artificial Intelligence! There are two main parts to Natural Language Processing:

1. Data Preprocessing
2. Algorithm Development

Applications	Core technologies
Machine Translation	Language modelling
Information Retrieval	Part-of-speech tagging
Question Answering	Syntactic parsing
Dialogue Systems	Named-entity recognition
Information Extraction	Coreference resolution
Summarization	Word sense disambiguation
Sentiment Analysis	Semantic Role Labelling

In Natural Language Processing, machine learning training algorithms study millions of examples of text — words, sentences, and paragraphs — written by humans. By studying the samples, the training algorithms gain an understanding of the "context" of human speech, writing, and other modes of communication. This training helps NLP software to differentiate between the meanings of various texts. The five phases of NLP involve lexical (structure) analysis, parsing, semantic analysis, discourse integration, and pragmatic analysis. Some well-known application areas of NLP are Optical Character Recognition (OCR), Speech Recognition, Machine Translation, and Chatbots.

Fig: Five Phases of NLP

Phase I: Lexical or morphological analysis

The first phase of NLP is word structure analysis, which is referred to as lexical or morphological analysis. A lexicon is defined as a collection of words and phrases in a given language, with the analysis of this collection being the process of splitting the lexicon into components, based on what the user sets as parameters – paragraphs, phrases, words, or characters.

Similarly, morphological analysis is the process of identifying the morphemes of a word. A morpheme is a basic unit of English language construction, which is a small element of a word, that carries meaning. These can be either a free morpheme (e.g. walk) or a bound morpheme (e.g. -ing, -ed), with the difference between the two being that the latter cannot stand on it's own to produce a word with meaning, and should be assigned to a free morpheme to attach meaning.

In search engine optimization (SEO), lexical or morphological analysis helps guide web searching. For instance, when doing on-page analysis, you can perform lexical and morphological analysis to understand how often the target keywords are used in their core form (as free morphemes, or when in composition with bound morphemes). This type of analysis can ensure that you have an accurate understanding of the different variations of the morphemes that are used. Morphological analysis can also be applied in transcription and translation projects, so can be very useful in content repurposing projects, and international SEO and linguistic analysis.

Phase II: Syntax analysis (parsing)

Syntax Analysis is the second phase of natural language processing. Syntax analysis or parsing is the process of checking grammar, word arrangement, and overall – the identification of relationships between words and whether those make sense. The process involved examination of all words and phrases in a sentence, and the structures between them.

As part of the process, there's a visualisation built of semantic relationships referred to as a syntax tree (similar to a knowledge graph). This process ensures that the structure and order and grammar of sentences makes sense, when considering the words and phrases that make up those sentences. Syntax analysis also involves tagging words and phrases with POS tags. There are two common methods, and multiple approaches to construct the syntax tree – top-down and bottom-up, however, both are logical and check for sentence formation, or else they reject the input.

Syntax analysis can be beneficial for SEO in several ways:

• Programmatic SEO: Checking whether the produced content makes sense, especially when producing content at scale using an automated or semi-automated approach.

• Semantic analysis: Once you have a syntax analysis conducted, semantic analysis is easy, as well as uncovering the relationship between the different entities recognized in the content.

Phase III: Semantic analysis

Semantic analysis is the third stage in NLP, when an analysis is performed to understand the meaning in a statement. This type of analysis is focused on uncovering the definitions of words, phrases, and sentences and identifying whether the way words are organized in a sentence makes sense semantically.

This task is performed by mapping the syntactic structure, and checking for logic in the presented relationships between entities, words, phrases, and sentences in the text. There are a couple of important functions of semantic analysis, which allow for natural language understanding:

• To ensure that the data types are used in a way that's consistent with their definition.

• To ensure that the flow of the text is consistent.

• Identification of synonyms, antonyms, homonyms, and other lexical items.

• Overall word sense disambiguation.

• Relationship extraction from the different entities identified from the text.

There are several things you can utilise semantic analysis for in SEO. Here are some examples:

• Topic modeling and classification – sort your page content into topics (predefined or modelled by an algorithm). You can then use this for ML-enabled internal linking, where you link pages together on your website using the identified topics. Topic modeling can also be used for classifying first-party collected data such as customer service tickets, or feedback users left on your articles or videos in free form (i.e. comments).

• Entity analysis, sentiment analysis, and intent classification – You can use this type of analysis to perform sentiment analysis and identify intent expressed in the content analysed. Entity identification and sentiment analysis are separate tasks, and both can be done on things like keywords, titles, meta descriptions, page content, but works best when analysing data like comments, feedback forms, or customer service or social media interactions. Intent classification can be done on user queries (in keyword research or traffic analysis), but can also be done in analysis of customer service interactions.

Phase IV: Discourse integration

Discourse integration is the fourth phase in NLP, and simply means contextualisation. Discourse integration is the analysis and identification of the larger context for any smaller part of natural language structure (e.g. a phrase, word or sentence).

During this phase, it's important to ensure that each phrase, word, and entity mentioned are mentioned within the appropriate context. This analysis involves considering not only sentence structure and semantics, but also sentence combination and meaning of the text as a whole. Otherwise, when analyzing the structure of text, sentences are broken up and analyzed and also considered in the context of the sentences that precede and follow them, and the impact that they have on the structure of text. Some common tasks in this phase include: information extraction, conversation analysis, text summarisation, discourse analysis.

Here are some complexities of natural language understanding introduced during this phase:

• Understanding of the expressed motivations within the text, and its underlying meaning.

• Understanding of the relationships between entities and topics mentioned, thematic understanding, and interactions analysis.

• Understanding the social and historical context of entities mentioned.

Discourse integration and analysis can be used in SEO to ensure that appropriate tense is used, that the relationships expressed in the text make logical sense, and that there is overall coherency in the text analysed. This can be especially useful for programmatic SEO initiatives or text generation at scale. The analysis can also be used as part of international

SEO localization, translation, or transcription tasks on big corpuses of data.

There are some research efforts to incorporate discourse analysis into systems that detect hate speech (or in the SEO space for things like content and comment moderation), with this technology being aimed at uncovering intention behind text by aligning the expression with meaning, derived from other texts. This means that, theoretically, discourse analysis can also be used for modeling of user intent (e.g search intent or purchase intent) and detection of such notions in texts.

Phase V: Pragmatic analysis

Pragmatic analysis is the fifth and final phase of natural language processing. As the final stage, pragmatic analysis extrapolates and incorporates the learnings from all other, preceding phases of NLP. Pragmatic analysis involves the process of abstracting or extracting meaning from the use of language, and translating a text, using the gathered knowledge from all other NLP steps performed beforehand.

Here are some complexities that are introduced during this phase

• Information extraction, enabling an advanced text understanding functions such as question-answering.

• Meaning extraction, which allows for programs to break down definitions or documentation into a more accessible language.

• Understanding of the meaning of the words, and context, in which they are used, which enables conversational functions between machine and human (e.g. chatbots).

