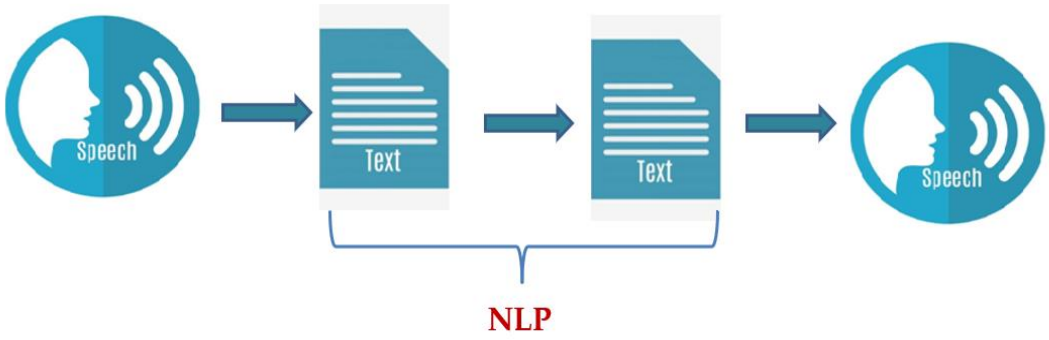
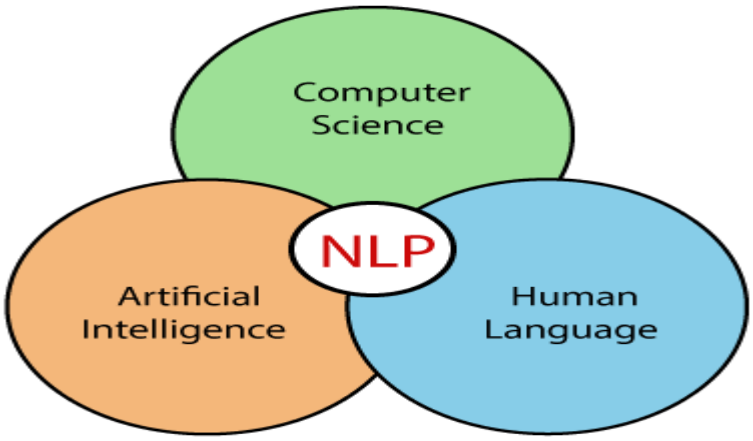
INTRODUCTION TO NLP

- Natural language processing (NLP) is an area of computer science and artificial intelligence concerned with the interaction between computers and humans in natural language.
- The ultimate goal of NLP is to help computers understand language like we do.
- Natural language processing (NLP) is a field of artificial intelligence in which computers analyze, understand, and derive meaning from human language in a smart and useful way.
- **NLP strives** to build machines that **understand *text or voice*** and **respond** in *text or voice* data in the same way humans do



NEED FOR NLP

Computers can understand the structured form of data like spreadsheets and the tables in the database, but human languages, texts, and voices form an unstructured category of data, and it gets difficult for the computer to understand it, and there arises the need for Natural Language Processing.

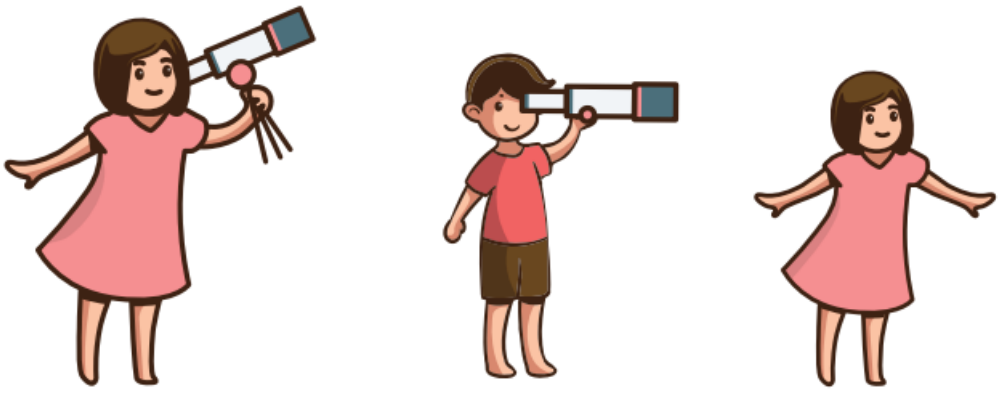


The word “natural” here is used to contrast natural languages with formal languages. In this sense, all the languages humans speak are natural. Many experts believe that language emerged naturally tens of thousands of years ago and has evolved organically ever since. Formal languages, on the other hand, are types of languages that are invented by humans and have strictly and explicitly defined syntax (i.e., what is grammatical) and semantics (i.e., what it means).

Programming languages such as C and Python are good examples of formal languages. These languages are defined in such a strict way that it is always clear what is grammatical and ungrammatical. When you run a compiler or an interpreter on the code you write in those languages, you either get a syntax error or not. The compiler won’t say something like, “Hmm, this code is maybe 50% grammatical.” Also, the behaviour of your program is always the same if it’s run on the same code, assuming external factors such as the random seed and the system states remain constant. Your interpreter won’t show one result 50% of the time and another the other 50% of the time.

This is not the case for human languages. You can write a sentence that is maybe grammatical. For example, do you consider the phrase “The person I spoke to” ungrammatical? There are some grammar topics where even experts disagree with each other. This is what makes human languages interesting but challenging, and why the entire field of NLP even exists. Human languages are ambiguous, meaning that their interpretation is often not unique. Both structures (how sentences are formed) and semantics (what sentences mean) can have ambiguities in human language. As an example,

- As an example, ***“ He Saw a girl with a telescope ”***



- Is it the boy, who’s using a telescope to see a girl (from somewhere far), or

- The girl, who has a telescope and is seen by the boy?

There seem to be at least two interpretations of this sentence

- The reason you are confused upon reading this sentence is because you don't know what the phrase "with a telescope" is about.
- More technically, you don't know what this prepositional phrase (PP) modifies.
- This is called a PP-attachment problem and is a classic example of **syntactic ambiguity.** A syntactically ambiguous sentence has more than one interpretation of how the sentence is structured. You can interpret the sentence in multiple ways, depending on which structure of the sentence you believe

Another type of ambiguity that may arise in natural language

"I saw a bat"

There is no question how this sentence is structured. The subject of the sentence is "I" and the object is "a bat," connected by the verb "saw."

In other words, there is no syntactical ambiguity in it. But how about its meaning? "Saw" has at least two meanings. One is the past tense of the verb "to see." The other is to cut some object with a saw.

Similarly, "a bat" can mean two very different things: is it a nocturnal flying mammal or a piece of wood used to hit a ball?

All in all, does this sentence mean that I observed a flying mammal or that I cut a baseball or cricket bat? Or even (cruelly) that I cut a nocturnal animal with a saw? You never know, at least from this sentence alone.

