DS JULY 2022 Batch
Module 22 : Natural language processing

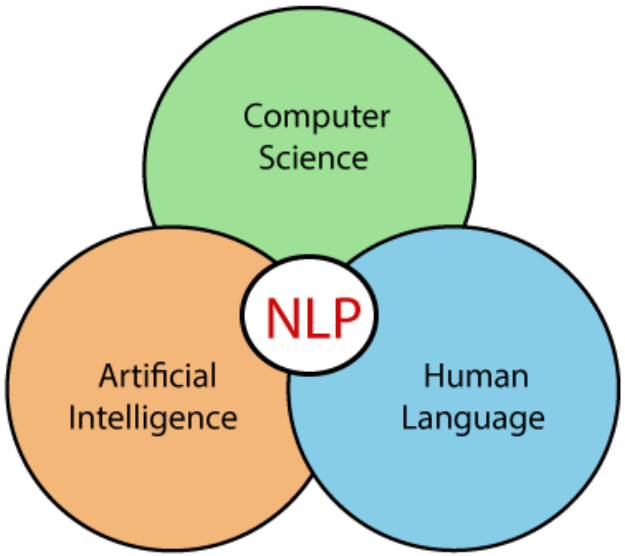
Topics

- Texts, Tokens
- Regex
- Stemming, lemmatization
- Bag of Words
- Basic text classification based on Bag of Words
- n-gram: Unigram, Bigram
- TF-IDF Vectorizer
- Word2Vec

Text, Tokens

Basic NLP

- NLP stands for Natural Language Processing, which is a part of Computer Science, Human language, and Artificial Intelligence.
- It is the technology that is used by machines to understand, analyse, manipulate, and interpret human's languages.
- It helps developers to organize knowledge for performing tasks such as translation, automatic summarization, Named Entity Recognition (NER), speech recognition, relationship extraction, and topic segmentation.

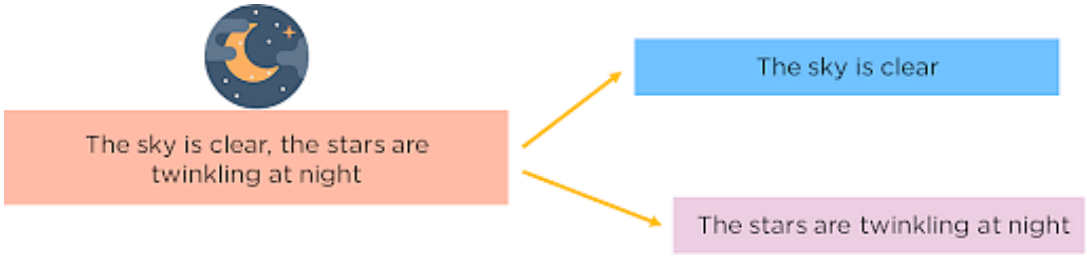


Basic NLP

- NLP combines the field of linguistics and computer science to decipher language structure and guidelines and to make models which can comprehend, break down and separate significant details from text and speech.
- In order to produce significant and actionable insights from text data, it is important to get acquainted with the basics of Natural Language Processing (NLP).
- What Can Natural Language Processing Do?
  - Intelligent Assistants
  - Search Results
  - Intuitive Typing
  - Translation
  - Data Analysis
  - Spam Detection
  - Sentiment Analysis
  - And many more .....

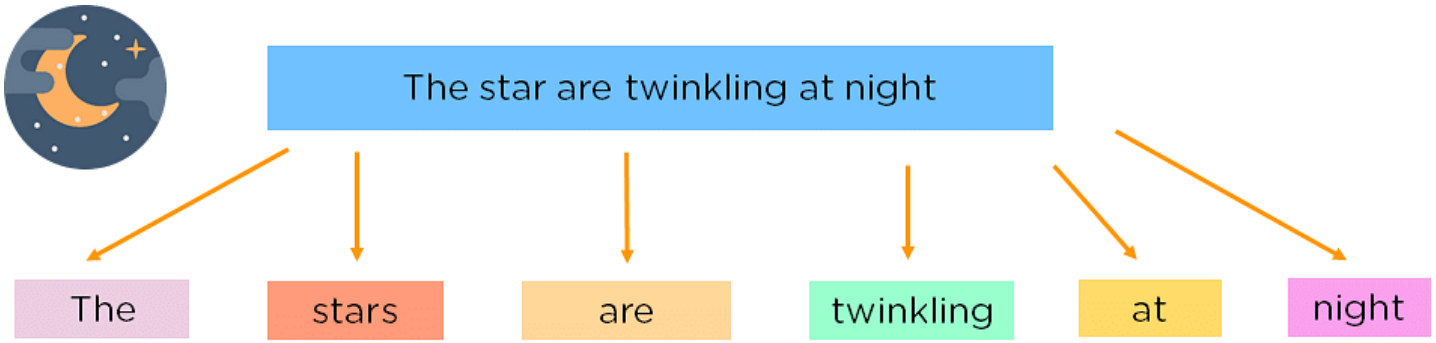
NLP : Text , Tokens

- **Segmentation:**
- Sentence Segment is the first step for building the NLP pipeline. It breaks the paragraph into separate sentences.
- Example: Consider the following paragraph -
  - Independence Day is one of the important festivals for every Indian citizen. It is celebrated on the 15th of August each year ever since India got independence from the British rule. The day celebrates independence in the true sense.
- Sentence Segment produces the following result:
  - "Independence Day is one of the important festivals for every Indian citizen."
  - "It is celebrated on the 15th of August each year ever since India got independence from the British rule."
  - "This day celebrates independence in the true sense."