Pragmatic analysis has multiple applications in SEO. One of the most straightforward ones is programmatic SEO and automated content generation. This type of analysis can also be used for generating FAQ sections on your product, using textual analysis of product documentation, or even capitalizing on the 'People Also Ask' featured snippets by adding an automatically-generated FAQ section for each page you produce on your site.

LANGUAGE SYNTAX AND STRUCTURE

For any language, syntax and structure usually go hand in hand, where a set of specific rules, conventions, and principles govern the way words are combined into phrases; phrases get combines into clauses; and clauses get combined into sentences. We will be talking specifically about the English language syntax and structure in this section. In English, words usually combine together to form other constituent units. These constituents include words, phrases, clauses, and sentences. Considering a sentence, "The brown fox is quick and he is jumping over the lazy dog", it is made of a bunch of words and just looking at the words by themselves don't tell us much.

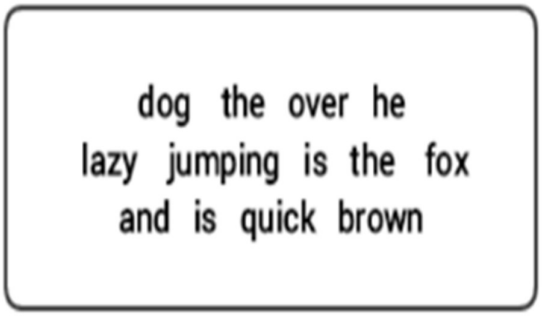

Fig. A bunch of unordered words don't convey much information

Knowledge about the structure and syntax of the language is helpful in many areas like text processing, annotation, and parsing for further operations such as text classification or summarization. Typical parsing techniques for understanding text syntax are mentioned below.

• Parts of Speech (POS) Tagging

• Shallow Parsing or Chunking

• Constituency Parsing

• Dependency Parsing

We will be looking at all of these techniques in subsequent sections. Considering the previous example sentence “The brown fox is quick and he is jumping over the lazy dog”, if we were to annotate it using basic POS tags, it would look like the following figure.

Fig. POS tagging for a sentence

Thus, a sentence typically follows a hierarchical structure consisting the following components,

sentence → clauses → phrases → words

Tagging Parts of Speech

Parts of speech (POS) are specific lexical categories to which words are assigned, based on their syntactic context and role.

Usually, words can fall into one of the following major categories.

• **N(oun):** This usually denotes words that depict some object or entity, which may be living or nonliving. Some

examples would be fox , dog , book , and so on. The POS tag symbol for nouns is N.

• **V(erb):** Verbs are words that are used to describe certain actions, states, or occurrences. There are a wide variety of further subcategories, such as auxiliary, reflexive, and transitive verbs (and many more). Some typical examples of verbs would be running , jumping , read , and write . The POS tag symbol for verbs is V.

• **Adj(ective):** Adjectives are words used to describe or qualify other words, typically nouns and noun phrases. The phrase beautiful flower has the noun (N) flower which is described or qualified using the adjective (ADJ) beautiful . The POS tag symbol for adjectives is ADJ .

• **Adv(erb):** Adverbs usually act as modifiers for other words including nouns, adjectives, verbs, or other adverbs. The phrase very beautiful flower has the adverb (ADV) very , which modifies the adjective (ADJ) beautiful , indicating the degree to which the flower is beautiful. The POS tag symbol for adverbs is ADV. Besides these four major categories of parts of speech , there are other categories that occur frequently in the English language. These include pronouns, prepositions, interjections, conjunctions, determiners, and many others. Furthermore,each POS tag like the noun (N) can be further subdivided into categories like singular nouns (NN), singular proper nouns(NNP), and plural nouns (NNS).

The process of classifying and labeling POS tags for words called parts of speech tagging or POS tagging . POS tags are used to annotate words and depict their POS, which is really helpful to perform specific analysis, such as narrowing down upon nouns and seeing which ones are the most prominent, word sense disambiguation, and grammar analysis.

Let us consider both nltk and spacy which usually use the Penn Treebank notation for POS tagging. NLTK and spaCy are two of the most popular Natural Language Processing (NLP) tools available in Python. You can build chatbots, automatic summarizers, and entity extraction engines with either of these libraries. While both can theoretically accomplish any NLP task, each one excels in certain scenarios. The Penn Treebank, or PTB for short, is a dataset maintained by the University of Pennsylvania.

```
# create a basic pre-processed corpus, don't lowercase to get POS context
corpus = normalize_corpus(news_df['full_text'], text_lower_case=False,
                          text_lemmatization=False, special_char_removal=False)
# demo for POS tagging for sample news headline
sentence = str(news_df.iloc[1].news_headline)
sentence_nlp = nlp(sentence)
# POS tagging with Spacy
spacy_pos_tagged = [(word, word.tag_, word.pos_) for word in sentence_nlp]
pd.DataFrame(spacy_pos_tagged, columns=['Word', 'POS tag', 'Tag type'])
# POS tagging with nltk
nltk_pos_tagged = nltk.pos_tag(sentence.split())
pd.DataFrame(nltk_pos_tagged, columns=['Word', 'POS tag'])
```

	Word	POS tag	Tag type
0	US	NNP	PROPN
1	unveils	VBZ	VERB
2	world	NN	NOUN
3	's	POS	PART
4	most	RBS	ADV
5	powerful	JJ	ADJ
6	supercomputer	NN	NOUN
7	,	,	PUNCT
8	beats	VBZ	VERB
9	China	NNP	PROPN

SpaCy POS tagging

	Word	POS tag
0	US	NNP
1	unveils	VBZ
2	world's	VBZ
3	most	RBS
4	powerful	JJ
5	supercomputer,	JJ
6	beats	NNS
7	China	NNP

NLTK POS tagging

Fig. Python code & Output of POS tagging a news headline

We can see that each of these libraries treat tokens in their own way and assign specific tags for them. Based on what we see, spacy seems to be doing slightly better than nltk.

Shallow Parsing or Chunking

Based on the hierarchy we depicted earlier, groups of words make up phrases. There are five major categories of phrases:

Noun phrase (NP): These are phrases where a noun acts as the head word. Noun phrases act as a subject or object to a verb.

Verb phrase (VP): These phrases are lexical units that have a verb acting as the head word. Usually, there are two forms of verb phrases. One form has the verb components as well as other entities such as nouns, adjectives, or adverbs as parts of the object.

Adjective phrase (ADJP): These are phrases with an adjective as the head word. Their main role is to describe or qualify nouns and pronouns in a sentence, and they will be either placed before or after the noun or pronoun.

Adverb phrase (ADVP): These phrases act like adverbs since the adverb acts as the head word in the phrase. Adverb phrases are used as modifiers for nouns, verbs, or adverbs themselves by providing further details that describe or qualify them.

Prepositional phrase (PP): These phrases usually contain a preposition as the head word and other lexical components like nouns, pronouns, and so on. These act like an adjective or adverb describing other words or phrases.