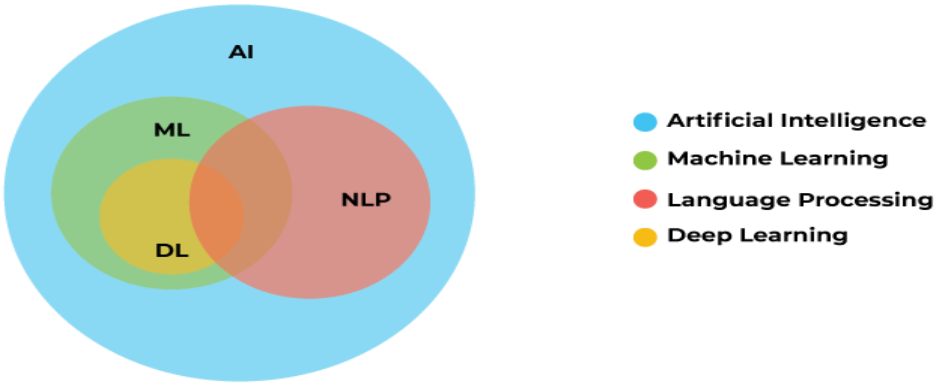
- Ambiguity is what makes natural languages rich but also challenging to process.
- **Human languages** are interesting but challenging, they are ambiguous, meaning that their **interpretation** is often not unique.
- Both **structures** (*how sentences are formed*) and **semantics** (*what sentences mean*) can have ambiguities in human language.

HOW NLP WORKS?

- NLP combines the field of linguistics and computer science to decipher language structure and guidelines to make models which can comprehend, break down and separate significant details from text and speech.
- NLP involves a variety of techniques, including computational linguistics, machine learning, and statistical modeling. These techniques are used to analyze, understand, and manipulate human language data, including text, speech, and other forms of communication.
- NLP includes a range of algorithms, tasks, and problems that take human-produced text as an input and produce some useful information, such as labels, semantic representations, and so on, as an output.
- Other tasks, such as translation, summarization, and text generation, directly produce text as output.

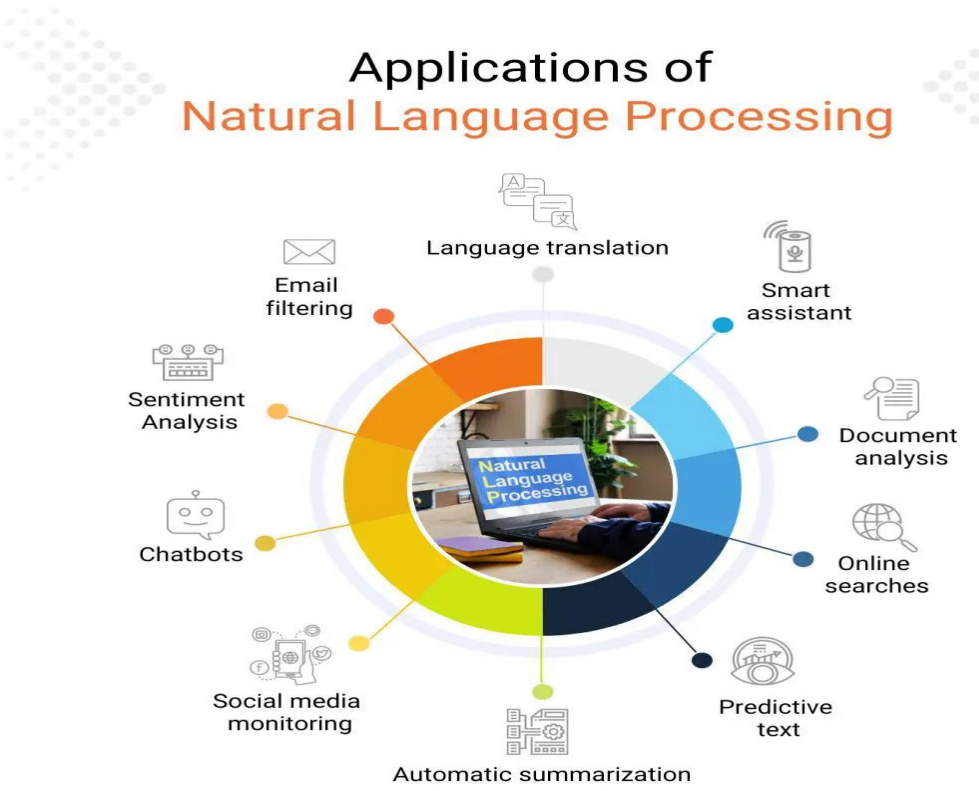
AI,ML,DL & NLP

- Natural Language Processing, Machine Learning, and Artificial Intelligence are used interchangeably, yet they have different definitions.
- AI is an umbrella term for machines that can simulate human intelligence, while ML,DL and NLP are subsets of AI.



- **Artificial Intelligence** is a part of the greater field of Computer Science that enables computers to solve problems previously handled by biological systems.

- **Machine Learning** is an application of AI that provides systems the ability to automatically learn and improve from experience without being explicitly programmed.
- **Deep learning** is a method in artificial intelligence (AI) that teaches computers to process data in a way that is inspired by the human brain. Deep learning models can recognize complex patterns in pictures, text, sounds, and other data to produce accurate insights and predictions.
- **Natural Language Processing** is a form of AI that gives machines the ability to not just read, but to understand and interpret human language.



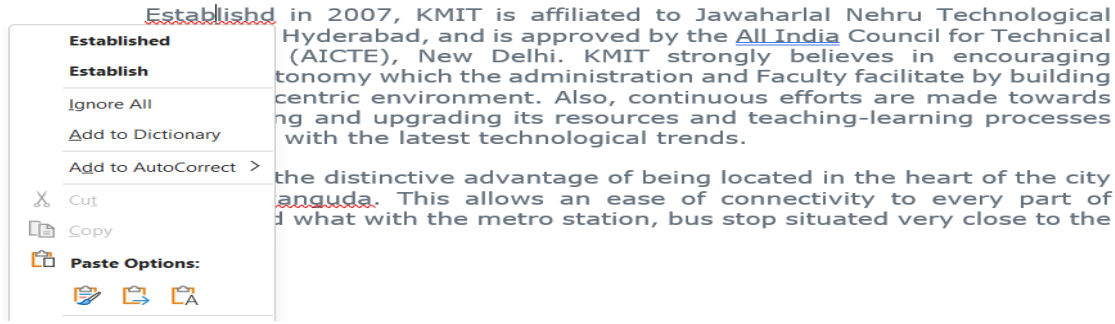
Smart Assistants

Amazon's Alexa and Apple's Siri are two of the prime examples of such interaction where humans use speech to interact with the system and perform different tasks.

Another example of natural interaction is Google's homepage where you can perform search operations via speech. Natural language processing lays at the foundation of such interaction.