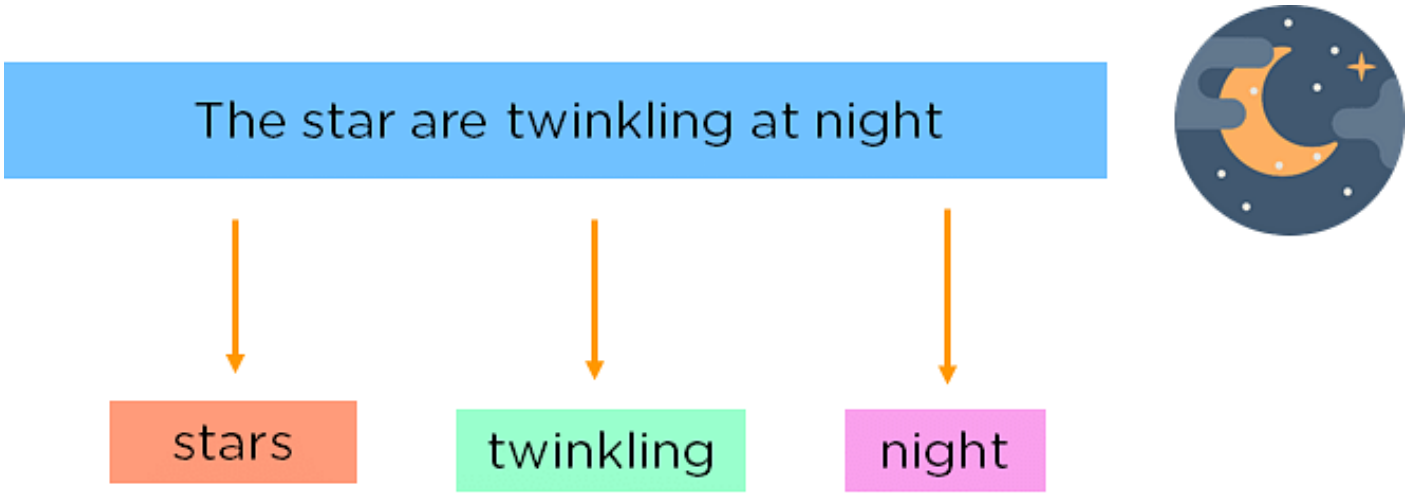
NLP : Text , Tokens

- **Tokenization:**
- We can see that unlike all the machine learning datasets we have worked with previously, the data isn't boolean, numeric, categorical etc. Usually a text is composed of paragraphs, paragraphs are composed of sentences, and sentences are composed of words. You could also go deeper into letters, but the letters have no meaning. It's only when they are combined into words, that the text starts to make sense. Hence, it is better to work at the word level.
- Tokenization is the process of splitting the text into smaller parts called tokens. Tokens are the basic units of a particular dataset. The choice of tokens could be based on the application we are working on.



NLP : Stop words

- You can make the learning process faster by getting rid of non-essential words, which add little meaning to our statement and are just there to make our statement sound more cohesive.
- Words such as was, in, is, and, the, are called stop words and can be removed.



Regex

Regex

- A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern.
- The Python module **re** provides full support regular expressions in Python.
- The re module offers a set of functions that allows us to search a string for a match:

Function	Description
findall	Returns a list containing all matches
search	Returns a Match object if there is a match anywhere in the string
split	Returns a list where the string has been split at each match
sub	Replaces one or many matches with a string

Regex

- A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern.
- The Python module **re** provides full support regular expressions in Python.
- The re module offers a set of functions that allows us to search a string for a match:

Function	Description
findall	Returns a list containing all matches
search	Returns a Match object if there is a match anywhere in the string
split	Returns a list where the string has been split at each match
sub	Replaces one or many matches with a string

Regex

- **Metacharacters**
- Metacharacters are characters with a special meaning:

Character	Description	Example
[]	A set of characters	"[a-m]"
\	Signals a special sequence (can also be used to escape special characters)	"\d"
.	Any character (except newline character)	"he..o"
^	Starts with	"^hello"
$	Ends with	"planet$"
*	Zero or more occurrences	"he.*o"
+	One or more occurrences	"he.+o"
?	Zero or one occurrences	"he.?o"
{}	Exactly the specified number of occurrences	"he.{2}o"
|	Either or	"falls|stays"

Regex

- **Special Sequences**
- A special sequence is a \ followed by one of the characters in the list below, and has a special meaning:

Character	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"

Regex

- **Sets**
- A set is a set of characters inside a pair of square brackets [] with a special meaning:

Set	Description
[arn]	Returns a match where one of the specified characters (a, r, or n) is present
[a-n]	Returns a match for any lower case character, alphabetically between a and n
[^arn]	Returns a match for any character EXCEPT a, r, and n
[0123]	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
[0-9]	Returns a match for any digit between 0 and 9
[0-5][0-9]	Returns a match for any two-digit numbers from 00 and 59
[a-zA-Z]	Returns a match for any character alphabetically between a and z, lower case OR upper case
[+]	In sets, +, *, ., |, (), $,{} has no special meaning, so [+] means: return a match for any + character in the string