Shallow parsing, also known as light parsing or chunking, is a popular natural language processing technique of analyzing the structure of a sentence to break it down into its smallest constituents (which are tokens such as words) and group them together into higher-level phrases. This includes POS tags and phrases from a sentence

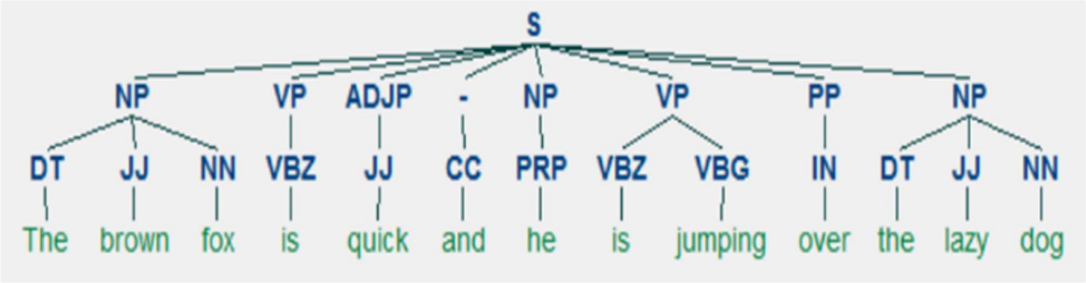

Fig. An example of shallow parsing depicting higher level phrase annotations

Constituency Parsing

Constituent-based grammars are used to analyze and determine the constituents of a sentence. These grammars can be used to model or represent the internal structure of sentences in terms of a hierarchically ordered structure of their constituents.

Each and every word usually belongs to a specific lexical category in the case and forms the head word of different phrases. These phrases are formed based on rules called phrase structure rules.

Phrase structure rules form the core of constituency grammars, because they talk about syntax and rules that govern the hierarchy and ordering of the various constituents in the sentences. These rules cater to two things primarily.

• They determine what words are used to construct the phrases or constituents.

• They determine how we need to order these constituents together.

The generic representation of a phrase structure rule is S → AB , which depicts that the structure S consists of constituents A and B , and the ordering is A followed by B .

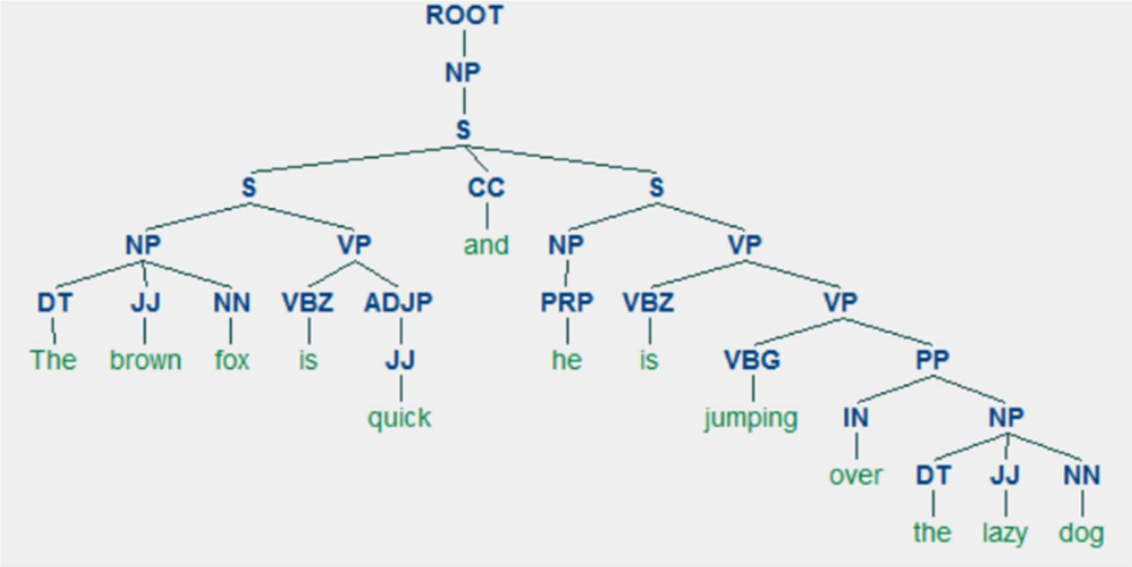

Fig. An example of constituency parsing showing a nested hierarchical structure

Dependency Parsing

In dependency parsing, we try to use dependency-based grammars to analyze and infer both structure and semantic dependencies and relationships between tokens in a sentence. The basic principle behind a dependency grammar is that in any sentence in the language, all words except one, have some relationship or dependency on other words in the sentence.

The word that has no dependency is called the root of the sentence. The verb is taken as the root of the sentence in most cases. All the other words are directly or indirectly linked to the root verb using links, which are the dependencies.

Considering the sentence “The brown fox is quick and he is jumping over the lazy dog”, if we wanted to draw the dependency syntax tree for this, we would have the structure.

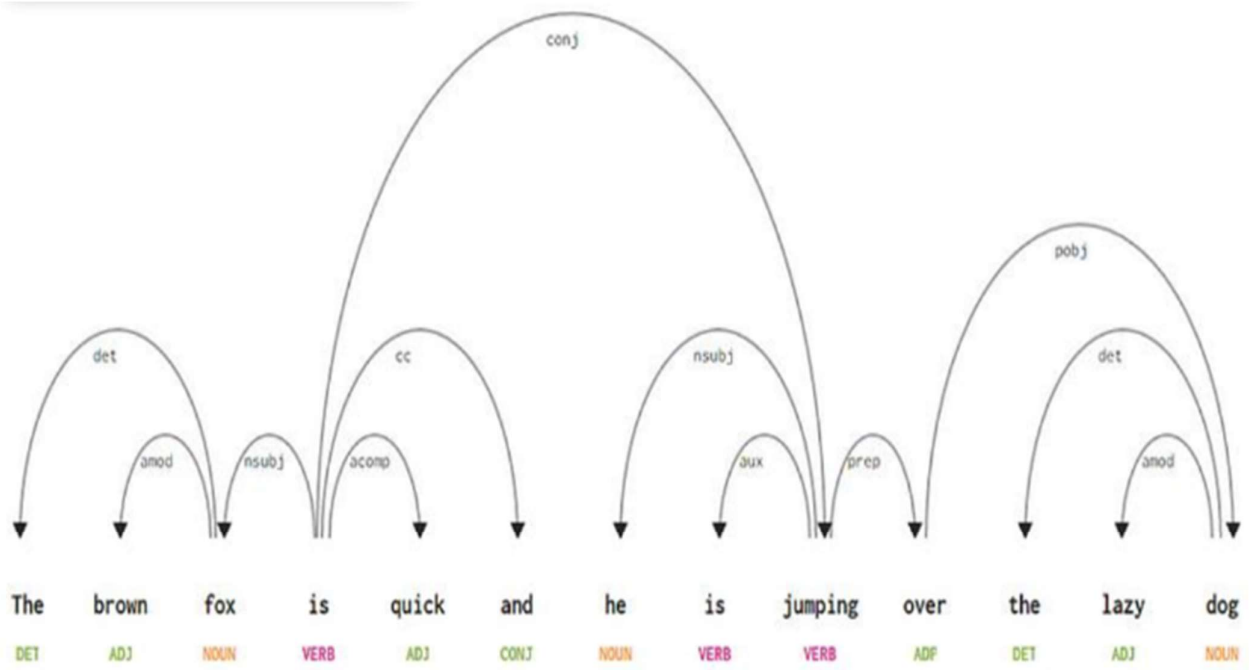

Fig. A dependency parse tree for a sentence

These dependency relationships each have their own meaning and are a part of a list of universal dependency types.