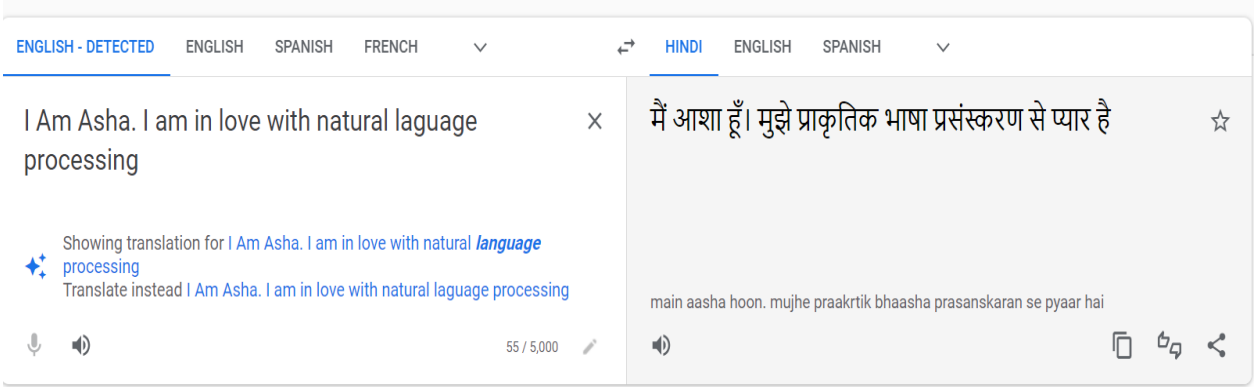
Spelling correction

Microsoft Corporation provides word processor software like MS-word, PowerPoint for the spelling correction.



Machine Translation

Machine translation is used to translate text or speech from one natural language to another natural language.

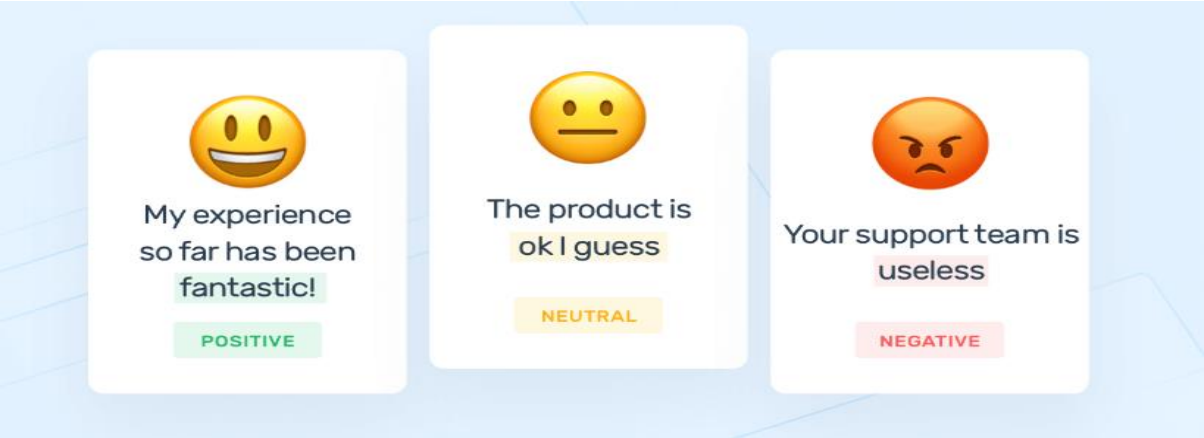


Chatbot

Implementing the Chatbot is one of the important applications of NLP. It provides the customer chat services

Sentiment Analysis

Sentiment Analysis is also known as **opinion mining.** It is used on the web to analyse the attitude, behaviour, and emotional state. This application is implemented through a combination of NLP and statistics by assigning the values to the text, identify the mood of the context.

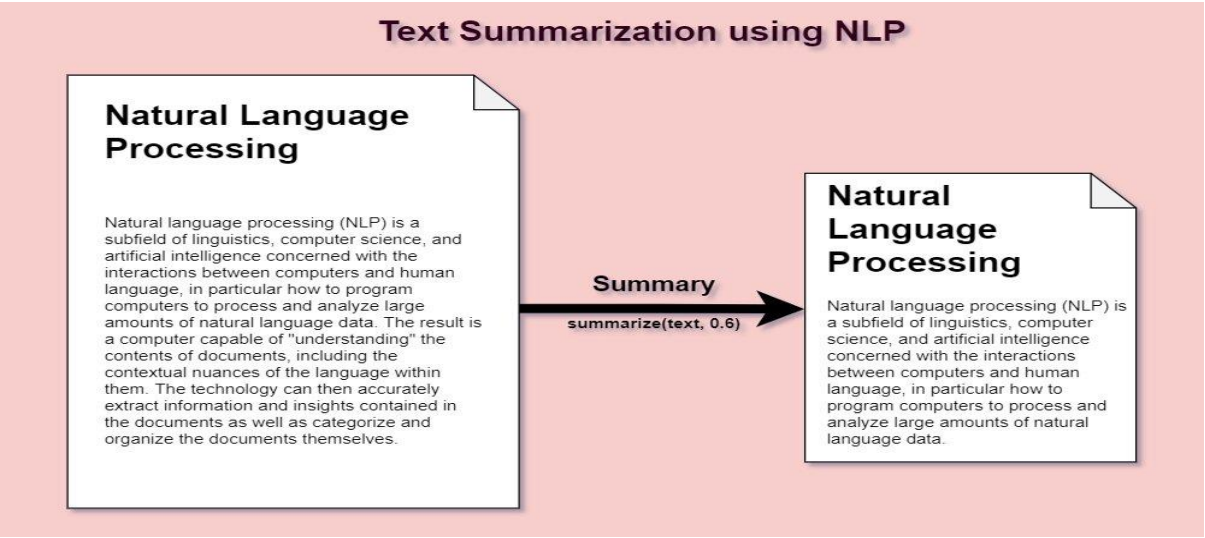


Email Filtering

It checks/ analyzing incoming emails for red flags that signal spam or phishing content and then automatically moving those emails to a separate folder.

eg: look for common trigger words,

such as "free" and "earn money".


Text Summarization using NLP
Natural Language Processing
Natural language processing (NLP) is a subfield of linguistics, computer science, and artificial intelligence concerned with the interactions between computers and human language, in particular how to program computers to process and analyze large amounts of natural language data. The result is a computer capable of "understanding" the contents of documents, including the contextual nuances of the language within them. The technology can then accurately extract information and insights contained in the documents as well as categorize and organize the documents themselves.
Summary
summarize(text, 0.6)
Natural Language Processing
Natural language processing (NLP) is a subfield of linguistics, computer science, and artificial intelligence concerned with the interactions between computers and human language, in particular how to program computers to process and analyze large amounts of natural language data.

Why is NLP important?

NLP enables humans to interact with computers in natural language, making technology more accessible and easier to use for a wide range of people.

NLP allows machines to analyze and understand large amounts of text data, which is valuable for extracting insights, detecting patterns, and making data-driven decisions.