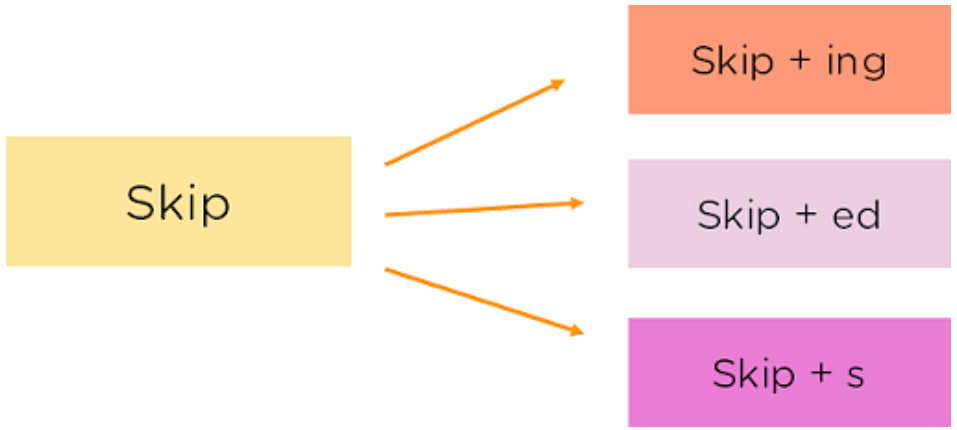
Stemming & Lemmatization

NLP : Stemming and lemmatization

- Owing to grammatical reasons, documents are going to use different forms of a word, such as discuss, discusses and discussing. Along with there are families of derivationally related words with similar meanings, such as liberal, liberty, and liberalization.
- Stemming and Lemmatization are Text Normalization (or sometimes called Word Normalization) techniques in the field of NLP that are used to prepare text, words, and documents for further processing.
- The goal of both the methods(stemming and lemmatization) is to reduce inflectional forms and derivationally related forms of a word to a common base form.
- For eg:
- am, are, is ⇒ be
- lion, lions, lion's, lions' ⇒ lion

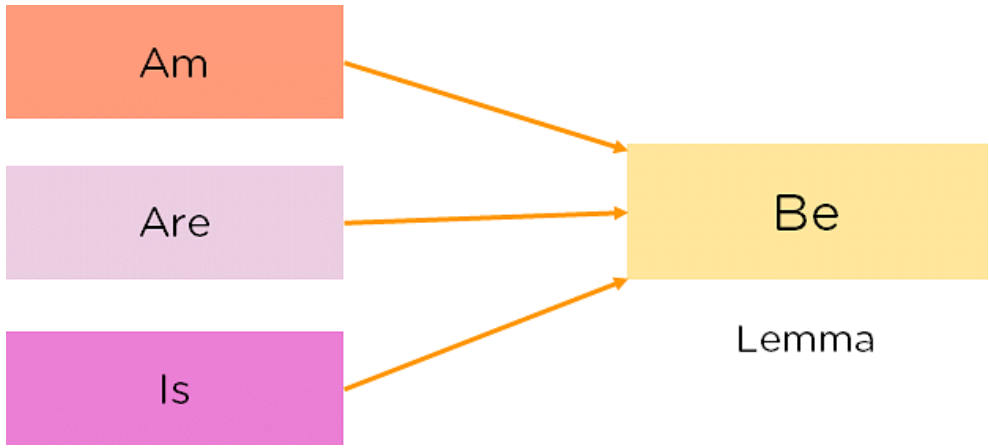
NLP : Stemming

- **Stemming** is the process of converting the words of a sentence to its non-changing portions. So stemming a word or sentence may result in words that are not actual words. Stems are created by removing the suffixes or prefixes used with a word.
- For eg: Likes, liked, likely, unlike ⇒ like
- Lot of different algorithms have been defined for the process, each with their own set of rules. The popular ones include:
  - Porter Stemmer(Implemented in almost all languages)
  - Paice Stemmer
  - Lovins Stemmer



NLP : lemmatization

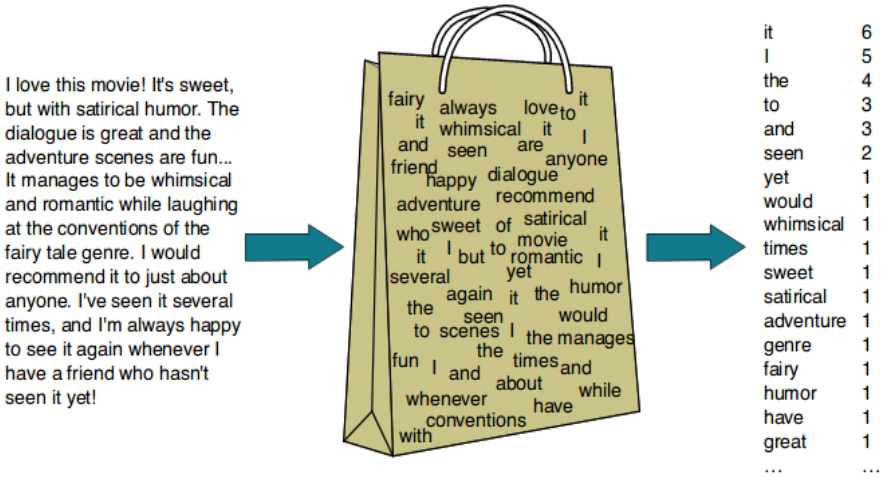
- This method is a more refined way of breaking words through the use of a vocabulary and morphological analysis of words. The aim is to always return the base form of a word known as lemma.
- Consider the following words:
  - 'Studied', 'Studious' ,'Studying'
  - Stemming of them will result in Studi
  - Lemmatisation of them will result in Study
- As it can be seen Lemmatization is more complex than stemming because it requires words to be categorized by a part-of-speech as well as by inflected form.



Bag of words

NLP : BOW

- The problem with modeling text is that there is no well defined fixed-length inputs.
- A bag of words model is a way of extracting features from text for use in modeling. In this approach, we use the tokenized words for each observation and find out the frequency of each token.
- A bag-of-words is a representation of text that describes the occurrence of words within a document. It involves two things:
  - A vocabulary of known words.
  - A measure of the presence of known words.