Some of the dependencies are as follows:

• The dependency tag det

is pretty intuitive—it denotes the determiner relationship between a nominal head and the determiner. Usually, the word with POS tag DET will also have the det dependency tag relation. Examples include fox → the and dog → the.

• The dependency tag amod stands for adjectival modifier and stands for any adjective that modifies the meaning of a noun. Examples include fox → brown and dog → lazy.

• The dependency tag nsubj stands for an entity that acts as a subject or agent in a clause. Examples include is → fox and jumping → he.

• The dependencies cc and conj have more to do with linkages related to words connected by coordinating conjunctions. Examples include is → and and is → jumping.

• The dependency tag aux indicates the auxiliary or secondary verb in the clause. Example: jumping → is.

• The dependency tag acomp stands for adjective complement and acts as the complement or object to a verb in the sentence. Example: is → quick

• The dependency tag prep denotes a prepositional modifier, which usually modifies the meaning of a noun, verb, adjective, or preposition. Usually, this representation is used for prepositions having a noun or noun phrase complement. Example: jumping → over.

• The dependency tag pobj is used to denote the object of a preposition. This is usually the head of a noun phrase following a preposition in the sentence. Example: over → dog.

TEXT PREPROCESSING OR WRANGLING

Text preprocessing or wrangling is a method to clean the text data and make it ready to feed data to the model. Text data contains noise in various forms like emotions, punctuation, text in a different case. When we talk about Human Language then, there are different ways to say the same thing, And this is only the main problem we have to deal with because machines will not understand words, they need numbers so we need to convert text to numbers in an efficient manner.

Techniques to perform text preprocessing or wrangling are as follows:

Contraction Mapping/ Expanding Contractions: Contractions are a shortened version of words or a group of words, quite common in both spoken and written language. In English, they are quite common, such as I will to I'll, I have to I've , do not to don't, etc. Mapping these contractions to their expanded form helps in text standardization.

Tokenization: Tokenization is the process of separating a piece of text into smaller units called tokens. Given a document, tokens can be sentences, words, subwords, or even characters depending on the application. Noise cleaning: Special characters and symbols contribute to extra noise in unstructured text. Using regular expressions to remove them or using tokenizers, which do the pre-processing step of removing punctuation marks and other special characters, is recommended.

Spell-checking: Documents in a corpus are prone to spelling errors; In order to make the text clean for the subsequent processing, it is a good practice to run a spell checker and fix the spelling errors before moving on to the next steps.

Stopwords Removal: Stop words are those words which are very common and often less significant. Hence, removing these is a pre-processing step as well. This can be done explicitly by retaining only those words in the document which are not in the list of stop words or by specifying the stop word list as an argument in CountVectorizer or TfidfVectorizer methods when getting Bag-of-Words(BoW)/TF-IDF scores for the corpus of text documents.

Stemming/Lemmatization: Both stemming and lemmatization are methods to reduce words to their base form. While stemming follows certain rules to truncate the words to their base form, often resulting in words that are not lexicographically correct, lemmatization always results in base forms that are lexicographically correct. However, stemming is a lot faster than lemmatization. Hence, to stem/lemmatize is dependent on whether the application needs quick pre-processing or requires more accurate base forms.

TOKENIZATION

Tokenization is a common task in Natural Language Processing (NLP). It's a fundamental step in both traditional NLP methods like Count Vectorizer and Advanced Deep Learning-based architectures like Transformers. As tokens are the building blocks of Natural Language, the most common way of processing the raw text happens at the token level.

Tokenization is a way of separating a piece of text into smaller units called tokens. Here, tokens can be either words, characters, or subwords. Hence, ***tokenization can be broadly classified into 3 types – word, character, and subword (n-gram characters) tokenization.***

For example, consider the sentence: "Never give up".

The most common way of forming tokens is based on space. Assuming space as a delimiter, the tokenization of the sentence results in 3 tokens – Never-give-up. As each token is a word, it becomes an example of Word tokenization.

Similarly, tokens can be either characters or sub-words. For example, let us consider "smarter":

1. ***Character tokens***: s-m-a-r-t-e-r

2. ***Sub-word tokens***: smart-er

Here, Tokenization is performed on the corpus to obtain tokens. The following tokens are then used to prepare a vocabulary.

Vocabulary refers to the set of unique tokens in the corpus. Remember that vocabulary can be constructed by considering each unique token in the corpus or by considering the top K Frequently Occurring Words.

Creating Vocabulary is the ultimate goal of Tokenization.

Word Tokenization

Word Tokenization is the most commonly used tokenization algorithm. It splits a piece of text into individual words based on a certain delimiter. Depending upon delimiters, different word-level tokens are formed. Pretrained Word Embeddings such as Word2Vec and GloVe comes under word tokenization.

Drawbacks of Word Tokenization

One of the major issues with word tokens is dealing with Out Of Vocabulary (OOV) words. OOV words refer to the new words which are encountered at testing. These new words do not exist in the vocabulary. Hence, these methods fail in handling Out-of-Vocabolary (OOV) words.

• A small trick can rescue word tokenizers from OOV words. The trick is to form the vocabulary with the Top K Frequent Words and replace the rare words in training data with **unknown tokens (UNK)**. This helps the model to learn the representation of OOV words in terms of UNK tokens

• So, during test time, any word that is not present in the vocabulary will be mapped to a UNK token. This is how we can tackle the problem of OOV in word tokenizers.

• The problem with this approach is that the entire information of the word is lost as we are mapping OOV to UNK tokens. The structure of the word might be helpful in representing the word accurately. And another issue is that every OOV word gets the same representation

Another issue with word tokens is connected to the size of the vocabulary. Generally, pre-trained models are trained on a large volume of the text corpus. So, just imagine building the vocabulary with all the unique words in such a large corpus. This explodes the vocabulary!

Character Tokenization

Character Tokenization splits apiece of text into a set of characters. It overcomes the drawbacks we saw above about Word Tokenization.

• Character Tokenizers handles OOV words coherently by preserving the information of the word. It breaks down

the OOV word into characters and represents the word in terms of these characters

• It also limits the size of the vocabulary. Want to talk a guess on the size of the vocabulary? 26 since the vocabulary contains a unique set of characters

Drawbacks of Character Tokenization

Character tokens solve the OOV problem but the length of the input and output sentences increases rapidly as we are representing a sentence as a sequence of characters. As a result, it becomes challenging to learn the relationship between the characters to form meaningful words. This brings us to another tokenization known as Subword Tokenization which is in between a Word and Character tokenization.