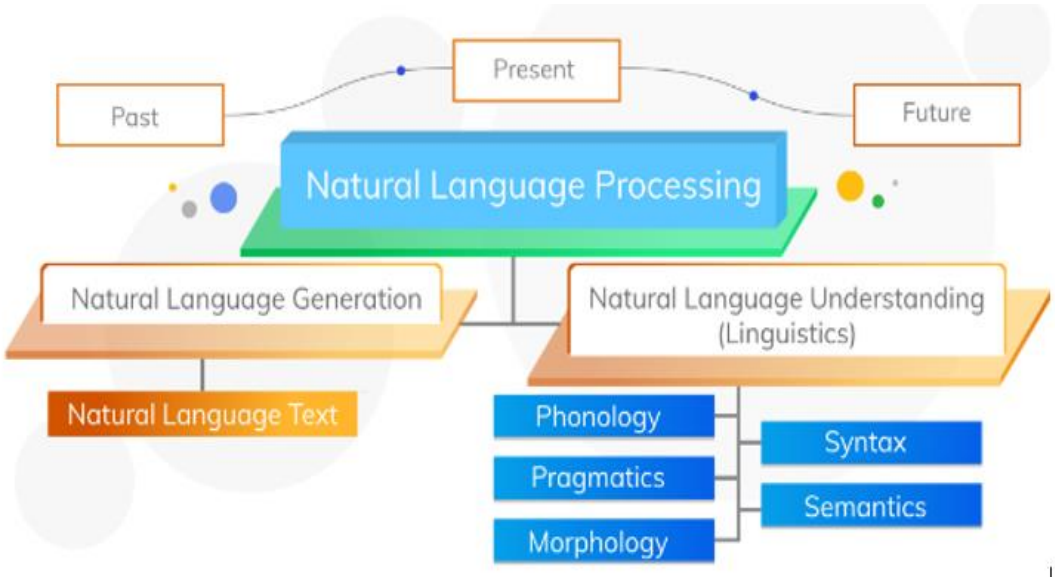
Large volumes of textual data

NLP makes it possible for computers to read text, hear speech, interpret it, measure sentiment etc.

Structuring a highly unstructured data source

NLP is important because it helps resolve ambiguity in language and adds useful numeric structure to the data for many downstream applications, such as speech recognition or text analytics.

Evolution of NLP



Evolving from human-computer interaction to human-computer conversation

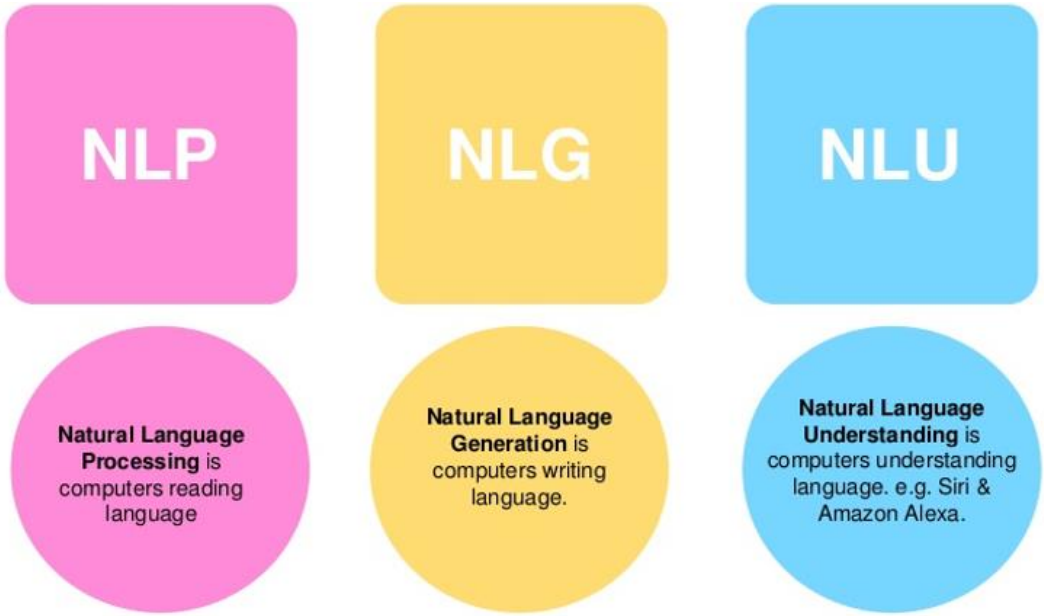
The first critical part of NLP Advancements – Biometrics

The second critical part of NLP advancements - Humanoid Robotics

NLP is broadly made of two parts:

Natural Language Understanding (NLU)

Natural Language Generation (NLG)



Natural Language Understanding (NLU)

NLU is branch of natural language processing (NLP), which helps computers understand and interpret human language by breaking down the elemental pieces of speech.

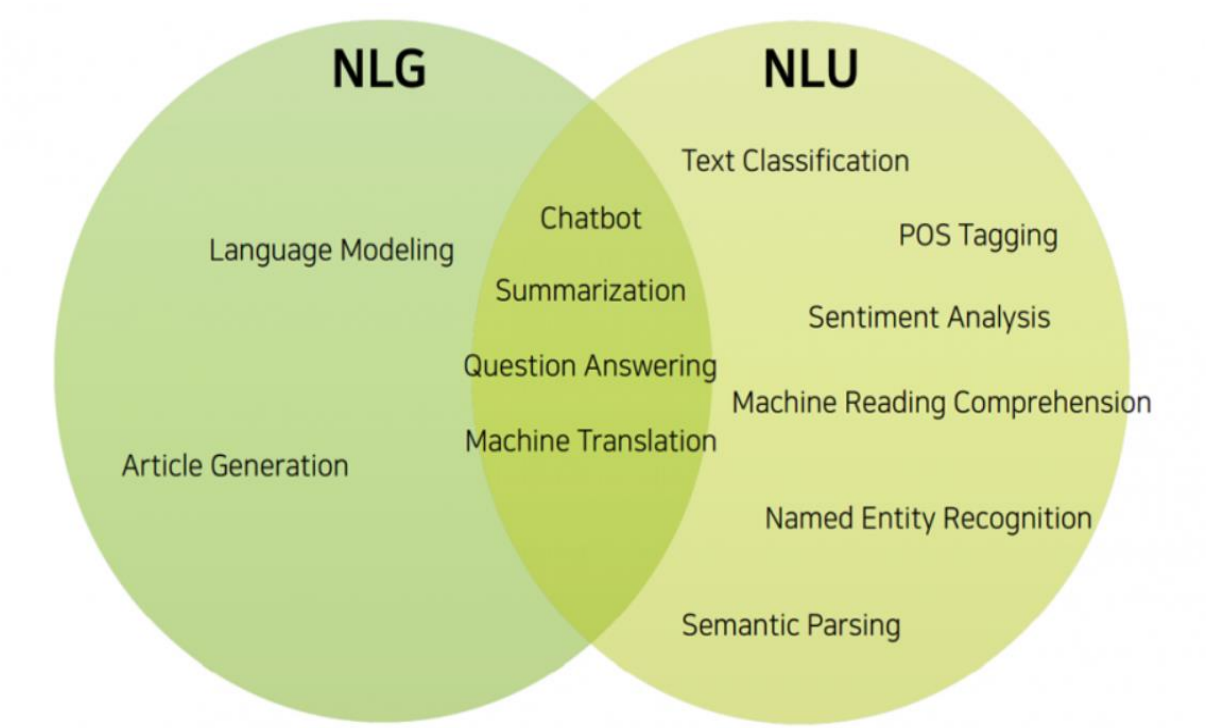
In NLU, machine learning models improve over time as they learn to recognize syntax, context, language patterns, unique definitions, sentiment, and intent.