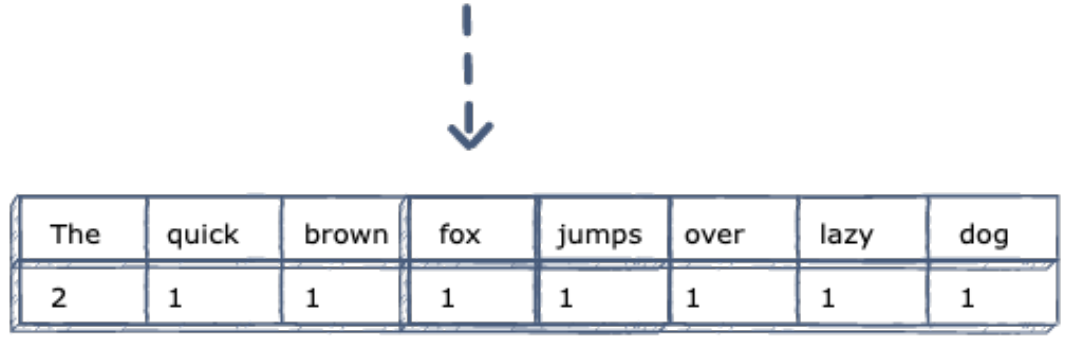
NLP : BOW

- Let's take an example to understand it, consider the following sentences:
  - "Hope is a good thing" ; "Maybe the best thing" ; "No good thing ever dies"
- We will treat each sentence as a different document and make a list of all unique words from the three documentations. We get:
  - "hope", "is", "a", "good", "thing", "maybe", "the", "best", "no", "ever", "dies"
- Next, we try to create vectors from it.
- In this, we take the first document = "Hope is a good thing" and check the frequency of words from the 10 unique words:
  - "hope" - 1 "is" - 1 "a" - 1 "good" - 1 "thing" - 1 "maybe" - 0 "the"-0 "best" - 0 "no" - 0 "ever" - 0 "dies" - 0
- Following is how each document will look like:
  - "Hope is a good thing" - [1,1,1,1,1,0,0,0,0,0,0]
  - "Maybe the best thing" - [0,0,0,0,1,1,1,1,0,0]
  - "No good thing ever dies" - [0,0,0,1,1,0,0,0,0,1,1]
- This process of converting text data to numbers is called vectorization
- There are multiple methods to convert words to numbers.

Count Vectorizer

- Count Vectorizer works on term frequency and building a sparse matrix of documents x tokens.
- For e.g. In the dataset in which we are working, the X column is a list of lowercased words
- The idea now is to convert the X column to numbers.
- One way to do that would be to represent every word as a key value pair in the form of a dictionary, where the key would be the word and the value would be the number of times that word has appeared in the list.
- This method of converting the counts of words in the list to convert them to a numeric format is called Count vectorization.

Data = ['The', 'quick', 'brown', 'fox', 'jumps', 'over', ' the', 'lazy', 'dog']

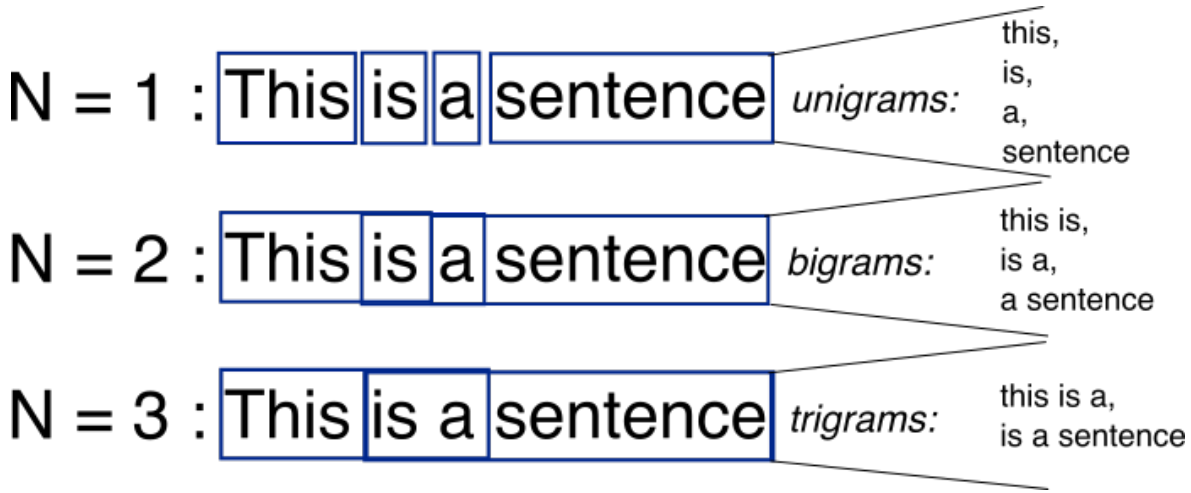


	The	quick	brown	fox	jumps	over	lazy	dog
Data	2	1	1	1	1	1	1	1

N-Gram

N-Gram

- N-grams are continuous sequences of words or symbols, or tokens in a document.
- In technical terms, they can be defined as the neighbouring sequences of items in a document. They come into play when we deal with text data in NLP (Natural Language Processing) tasks.
- They have a wide range of applications, like language models, semantic features, spelling correction, machine translation, text mining, etc.