Subword Tokenization

Subword Tokenization splits the piece of text into subwords (or n-gram characters). For example, words like lower can be segmented as low-er, smartest as smart-est, and so on. Transformed-based models – the SOTA in NLP – rely on Subword Tokenization algorithms for preparing vocabulary.

Byte Pair Encoding (BPE)

Byte Pair Encoding (BPE) is a widely used tokenization method among transformer-based models. BPE addresses the issues of Word and Character Tokenizers. BPE tackles OOV effectively. It segments OOV as subwords and represents the word in terms of these subwords

- The length of input and output sentences after BPE are shorter compared to character tokenization
- BPE is a word segmentation algorithm that merges the most frequently occurring character or character sequences iteratively.

BPE is a word segmentation algorithm that merges the most frequently occurring character or character sequences
iteratively. Here is a step by step guide to learn BPE.
1. Split the words in the corpus into characters after appending </w>
2. Initialize the vocabulary with unique characters in the corpus
3. Compute the frequency of a pair of characters or character sequences in corpus
4. Merge the most frequent pair in corpus
5. Save the best pair to the vocabulary
6. Repeat steps 3 to 5 for a certain number of iterations

STEMMING

Stemming is the process of producing morphological variants of a root/base word. Stemming programs are commonly referred to as stemming algorithms or stemmers. A stemming algorithm reduces the words “chocolates”, “chocolatey”, “choco” to the root word, “chocolate” and “retrieval”, “retrieved”, “retrieves” reduce to the stem “retrieve”. Stemming is an important part of the pipelining process in Natural language processing. The input to the stemmer is tokenized words.

How do we get these tokenized words? Well, tokenization involves breaking down the document into different words.

Stemming is a natural language processing technique that is used to reduce words to their base form, also known as the root form. The process of stemming is used to normalize text and make it easier to process. It is an important step in text pre-processing, and it is commonly used in information retrieval and text mining applications. There are several different algorithms for stemming as follows:

- Porter stemmer
- Snowball stemmer
- Lancaster stemmer.

The Porter stemmer is the most widely used algorithm, and it is based on a set of heuristics that are used to remove common suffixes from words. The Snowball stemmer is a more advanced algorithm that is based on the Porter stemmer, but it also supports several other languages in addition to English. The Lancaster stemmer is a more aggressive stemmer and it is less accurate than the Porter stemmer and Snowball stemmer. Stemming can be useful for several natural language processing tasks such as text classification, information retrieval, and text summarization. However, stemming can also have some negative effects such as reducing the readability of the text, and it may not always produce the correct root form of a word. It is important to note that stemming is different from Lemmatization. Lemmatization is the process of reducing a word to its base form, but unlike stemming, it takes into account the context of the word, and it produces a valid word, unlike stemming which can produce a non-word as the root form. Some more examples stemming from the root word "like" include:

->"likes"
->"liked"
->"likely"
->"liking"

Errors in Stemming:

There are mainly two errors in stemming –
- over-stemming
- under-stemming

Over-stemming occurs when two words are stemmed from the same root that are of different stems. Over-stemming can also be regarded as false-positives. Over-stemming is a problem that can occur when using stemming algorithms in natural language processing. It refers to the situation where a stemmer produces a root form that is not a valid word or is not the correct root form of a word. This can happen when the stemmer is too aggressive in removing suffixes or when it does not consider the context of the word.
Over-stemming can lead to a loss of meaning and make the text less readable. For example, the word “arguing” may be stemmed to “argu,” which is not a valid word and does not convey the same meaning as the original word. Similarly, the word “running” may be stemmed to “run,” which is the base form of the word but it does not convey the meaning of the original word.
To avoid over-stemming, it is important to use a stemmer that is appropriate for the task and language. It is also important to test the stemmer on a sample of text to ensure that it is producing valid root forms. In some cases, using a lemmatizer instead of a stemmer may be a better solution as it takes into account the context of the word, making it less prone to errors.
Another approach to this problem is to use techniques like semantic role labeling, sentiment analysis, context-based information, etc. that help to understand the context of the text and make the stemming process more precise.

Under-stemming occurs when two words are stemmed from the same root that are not of different stems. Under-stemming can be interpreted as false-negatives. Under-stemming is a problem that can occur when using stemming algorithms in natural language processing. It refers to the situation where a stemmer does not produce the correct root form of a word or does not reduce a word to its base form. This can happen when the stemmer is not aggressive enough in removing suffixes or when it is not designed for the specific task or language.
Under-stemming can lead to a loss of information and make it more difficult to analyze text. For example, the word “arguing” and “argument” may be stemmed to “argu,” which does not convey the meaning of the original words. Similarly, the word “running” and “runner” may be stemmed to “run,” which is the base form of the word but it does not convey the meaning of the original words.
To avoid under-stemming, it is important to use a stemmer that is appropriate for the task and language. It is also important to test the stemmer on a sample of text to ensure that it is producing

the correct root forms. In some cases, using a lemmatizer instead of a stemmer may be a better solution as it takes into account the context of the word, making it less prone to errors.
Another approach to this problem is to use techniques like semantic role labeling, sentiment analysis, context-based information, etc. that help to understand the context of the text and make the stemming process more precise.

Applications of stemming:
Stemming is used in information retrieval systems like search engines. It is used to determine domain vocabularies in domain analysis. To display search results by indexing while documents are evolving into numbers and to map documents to common subjects by stemming. Sentiment Analysis, which examines reviews and comments made by different users about anything, is frequently used for product analysis, such as for online retail stores. Before it is interpreted, stemming is accepted in the form of the text-preparation mean.
A method of group analysis used on textual materials is called document clustering (also known as text clustering). Important uses of it include subject extraction, automatic document structuring, and quick information retrieval.

Fun Fact: Google search adopted a word stemming in 2003. Previously a search for “fish” would not have returned “fishing”
or “fishes”.
Some Stemming algorithms are:

Porter’s Stemmer algorithm
It is one of the most popular stemming methods proposed in 1980. It is based on the idea that the suffixes in the English language are made up of a combination of smaller and simpler suffixes. This stemmer is known for its speed and simplicity. The main applications of Porter Stemmer include data mining and Information retrieval. However, its applications are only limited to English words. Also, the group of stems is mapped on to the same stem and the output stem is not necessarily a meaningful word. The algorithms are fairly lengthy in nature and are known to be the oldest stemmer.

Example: EED -> EE means “if the word has at least one vowel and consonant plus EED ending, change the ending to EE” as ‘agreed’ becomes ‘agree’.

Advantage: It produces the best output as compared to other stemmers and it has less error rate.
Limitation: Morphological variants produced are not always real words.

Lovins Stemmer It is proposed by Lovins in 1968, that removes the longest suffix from a word then the word is recorded to convert this stem into valid words.
Example: sitting -> sitt -> sit
Advantage: It is fast and handles irregular plurals like 'teeth' and 'tooth' etc.
Limitation: It is time consuming and frequently fails to form words from stem.