NLU is a subset of natural language processing, which uses syntactic and semantic analysis of text and speech to determine the meaning of a sentence. Syntax refers to the grammatical structure of a sentence, while semantics alludes to its intended meaning. NLU also establishes a relevant ontology: a data structure which specifies the relationships between words and phrases. While humans naturally do this in conversation, the combination of these analyses is required for a machine to understand the intended meaning of different texts.

Natural Language Generation (NLG)

Natural language generation is another subset of natural language processing. While natural language understanding focuses on computer reading comprehension, natural language generation enables computers to write. NLG is the process of producing a human language text response based on some data input. This text can also be converted into a speech format through text-to-speech services.

Delivering a meaningful, personalized experience beyond pre-scripted responses requires natural language generation. This enables the chatbot to interrogate data repositories, including integrated back-end systems and third-party databases, and to use that information in creating a response.



NLU is about analysis. NLG is about synthesis.

An NLP application may involve one or both.

Sentiment analysis and semantic search are examples of NLU.

Captioning an image or video is mainly an NLG task since input is not textual.

Text summarization and chatbot are applications that involve NLU and NLG.

There's also Natural Language Interaction (NLI) of which Amazon Alexa and Siri are examples.

Challenges

Systems are as yet incapable of understanding the way humans do. Until then, progress will be limited to better pattern matching.

In the area of chatbots, there's a need to model common sense.

Africa alone has about 2100 languages. We need to find a way to solve this even if training data is limited.

Just measuring progress is a challenge. We need datasets and evaluation procedures tuned to concrete goals.

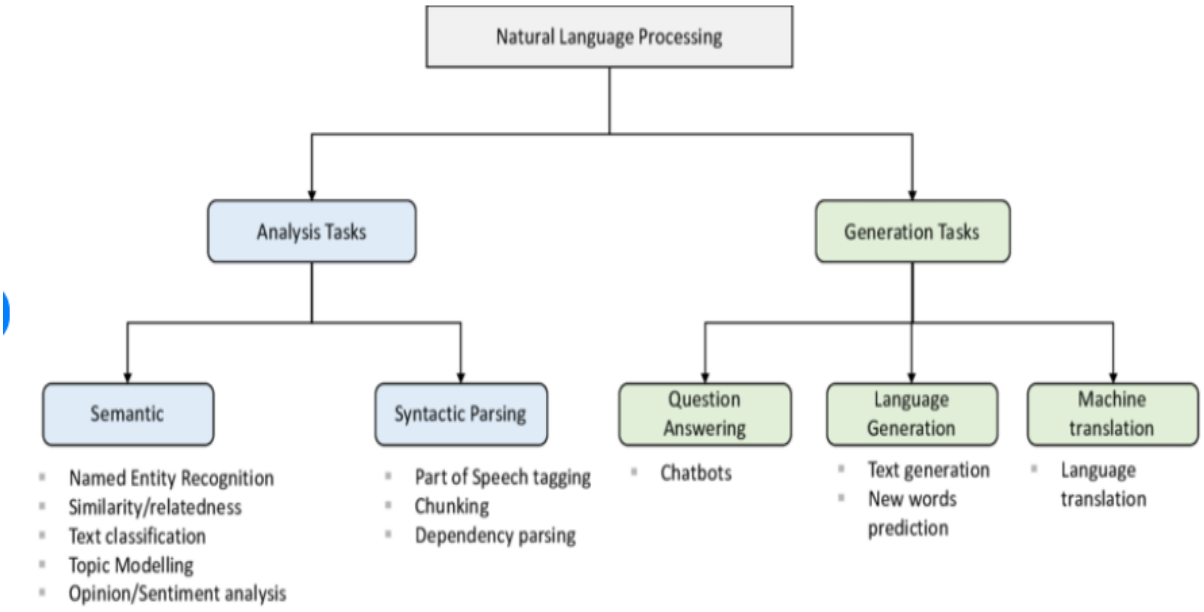
Language is inherently ambiguous, with words and phrases often having multiple meanings depending on context. Resolving ambiguity is challenging for NLP systems.

Understanding the context in which a word or phrase is used is crucial for accurately interpreting meaning. NLP systems need to be able to understand and use context effectively.

NLP models often require large amounts of annotated data for training, and obtaining such data can be costly and time-consuming, especially for languages with fewer resources.

NLP TASKS

NLP stands for Natural Language Processing, which is a field of artificial intelligence focused on enabling computers to understand, interpret, and generate human language. NLP tasks include



1. **Text Classification:**

Assigning predefined categories or labels to text, such as spam detection or sentiment analysis.

Text classification is the process of classifying pieces of text into different categories.

This NLP task is one of the simplest yet most widely used.

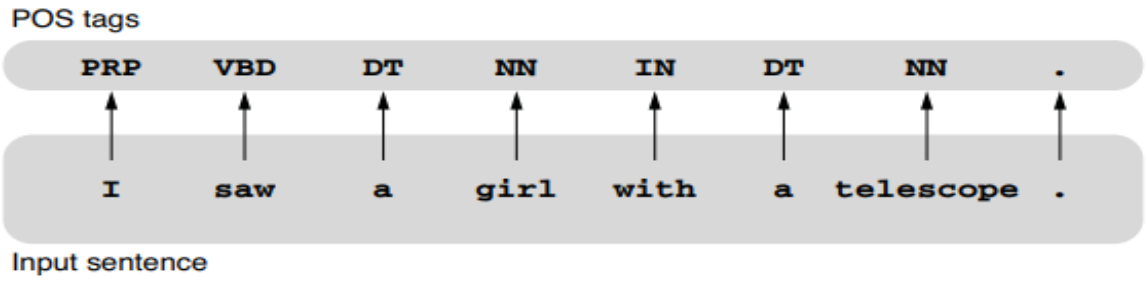
For example, spam filtering is one type of text classification. It classifies emails (or other types of text, such as web pages) into two categories—spam or not spam.

2. **PART-OF-SPEECH TAGGING**

Part-of-speech tagging is the process of tagging each word in a sentence with a corresponding part-of-speech tag.

As an example, let's take the sentence "I saw a girl with a telescope."