N-Gram

- We are trying to teach machine how to do natural language processing. We human can understand language easily but machines cannot so we trying to teach them specific pattern of language. As specific word has meaning but when we combine the words(i.e group of words) than it will be more helpful to understand the meaning.
- n-gram is basically set of occurring words within given window so when
  - n=1 it is Unigram
  - n=2 it is bigram
  - n=3 it is trigram and so on
- Now suppose machine try to understand the meaning of sentence "I have a lovely dog" then it will split sentences into a specific chunk.
- It will consider word one by one which is unigram so each word will be a gram.
- "I", "have", "a" , "lovely" , "dog"
- It will consider two words at a time so it will be bigram so each two adjacent words will be bigram
- "I have" , "have a" , "a lovely" , "lovely dog"
- So like this machine will split sentences into small group of words to understand its meaning

TF-IDF

TF-IDF Vectorizer

- A count vectorizer, counts the occurrences of the words in a document and all the documents are considered independent of each other. Very similar to a one hot encoding or pandas getdummies function. However in cases where multiple documents are involved, count vectorizer still does not assume any interdependence between the documents and considers each of the documents as a separate entity.
- It does not rank the words based on their importance in the document, but just based on whether they exist or not.
- This is not a wrong approach, but it intuitively makes more sense to rank words based on their importance in the document right? In fact, the process of converting, text to numbers should essentially be a ranking system of the words so that the documents can each get a score based on what words they contain.
- All words cannot have the same importance or relevance in the document right?
- There are two ways to approach document similarity:
  - TF-IDF Score
  - Cosine Similarity

TF-IDF Vectorizer

- TF-IDF or Term Frequency and Inverse Document Frequency
- TF-IDF takes it a step further and ranks the words based not just on their occurrences in one document but across all the documents.
- Hence if CV or Count vectorizer was giving more importance to words because they have appeared multiple times in the document, TF-IDF will rank them high if they have appeared only in that document, meaning that they are rare, hence higher importance and lower if they have appeared in all or most documents, because they are more common, hence lower ranking.

$$w_{x,y} = tf_{x,y} \times \log\left(\frac{N}{df_x}\right)$$

TF-IDF

Term x within document y

$tf_{x,y}$ = frequency of x in y

df_x = number of documents containing x

N = total number of documents

TF-IDF Vectorizer

- TF: Term Frequency, which measures how frequently a term occurs in a document. Since every document is different in length, it is possible that a term would appear much more times in long documents than shorter ones. Thus, the term frequency is often divided by the document length (aka. the total number of terms in the document) as a way of normalization:
  - **TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document).**
- IDF: Inverse Document Frequency, which measures how important a term is. While computing TF, all terms are considered equally important. However it is known that certain terms, such as "is", "of", and "that", may appear a lot of times but have little importance. Thus we need to weigh down the frequent terms while scale up the rare ones, by computing the following:
  - **IDF(t) = log_e(Total number of documents / Number of documents with term t in it).**
- Consider a document containing 100 words wherein the word cat appears 3 times. The term frequency (i.e., tf) for cat is then (3 / 100) = 0.03. Now, assume we have 10 million documents and the word cat appears in one thousand of these. Then, the inverse document frequency (i.e., idf) is calculated as log(10,000,000 / 1,000) = 4. Thus, the Tf-idf weight is the product of these quantities: 0.03 * 4 = 0.12.

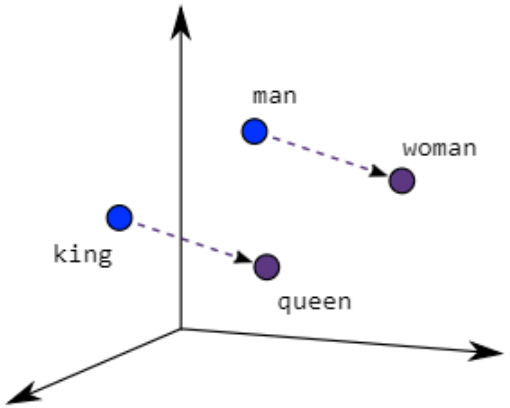
Word2Vec

Word Embedding

- Let us consider the two sentences – “You can scale your business.” and “You can grow your business.”. These two sentences have the same meaning. If we consider a vocabulary considering these two sentences, it will constitute of these words: {You, can, scale, grow, your, business}.
- A one-hot encoding of these words would create a vector of length 6. The encodings for each of the words would look like this:
- You: [1,0,0,0,0,0], Can: [0,1,0,0,0,0], Scale: [0,0,1,0,0,0], Grow: [0,0,0,1,0,0],
- Your: [0,0,0,0,1,0], Business: [0,0,0,0,0,1]
- In a 6-dimensional space, each word would occupy one of the dimensions, meaning that none of these words has any similarity with each other – irrespective of their literal meanings.
- **Word2Vec, a word embedding methodology**, solves this issue and enables similar words to have similar dimensions and, consequently, helps bring context.
- There are two main architectures which yield the success of word2vec. **The skip-gram and CBOW architectures.**