Dawson Stemmer
It is an extension of Lovins stemmer in which suffixes are stored in the reversed order indexed by their length and last letter.
Advantage: It is fast in execution and covers more suffices.
Limitation: It is very complex to implement.

Krovetz Stemmer
It was proposed in 1993 by Robert Krovetz. Following are the steps:
1) Convert the plural form of a word to its singular form.
2) Convert the past tense of a word to its present tense and remove the suffix ‘ing’.
Example: ‘children’ -> ‘child’

Advantage: It is light in nature and can be used as pre-stemmer for other stemmers.
Limitation: It is inefficient in case of large documents.

Xerox Stemmer
Example:
'children' -> 'child'
'understood' -> 'understand'
'whom' -> 'who'
'best' -> 'good'

N-Gram Stemmer
An n-gram is a set of n consecutive characters extracted from a word in which similar words will have a high proportion of n-grams in common.
Example: 'INTRODUCTIONS' for n=2 becomes : *I, IN, NT, TR, RO, OD, DU, UC, CT, TI, IO, ON, NS, S*
Advantage: It is based on string comparisons and it is language dependent.
Limitation: It requires space to create and index the n-grams and it is not time efficient.

Snowball Stemmer:
When compared to the Porter Stemmer, the Snowball Stemmer can map non-English words too. Since it supports other languages the Snowball Stemmers can be called a multi-lingual stemmer. The Snowball stemmers are also imported from the nltk package. This stemmer is based on a programming language called 'Snowball' that processes small strings and is the most widely used stemmer. The Snowball stemmer is way more aggressive than Porter Stemmer and is also referred to as Porter2 Stemmer. Because of the improvements added when compared to the Porter Stemmer, the Snowball stemmer is having greater computational speed.

Lancaster Stemmer:
The Lancaster stemmers are more aggressive and dynamic compared to the other two stemmers. The stemmer is really faster, but the algorithm is really confusing when dealing with small words. But they are not as efficient as Snowball Stemmers. The Lancaster stemmers save the rules externally and basically uses an iterative algorithm. Lancaster Stemmer is straightforward, although it often produces results with excessive stemming. Over-stemming renders stems non-linguistic or meaningless.

LEMMATIZATION

Lemmatization is a text pre-processing technique used in natural language processing (NLP) models to break a word down to its root meaning to identify similarities. For example, a lemmatization algorithm would reduce the word better to its root word, or lemme, good. In stemming, a part of the word is just chopped off at the tail end to arrive at the stem of the word.

There are different algorithms used to find out how many characters have to be chopped off, but the algorithms don't actually know the meaning of the word in the language it belongs to. In lemmatization, the algorithms do have this knowledge. In fact, you can even say that these algorithms refer to a dictionary to understand the meaning of the word before reducing it to its root word, or lemma. So, a lemmatization algorithm would know that the word better is derived from the word good, and hence, the lemme is good. But a stemming algorithm wouldn't be able to do the same. There could be over-stemming or under-stemming, and the word better could be reduced to either bet, or bett, or just retained as better. But there is no way in stemming that can reduce better to its root word good. This is the difference between stemming and lemmatization.

Original	Stemming	Lemmatization
New	New	New
York	York	York
is	is	be
the	the	the
most	most	most
densely	dens	densely
populated	popul	populated
city	citi	city
in	in	in
the	the	the
United	Unite	United
States	State	States

Stemming vs Lemmatization

Lemmatization gives more context to chatbot conversations as it recognizes words based on their exact and contextual meaning. On the other hand, lemmatization is a time-consuming and slow process. The obvious advantage of lemmatization is that it is more accurate than stemming. So, if you're dealing with an NLP application such as a chat bot or a virtual assistant, where understanding the meaning of the dialogue is crucial, lemmatization would be useful. But this accuracy comes at a cost. Because lemmatization involves deriving the meaning of a word from something like a dictionary, it's very time-consuming. So, most lemmatization algorithms are slower compared to their stemming counterparts.

REMOVING STOP-WORDS

The words which are generally filtered out before processing a natural language are called stop words. These are actually the most common words in any language (like articles, prepositions, pronouns, conjunctions, etc) and does not add much information to the text. Examples of a few stop words in English are "the", "a", "an", "so", "what". Stop words are available in abundance in any human language. By removing these words, we remove the low-level information from our text in order to give more focus to the important information. In order words, we can say that the removal of such words does not show any negative consequences on the model we train for our task.

Removal of stop words definitely reduces the dataset size and thus reduces the training time due to the fewer number of tokens involved in the training. We do not always remove the stop words. The removal of stop words is highly dependent on the task we are performing and the goal we want to achieve. For example, if we are training a model that can perform the sentiment analysis task, we might not remove the stop words.

Movie review: "The movie was not good at all."
Text after removal of stop words: "movie good"

We can clearly see that the review for the movie was negative. However, after the removal of stop words, the review became positive, which is not the reality. Thus, the removal of stop words can be problematic here. Tasks like text classification do not generally need stop words as the other words present in the dataset are more important and give the general idea of the text. So, we generally remove stop words in such tasks.

In a nutshell, NLP has a lot of tasks that cannot be accomplished properly after the removal of stop words. So, think before performing this step. The catch here is that no rule is universal and no stop words list is universal. A list not conveying any important information to one task can convey a lot of information to the other task.

Word of caution: Before removing stop words, research a bit about your task and the problem you are trying to solve, and then make your decision.

Sample text with Stop Words	Without Stop Words
GeeksforGeeks – A Computer Science Portal for Geeks	GeeksforGeeks , Computer Science, Portal ,Geeks
Can listening be exhausting?	Listening, Exhausting
I like reading, so I read	Like, Reading, read

Next comes a very important question: why we should remove stop words from the text?. So, there are two main reasons for that:

1. They provide no meaningful information, especially if we are building a text classification model. Therefore, we have to remove stop words from our dataset.
2. As the frequency of stop words are too high, removing them from the corpus results in much smaller data in terms of size. Reduced size results in faster computations on text data and the text classification model need to deal with a lesser number of features resulting in a robust model.

FEATURE ENGINEERING FOR TEXT REPRESENTATION

Feature engineering is one of the most important steps in machine learning. It is the process of using domain knowledge of the data to create features that make machine learning algorithms work. Think machine learning algorithm as a learning child the more accurate information you provide the more they will be able to interpret the information well. Focusing first on our data will give us better results than focusing only on models. Feature engineering helps us to create better data which helps the model understand it well and provide reasonable results.

NLP is a subfield of artificial intelligence where we understand human interaction with machines using natural languages. To understand a natural language, you need to understand how we write a sentence, how we express our thoughts using different words, signs, special characters, etc basically we should understand the context of the sentence to interpret its meaning.