The POS tags for this sentence are shown in figure



POS tag	Description
DT	Determiner
IN	Preposition
NN	Noun (singular or mass)
PRP	Pronoun
VBD	Verb (past tense)

These tags come from the Penn Treebank POS tagset, which is the most popular standard corpus for training and evaluating various NLP tasks such as POS tagging and parsing. The results of POS tagging are often used as the input to other downstream NLP tasks, such as machine translation and parsing.

3. PARSING

Parsing is the task of analyzing the structure of a sentence.

Broadly speaking, there are two main types of parsing,

constituency parsing and

dependency parsing.

Constituency parsing uses context-free grammars to represent natural language sentences. A context-free grammar is a way to specify how smaller building blocks of a language (e.g.,

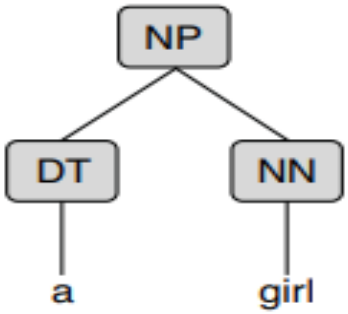
words) are combined to form larger building blocks (e.g., phrases and clauses) and eventually sentences. To put it another way, it specifies how the largest unit (a sentence) is broken down to phrases and clauses and all the way down to words. The ways the linguistic units interact with each other are specified by a set of production rules as follows:

```
S  -> NP VP

NP  -> DT NN  |  PRN  |  NP PP
VP  -> VBD NP  |  VBD PN PP
PP  -> IN NP

DT  -> a
IN  -> with
NN  -> girl  |  telescope
PRN  -> I
VBD  -> saw
```

A production rule describes a transformation from the symbol on the left-hand side (e.g., “S”) to the symbols on the right-hand side (e.g., “NP VP”).

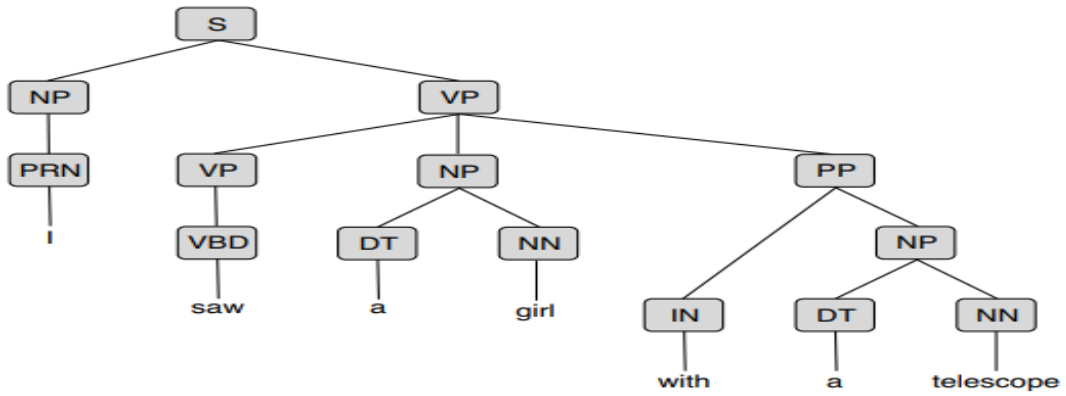
The first rule means that a sentence is a noun phrase (NP) followed by a verb phrase (VP). Now the parser’s job is to figure out how to reach the final symbol (in this case, “S”) starting from the raw words in the sentence. You can think of those rules as transformation rules from the symbols on the right to the ones on the left by traversing the arrow backward.

For example, using the rule “DT □ a” and “NN □ girl,” you can convert “a girl” to “DT NN.” Then, if you use “NP □ DT NN,” you can reduce the entire phrase to “NP.” If you illustrate this process in a tree-like diagram, you get something like the one shown in figure below.

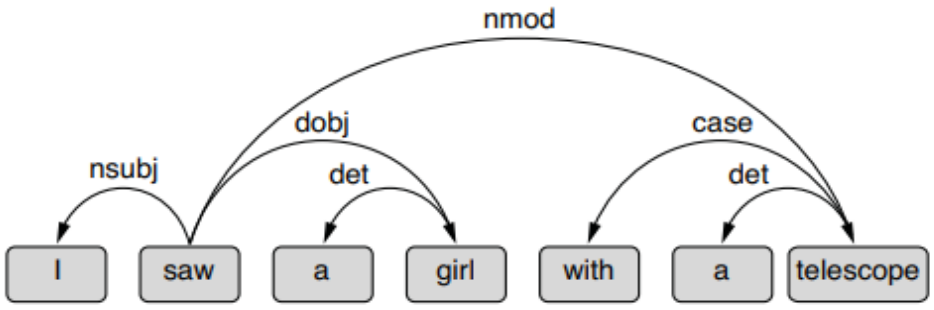


Tree structures that are created in the process of parsing are called parse trees, or simply parses. The figure is a subtree because it doesn’t cover the entirety of the tree (i.e., it doesn’t show all the way from “S” to words). Using the sentence “I saw a girl with a telescope” that

we discussed earlier and see if you can parse it by hand. If you keep breaking down the sentence using the production rules until you get the final “S” symbol, you get the tree-like structure shown in figure



The other type of parsing is called dependency parsing. Dependency parsing uses dependency grammars to describe the structure of sentences, not in terms of phrases but in terms of words and the binary relations between them. For example, the result of dependency parsing of the earlier sentence is shown in figure below



Notice that each relation is directional and labeled. A relation specifies which word depends on which word and the type of relationship between the two. For example, the relation connecting “a” to “girl” is labeled “det,” meaning the first word is the determiner of the second. If you take the most central word, “saw,” and pull it upward, you’ll notice that these words and relations form a tree. Such trees are called dependency trees

4. **TEXT GENERATION**

Text generation, also called natural language generation (NLG), is the process of generating natural language text from something else. Summarization, text simplification, and grammatical error correction all produce natural language text as output and are instances of text-generation tasks.

Because all of these tasks take natural language text as their input, they are called text-to-text generation.

Another class of text-generation task is called data-to-text generation. For those tasks, the input is data that is not text. A publisher may wish to generate news text based on events such as sports game outcomes and weather. There is also a growing interest in generating natural language text that best describes a given image, called image captioning.

Finally, a third class of text classification is unconditional text generation, where natural language text is generated randomly from a model. You can train models so that they can generate random academic papers, Linux source code, or even poems and play scripts

5. **Named Entity Recognition (NER):** Identifying and classifying named entities mentioned in text, such as names of people, organizations, and locations.

NER attempts to extract entities (for example, person, location, and organization) from a given body of text or a text corpus.

For example, the sentence, *John gave Mary two apples at school on Monday* will be transformed to *[John]* $_{name}$ *gave [Mary]* $_{name}$ *[two]* $_{number}$ *apples at [school]* $_{organization}$ *on [Monday.]* $_{time}$. NER is an imperative topic in fields such as information retrieval and knowledge representation.

6. **Question Answering:** Providing relevant answers to questions posed in natural language.