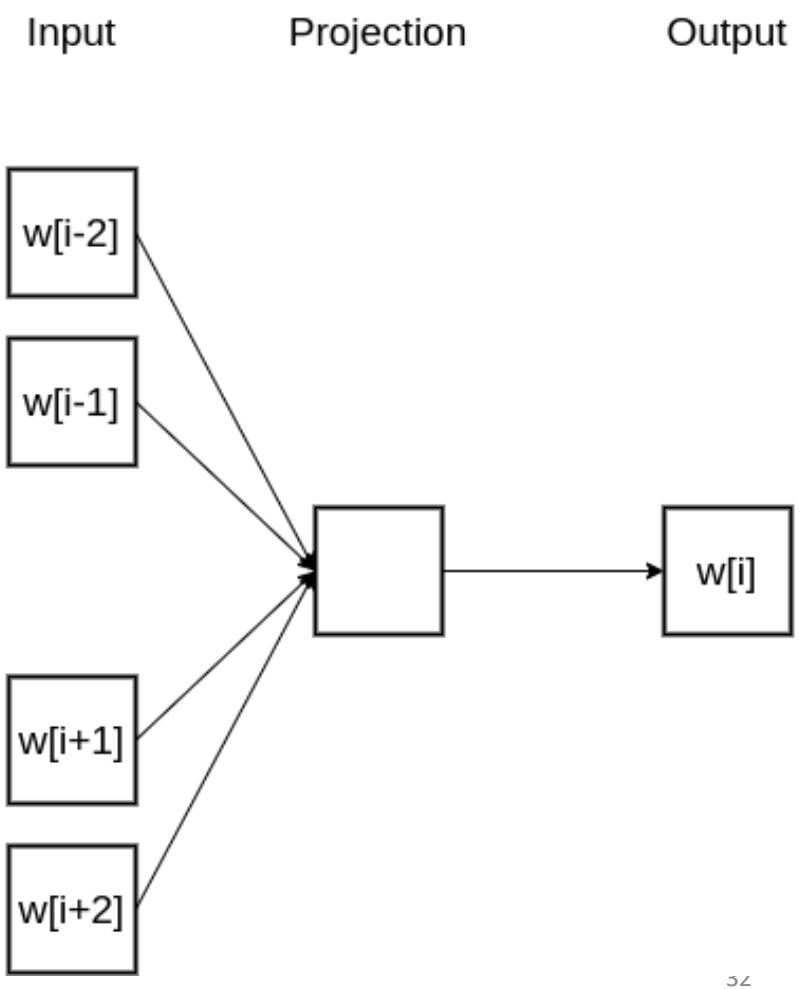
Word2Vec

- In Bag of Words and TF-IDF, we saw how every word was treated as an individual entity, and semantics were completely ignored. With the introduction of Word2Vec, the vector representation of words was said to be contextually aware, probably for the first time ever.
- Perhaps, one of the most famous examples of Word2Vec is the following expression:
  - king – man + woman = queen
- Since every word is represented as an n-dimensional vector, one can imagine that all of the words are mapped to this n-dimensional space in such a manner that words having similar meanings exist in close proximity to one another in this hyperspace.
- Word2Vec creates vectors of the words that are distributed numerical representations of word features – these word features could comprise of words that represent the context of the individual words present in our vocabulary. Word embeddings eventually help in establishing the association of a word with another similar meaning word through the created vectors.



Word2Vec : CBOW

- This architecture is very similar to a feed forward neural network.
- This model architecture essentially tries to predict a target word from a list of context words.
- The intuition behind this model is quite simple: given a phrase "Have a great day" , we will choose our target word to be “a” and our context words to be [“have”, “great”, “day”].
- What this model will do is take the distributed representations of the context words to try and predict the target word.