Extracting Features from Text

In this section, we will learn about common feature extraction techniques and methods. We'll also talk about when to use them and some challenges we might face implementing those techniques. ***Feature extraction methods can be divided into 3 major categories, basic, statistical, and advanced/vectorized.***

Basic Methods
These feature extraction methods are based on various concepts from NLP and linguistics. These are some of the oldest methods but still can be very reliable are used frequently in many areas.

- Parsing
- PoS Tagging
- Name Entity Recognition (NER)
- Bag of Words (BoW)

Statistical Methods
This is a bit more advanced feature extraction method and uses the concepts from statistics and probability to extract features
from text data.

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

Advanced Methods
These methods can also be called vectorized methods as they aim to map a word, sentence, document to a fixed-length vector of real numbers. The goal of this method is to extract semantics from a piece of text, both lexical and distributional. Lexical semantics is just the meaning reflected by the words whereas distributional semantics refers to finding meaning based on various distributions in a corpus.

- Word2Vec
- GloVe: Global Vector for word representation

BAG OF WORDS MODEL

The bag-of-words model is a simplifying representation used in natural language processing and information retrieval (IR). In this model, a text (such as a sentence or a document) is represented as the bag (multiset) of its words, disregarding grammar and even word order but keeping multiplicity. A bag-of-words model, or BoW for short, is a way of extracting features from text for use in modelling, such as with machine learning algorithms.

The approach is very simple and flexible, and can be used in a myriad of ways for extracting features from documents.
A bag-of-words is a representation of text that describes the occurrence of words within a document. It involves two things:
1. A vocabulary of known words.
2. A measure of the presence of known words.

It is called a “bag” of words, because any information about the order or structure of words in the document is discarded. The model is only concerned with whether known words occur in the document, not where in the document. The intuition is that documents are similar if they have similar content. Further, that from the content alone we can learn something about the meaning of the document. The bag-of-words can be as simple or complex as you like. The complexity comes both in deciding how to design the vocabulary of known words (or tokens) and how to score the presence of known words.

One of the biggest problems with text is that it is messy and unstructured, and machine learning algorithms prefer structured, well defined fixed-length inputs and by using the Bag-of-Words technique we can convert variable-length texts into a fixed-length vector.

Also, at a much granular level, the machine learning models work with numerical data rather than textual data. So to be more specific, by using the bag-of-words (BoW) technique, we convert a text into its equivalent vector of numbers.

Let us see an example of how the bag of words technique converts text into vectors

Example (1) without preprocessing:

Sentence 1: “Welcome to Great Learning, Now start learning”
Sentence 2: “Learning is a good practice”

Sentence	Welcome	to	Great	Learning	,	Now	start	learning	is	a	good	practice
Sentence1	1	1	1	1	1	1	1	1	0	0	0	0
Sentence2	0	0	0	0	0	0	0	1	1	1	1	1

But is this the best way to perform a bag of words. The above example was not the best example of how to use a bag of words. The words Learning and learning, although having the same meaning are taken twice. Also, a comma ’,’ which does not convey any information is also included in the vocabulary.

Example(2) with preprocessing:

Sentence 1: ”Welcome to Great Learning, Now start learning”
Sentence 2: “Learning is a good practice”

Step 1: Convert the above sentences in lower case as the case of the word does not hold any information.
Step 2: Remove special characters and stopwords from the text. Stopwords are the words that do not contain much information about text like ‘is’, ‘a’,’the and many more’.

After applying the above steps, the sentences are changed to

Sentence 1: ”welcome great learning now start learning”
Sentence 2: “learning good practice”

Step 3: Go through all the words in the above text and make a list of all of the words in our model vocabulary.

- welcome
- great
- learning
- now
- start
- good
- Practice

Now as the vocabulary has only 7 words, we can use a fixed-length document-representation of 7, with one position in the vector to score each word.The scoring method we use here is the same as used in the previous example. For sentence 1, the count of words is as follow:

Word	Frequency
welcome	1
great	1
learning	2
now	1
start	1
good	0
practice	0

Writing the above frequencies in the vector

Sentence 1 → [1,1,2,1,1,0,0]

Now for sentence 2, the scoring would be like

Word	Frequency
welcome	0
great	0
learning	1
now	0
start	0
good	1
practice	1

Similarly, writing the above frequencies in the vector form

Sentence 2 → [0,0,1,0,0,1,1]

Sentence	welcome	great	learning	now	start	good	practice
Sentence1	1	1	2	1	1	0	0
Sentence2	0	0	1	0	0	1	1

The approach used in example two is the one that is generally used in the Bag-of-Words technique, the reason being that the datasets used in Machine learning are tremendously large and can contain vocabulary of a few thousand or even millions of words. Hence, preprocessing the text before using bag-of-words is a better way to go.There are various preprocessing steps that can increase the performance of Bag-of-Words.

Limitations of Bag-of-Words

The bag-of-words model is very simple to understand and implement and offers a lot of flexibility for customization on your specific text data. It has been used with great success on prediction problems like language modeling and documentation classification.
Nevertheless, it suffers from some shortcomings, such as:

•Vocabulary: The vocabulary requires careful design, most specifically in order to manage the size, which impacts the sparsity of the document representations.
•Sparsity: Sparse representations are harder to model both for computational reasons (space and time complexity) and also for information reasons, where the challenge is for the models to harness so little information in such a large representational space.
•Meaning: Discarding word order ignores the context, and in turn meaning of words in the document (semantics). Context and meaning can offer a lot to the model, that if modeled could tell the difference between the same words differently arranged ("this is interesting" vs "is this interesting"), synonyms ("old bike" vs "used bike"), and much more.

What are N-Grams?

Again same questions, what are n-grams and why do we use them? Let us understand this with an example below-
Sentence 1: "This is a good job. I will not miss it for anything"
Sentence 2: "This is not good at all"
For this example, let us take the vocabulary of 5 words only. The five words being-

- good
- job
- miss
- not
- all

So, the respective vectors for these sentences are:
"This is a good job. I will not miss it for anything"=[1,1,1,1,0]
"This is not good at all"=[1,0,0,1,1]
Can you guess what is the problem here? Sentence 2 is a negative sentence and sentence 1 is a positive sentence. Does this reflect in any way in the vectors above? Not at all. So how can we solve this problem? Here come the N-grams to our rescue.

An N-gram is an N-token sequence of words: a 2-gram (more commonly called a bigram) is a two-word sequence of words like "really good", "not good", or "your homework", and a 3-gram (more commonly called a trigram) is a three-word sequence of words like "not at all", or "turn off light".For example, the bigrams in the first line of text in the previous section: "This is not good at all" are as follows:

- "This is"
- "is not"
- "not good"

- “good at”
- “at all”
-

Now if instead of using just words in the above example, we use bigrams (Bag-of-bigrams) as shown above. The model can differentiate between sentence 1 and sentence 2. So, using bi-grams makes tokens more understandable (for example, “HSR Layout”, in Bengaluru, is more informative than “HSR” and “layout”)So we can conclude that a bag-of-bigrams representation is much more powerful than bag-of-words, and in many cases proves very hard to beat.