QA techniques possess a high commercial value, and such techniques are found at the foundation of chatbots and VA (for example, Google Assistant and Apple Siri).

Chatbots have been adopted by many companies for customer support. Chatbots can be used to answer and resolve straightforward customer concerns (for example, changing a customer's monthly mobile plan), which can be solved without human intervention.

QA touches upon many other aspects of NLP such as information retrieval, and knowledge representation. Consequently, all this makes developing a QA system very difficult.

7. **Machine Translation (MT)**

MT is the task of transforming a sentence/phrase from a source language (for example, German) to a target language (for example, English). This is a very challenging task as, different languages have highly different morphological structures, which means that it is not a one-to-one transformation. Furthermore, word-to-word relationships between languages can be one-to-many, one-to-one, many-to-one, or many-to-many.

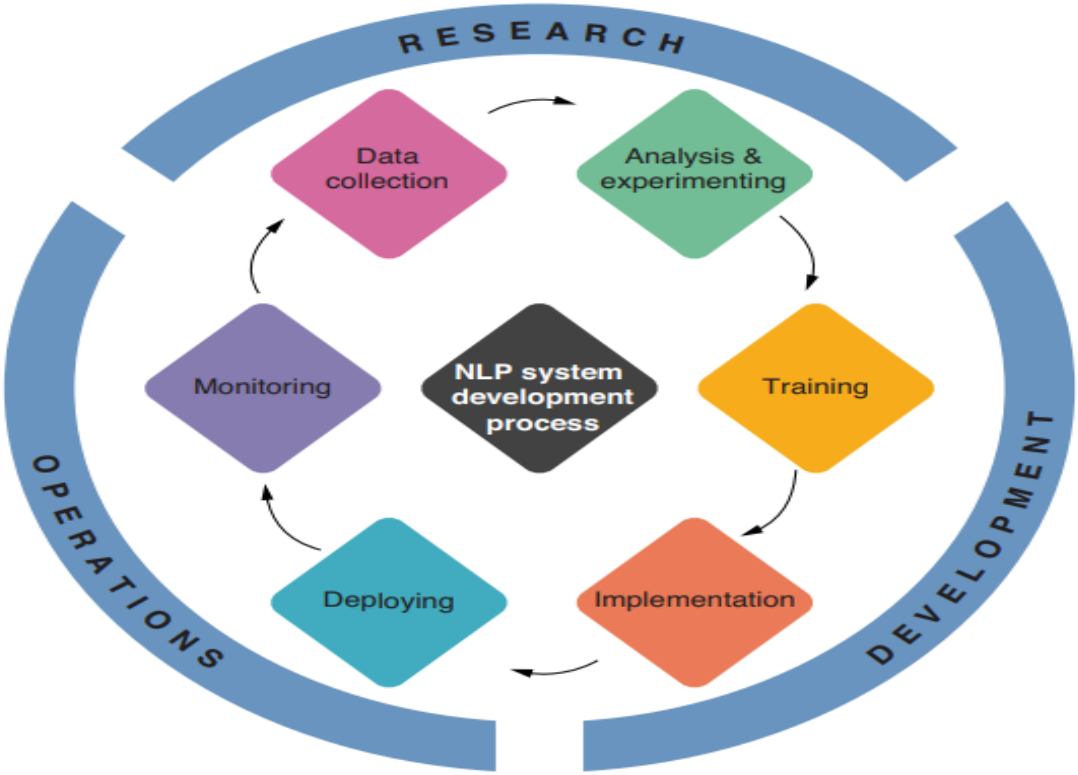
8. **Sentiment Analysis:** Determining the sentiment (positive, negative, or neutral) expressed in a piece of text.

9. **Language Generation:** Generating human-like text, such as in chatbots or automated content creation.

Development of NLP Applications

Structure of NLP Applications

The development of NLP applications is a highly iterative process, consisting of many phases of research, development, and operations.



DATA COLLECTION

- Most modern NLP applications are based on machine learning.
- Machine learning, by definition, requires data on which NLP models are trained. Data can be collected from humans (e.g., by hiring in-house annotators and having them go through a bunch of text instances), crowdsourcing (e.g., using platforms such as Amazon Mechanical Turk), or automated mechanisms (e.g., from application logs or clickstreams).

ANALYSIS AND EXPERIMENTING

- After collecting the data, you move on to the next phase where you analyze and run some experiments.
- For analyses, you usually look for signals such as: What are the characteristics of the text instances? How are the training labels distributed? Can you come up with signals

that are correlated with the training labels? Can you come up with some simple rules that can predict the training labels with reasonable accuracy? Should we even use ML? This list goes on and on.

- This analysis phase includes aspects of data science, where various statistical techniques may come in handy. The goal in this phase is to narrow down the possible set of approaches to a couple of promising ones, before you go all-in and start training a gigantic model.

- **TRAINING**

  This is when you start adding more data and computational resources (e.g., GPUs) for training your model. It is not uncommon for modern NLP models to take days if not weeks to train, especially if they are based on neural network models. It is critical at this phase that you keep your training pipeline reproducible. Chances are, you will need to run this several times with different sets of hyperparameters, which are tuning values set before starting the model's learning process. It is also likely that you will need to run this pipeline several months later, if not years

- **IMPLEMENTATION**

  When you have a model that is working with acceptable performance, you move on to the implementation phase. This is when you start making your application "production ready." This process basically follows software engineering best practices, including: writing unit and integration tests for your NLP modules, refactoring your code, having your code reviewed by other developers, improving the performance of your NLP modules, and dockerizing your application.

- **DEPLOYING**

  Your NLP application is finally ready to deploy. You can deploy your NLP application in many ways—it can be an online service, a recurring batch job, an offline application, or an offline one-off task.

  If this is an online service that needs to serve its predictions in real time, it is a good idea to make this a microservice to make it loosely coupled with other services.

  MONITORING

An important final step for developing NLP applications is monitoring. This not only includes monitoring the infrastructure such as server CPU, memory, and request latency, but also higher-level ML statistics such as the distributions of the input and the predicted labels. Some of the important questions to ask at this stage are:

- What do the input instances look like?
- Are they what you expected when you built your model?
- What do the predicted labels look like?
- Does the predicted label distribution match the one in the training data?