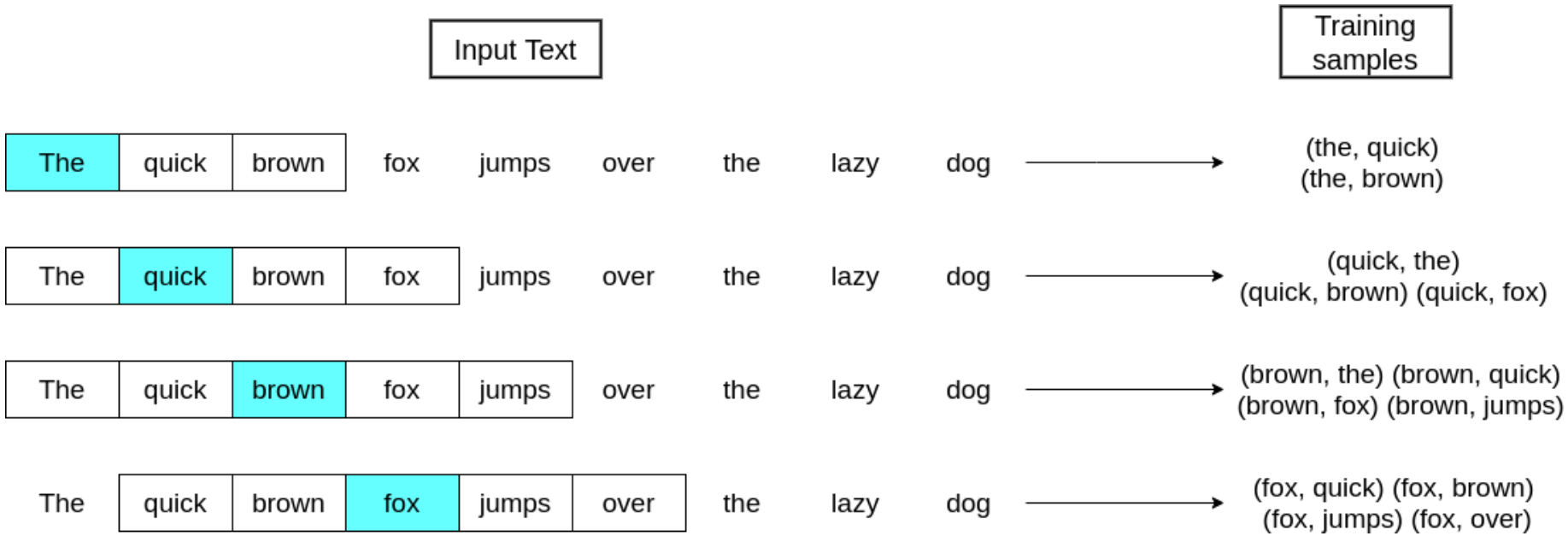


Word2Vec: Skip- gram model

- The skip-gram model is a simple neural network with one hidden layer trained in order to predict the probability of a given word being present when an input word is present. Intuitively, you can imagine the skip-gram model being the opposite of the CBOW model.
- In this architecture, it takes the current word as an input and tries to accurately predict the words before and after this current word.
- This model essentially tries to learn and predict the context words around the specified input word. Based on experiments assessing the accuracy of this model it was found that the prediction quality improves given a large range of word vectors, however it also increases the computational complexity.
- The interesting part is, we don't actually use this trained Neural Network. Instead, the goal is just to learn the weights of the hidden layer while predicting the surrounding words correctly. These weights are the word embeddings.
- How many neighbouring words the network is going to predict is determined by a parameter called "window size". This window extends in both the directions of the word, i.e. to its left and right.
- Let's say we want to train a skip-gram word2vec model over an input sentence:
  - **"The quick brown fox jumps over the lazy dog"**

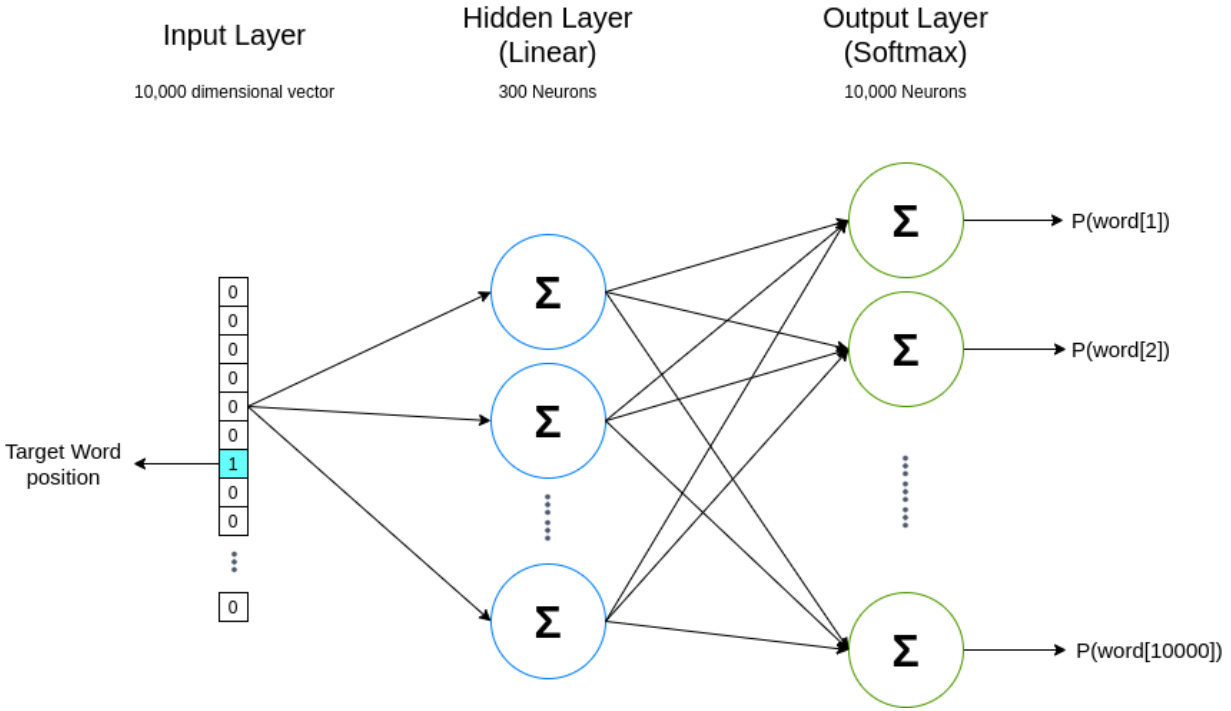
Word2Vec: Skip- gram model

- 'The' becomes the first target word and since it's the first word of the sentence, there are no words to its left, so the window of size 2 only extends to its right resulting in the listed training samples.
- As our target shifts to the next word, the window expands by 1 on left because of the presence of a word on the left of the target.
- Finally, when the target word is somewhere in the middle, training samples get generated as intended.



Word2Vec: Skip- gram model

- Let's say our vocabulary contains around 10,000 words and our current target word 'fox' is present somewhere in between. What we'll do is, put a 1 in the position corresponding to the word 'fox' and 0 everywhere else, so we'll have a 10,000-dimensional vector with a single 1 as the input.
- Similarly, the output coming out of our network will be a 10,000-dimensional vector as well, containing, for every word in our vocabulary, the probability of it being the context word for our input target word.