N-grams:

N-grams are contiguous sequences of n items from a given sample of text or speech. In the context of natural language processing (NLP), these items are typically words, characters, or symbols. N-grams are used for various tasks in NLP, including language modeling, text generation, and feature extraction. The value of ’n’ in n-grams determines the number of consecutive elements considered. Here’s a breakdown of different types of n-grams:

1. Unigrams (1-grams): Unigrams are single words considered individually. For example, in the sentence “I love natural language processing,” the unigrams would be [‘I’, ‘love’, ‘natural’, ‘language’, ‘processing’].
2. Bigrams (2-grams): Bigrams consist of sequences of two adjacent words. For the same sentence, the bigrams would be [‘I love’, ‘love natural’, ‘natural language’, ‘language processing’].
3. Trigrams (3-grams): Trigrams are sequences of three adjacent words. Continuing with the example sentence, the trigrams would be [‘I love natural’, ‘love natural language’, ‘natural language processing’].
4. N-grams (N-grams): N-grams refer to sequences of N adjacent elements, which can be words, characters, or symbols. For instance, if ’N’ is 4, then we have 4-grams, also known as quadgrams.

N-grams are valuable because they capture more context than individual words, especially for tasks where word order is important. They can help in tasks like language modeling, where predicting the next word in a sentence relies on the preceding words. Additionally, n-grams can be used to identify common phrases or expressions in text, aiding in tasks such as sentiment analysis or document categorization.
N-grams can be used to generate text by predicting the next word or character based on the preceding n-1 words or characters. This approach is commonly used in chatbots, text summarization systems, and content generation tools.
However, the main challenge with using n-grams is the exponential increase in the number of unique combinations as ’n’ increases. This can lead to high dimensionality, increased computational complexity, and issues with sparsity, especially when dealing with large vocabularies or corpora.

TF-IDF MODEL

TF-IDF (Term Frequency-Inverse Document Frequency) is a numerical statistic used in natural language processing and information retrieval to evaluate the importance of a term (word) within a document relative to a collection of documents (corpus). TF-IDF is calculated based on two main components: term frequency (TF) and inverse document frequency (IDF).

Here’s a breakdown of how TF-IDF is calculated:

1. Term Frequency (TF): Term frequency measures how frequently a term appears in a document. It is calculated as the ratio of the number of times a term occurs in a document

to the total number of terms in the document. The idea is to give higher weight to terms that appear more frequently within a document.

The formula for TF is typically:

TF(t, d) = Number of times term t appears in document d /

Total number of terms in document d

2. **Inverse Document Frequency (IDF):** Inverse document frequency measures how unique or rare a term is across a collection of documents. It is calculated as the logarithm of the ratio of the total number of documents in the corpus to the number of documents containing the term. The formula for IDF is typically:

IDF(t, D) = log (Total number of documents in the corpus /D/ /

Number of documents containing term t)

3. **TF-IDF Calculation:** The TF-IDF score for a term 't' in a document 'd' is calculated by multiplying the term frequency (TF) of 't' in 'd' by the inverse document frequency (IDF) of 't' across the entire corpus. The formula for TF-IDF is:

TF-IDF(t,d,D) = TF(t,d) × IDF(t,D)

Here, D represents the entire corpus of documents.

TF-IDF assigns higher weights to terms that are frequent within a document (high TF) but rare across the entire corpus (high IDF), indicating their importance in distinguishing that document from others. Conversely, common terms that appear frequently across many documents are assigned lower weights.

TF-IDF is commonly used in various text processing tasks, including:

- Document retrieval: To rank documents based on their relevance to a query.
- Text classification: To extract features for training machine learning models.
- Keyword extraction: To identify important terms or phrases within a document.
- Information retrieval: To index and search text documents efficiently.

Overall, TF-IDF is a useful technique for representing and evaluating the importance of terms in text data, aiding in various NLP and information retrieval tasks.

Let's understand it using a simple example:
Sentence 1: good boy

Sentence 2: good girl
Sentence 3: boy girl good

First we calculate, words Frequency: *good: 3, boy: 2, girl: 2*

We will calculate TF:

	Sent1	Sent2	Sent3
good	1/2	1/2	1/3
boy	1/2	0	1/3
girl	0	1/2	1/3

Then we will calculate IDF:

Words	IDF
good	log(3/3)=0
boy	log(3/2)
girl	log(3/2)

Now multiply last two tables:

	good	boy	girl
Sent 1	(1/2)*0	1/2* log(3/2)	0
Sent 2	0	0	1/2*log(3/2)
Sent 3	0	1/3*log(3/2)	1/3*log(3/2)

Hence we can see that, for Sentence 1, `boy` value is higher compared to other words. so there is some semantic meaning
And similarly, For Sentence 2 `girl` is given importance and for Sentence 3: `boy` and `girl`

Implementation of TF-IDF

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Sample corpus (collection of documents)
corpus = [
    'This is the first document.',
    'This document is the second document.',
    'And this is the third one.',
    'Is this the first document?',
]

# Create an instance of TfidfVectorizer
vectorizer = TfidfVectorizer()

# Fit the vectorizer to the corpus and transform the corpus into a TF-IDF matrix
tfidf_matrix = vectorizer.fit_transform(corpus)
```

```
# Get the list of unique words (vocabulary)
vocab = vectorizer.get_feature_names_out()

# Convert the TF-IDF matrix to a dense numpy array for easier manipulation
tfidf_matrix_dense = tfidf_matrix.toarray()

# Print the TF-IDF matrix and the corresponding vocabulary
print("TF-IDF matrix:")
print(tfidf_matrix_dense)
print("\nVocabulary:")
print(vocab)
```

Output of above:

```
print("TF-IDF matrix:")
print(tfidf_matrix_dense)
print("\nVocabulary:")
print(vocab)
```

```
TF-IDF matrix:
[[0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]
 [0.         0.6876236  0.         0.28108867 0.         0.53864762
  0.28108867 0.         0.28108867]
 [0.51184851 0.         0.         0.26710379 0.51184851 0.
  0.26710379 0.51184851 0.26710379]
 [0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]]

Vocabulary:
['and' 'document' 'first' 'is' 'one' 'second' 'the' 'third' 'this']
```

Disadvantages of TF-IDF:

1. Semantic info is not stored, as order of words may not be same.
2. TF-IDF gives importance to uncommon words.
3. It doesn't consider the order of words within a document.

To overcome these problems of TF-IDF, we can use word2vec model.

One more example.

Let's say that the term "car" appears 25 times in a document that contains 1,000 words.
We'd calculate the term frequency (TF) as follows:

TF = 25/1,000 = 0.025

Next, let's say that a collection of related documents contains a total of 15,000 documents.
If 300 documents out of the 15,000 contain the term "car," we would calculate the inverse document frequency as follows:

IDF = log 15,000/300 = 1.69

Now, we can calculate the TF-IDF score by multiplying these two numbers:

TF-IDF = TF x IDF = 0.025 x 1.69 = 0.04225