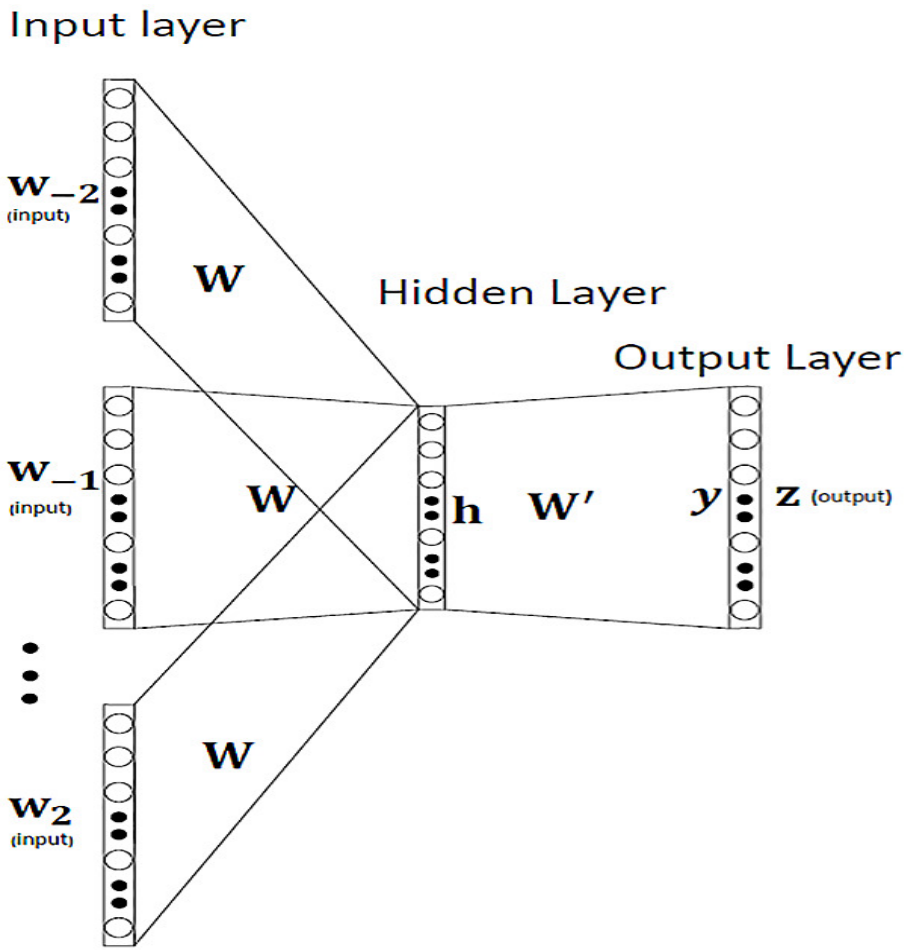
Word2Vec: Skip- gram model

- As it can be seen that input is a 10,000-dimensional vector given our vocabulary size =10,000, containing a 1 corresponding to the position of our target word.
- The output layer consists of 10,000 neurons with the Softmax activation function applied, so as to obtain the respective probabilities against every word in our vocabulary.
- Now the most important part of this network, the hidden layer is a linear layer i.e. there's no activation function applied there, and the optimized weights of this layer will become the learned word embeddings.
- For example, let's say we decide to learn word embeddings with the above network. In that case, the hidden layer weight matrix shape will be M x N, where M = vocabulary size (10,000 in our case) and N = hidden layer neurons (300 in our case).
- Once the model gets trained, the final word embedding for our target word will be given by the following calculation:

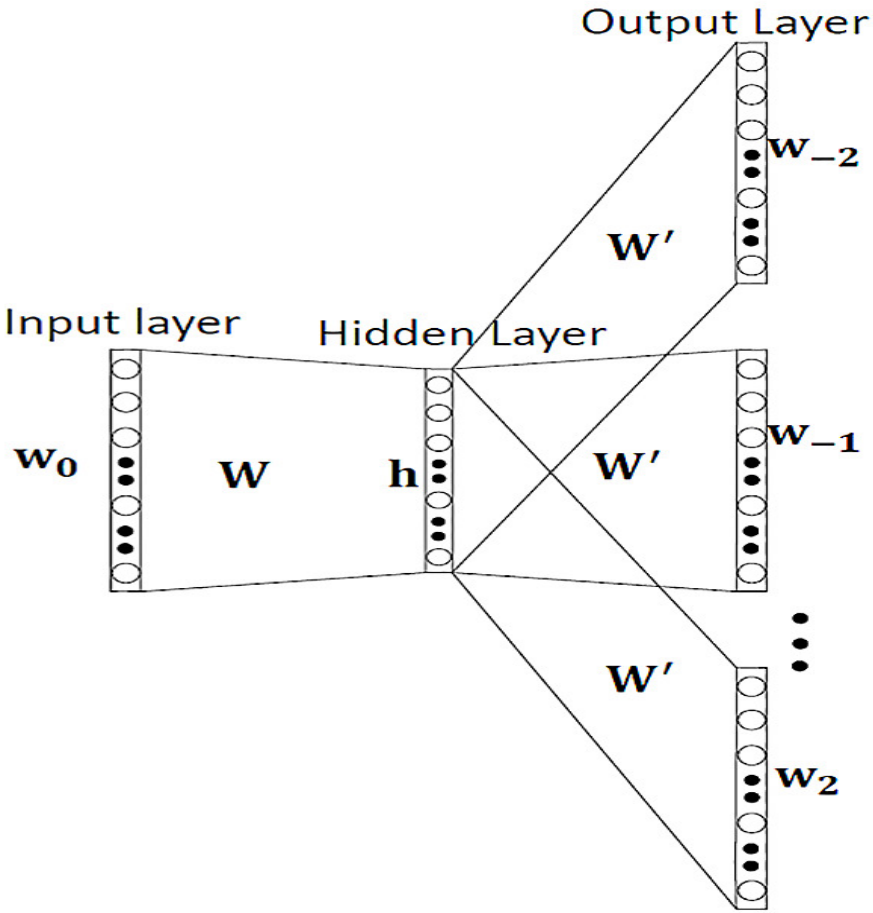
$$1\times10000 \text{ input vector} * 10000\times300 \text{ matrix} = 1\times300 \text{ vector}$$

- 300 hidden layer neurons were used by Google in their trained model, however, this is a hyperparameter and can be tuned accordingly to obtain the best results.

Word2Vec: CBOW / Skip- gram model



(a) CBOW



(b) Skip-Gram


Thank you!