The purpose of the monitoring is to check that the model you built is behaving as intended. If the incoming text or data instances or the predicted labels do not match your expectation, you may have an out-of-domain problem, meaning that the domain of the natural language data you are receiving is different

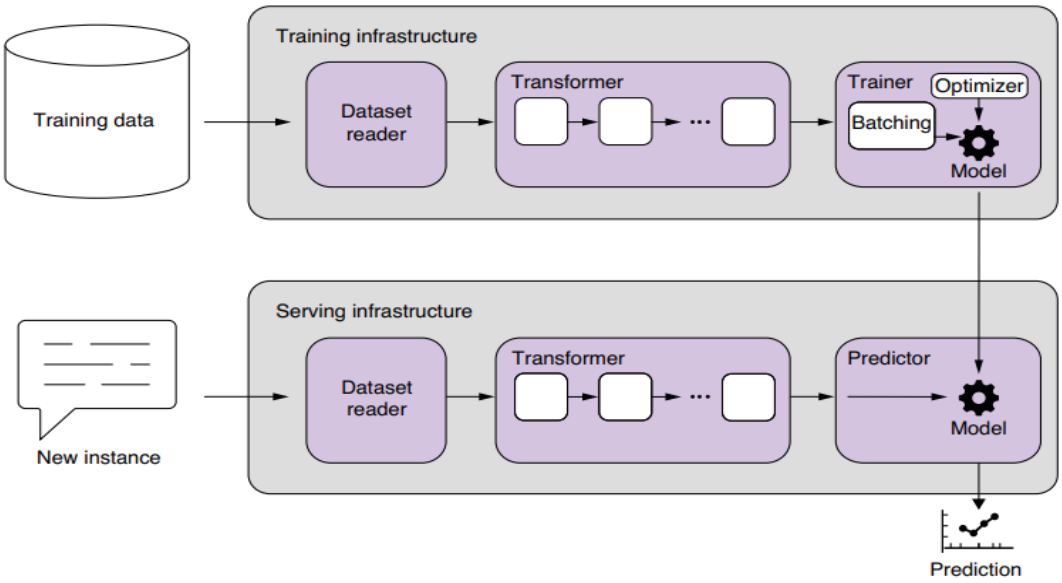
Structure of NLP applications

The structures of modern, machine learning–based NLP applications are becoming surprisingly similar for two main reasons—

1. one is that most modern NLP applications rely on machine learning to some degree, and they should follow best practices for machine learning applications.
2. The other is that, due to the advent of neural network models, a number of NLP tasks, including text classification, machine translation, dialog systems, and speech recognition, can now be trained end-to-end.

Figure below illustrates the typical structure of a modern NLP application.

There are two main infrastructures: the training and the serving infrastructure.



The training infrastructure is usually offline and serves the purpose of training the machine learning model necessary for the application. It takes the training data, converts it to some data structure that can be handled by the pipeline, and further processes it by transforming the data and extracting the features. This part varies greatly from task to task. Finally if the model is a neural network, data instances are batched and fed to the model, which is optimized to minimize the loss. The trained model is usually serialized and stored to be passed to the serving infrastructure.

The serving infrastructure's job is to, given a new instance, produce the prediction, such as classes, tags, or translations. The first part of this infrastructure, which reads the instance and transforms it into some numbers, is similar to the one for training. In fact, you must keep the dataset reader and the transformer identical. Otherwise, discrepancies will arise in the way those two process the data, also known as training-serving skew. After the instance is processed, it's fed to the pretrained model to produce the prediction.

- **Introducing sentiment Analysis**
- **working with NLP datasets.**

What is Sentiment Analysis?

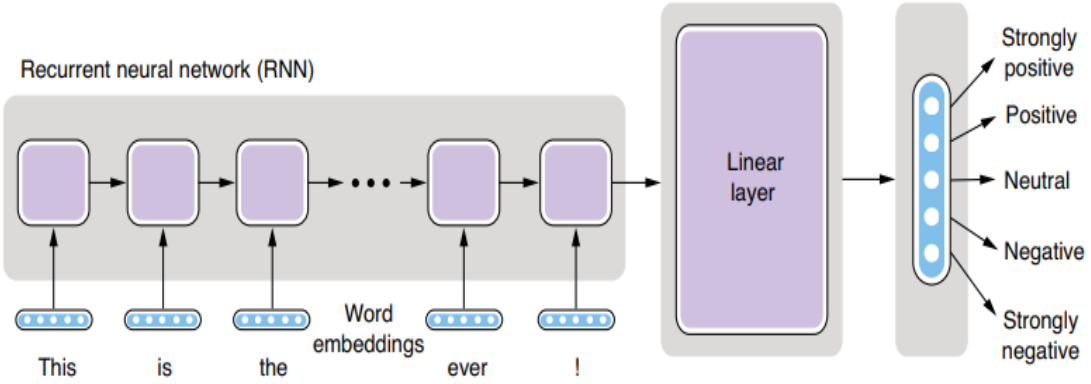
Sentiment analysis in natural language processing (NLP) is the process of determining the sentiment or emotion expressed in a piece of text. It involves using computational methods to analyze and classify the sentiment of text as positive, negative, or neutral. Sentiment analysis is widely used in various applications, including social media monitoring, customer feedback analysis, and market research.In machine learning, classification means categorizing something into a set of predefined, discrete categories.

- One of the most basic tasks in sentiment analysis is the classification of polarity, that is, to classify whether the expressed opinion is positive, negative, or neutral.
- Classification of polarity is one type of sentence classification task.
- Another type of sentence classification task is spam filtering, where each sentence is categorized into two classes—spam or not spam. It's called binary classification if there are only two classes. If there are more than two classes (the five-star classification system mentioned earlier, for example), it's called multiclass classification.

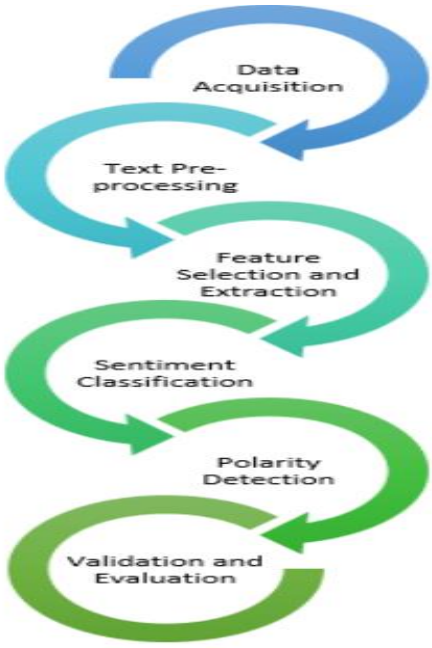
In contrast, when the prediction is a continuous value instead of discrete categories, it's called regression.

- predict the price of a house based on its properties
- predict stock prices based on the information
- collected from news articles and social media posts

But most linguistic units such as characters, words, and part-of-speech tags are discrete. For this reason, most uses of machine learning in NLP are classification, not regression.



Steps Involved In Sentiment Analysis Classification



Applications

Applications of sentiment analysis in NLP are diverse and include:

1. **Social Media Monitoring:** Analyzing the sentiment of social media posts to understand public opinion or customer feedback.

2. **Customer Feedback Analysis:** Analyzing reviews, survey responses, and customer support interactions to gauge customer satisfaction.
3. **Brand Monitoring:** Monitoring mentions of a brand online to understand public perception and sentiment.
4. **Market Research:** Analyzing sentiment in market reports, news articles, and other sources to understand market trends and consumer behavior.
5. **Political Analysis:** Analyzing sentiment in political speeches, news articles, and social media to understand public opinion and political trends.

Challenges

1. **Ambiguity:** Sentiment can be expressed in complex ways, making it challenging to accurately interpret.
2. **Context:** Understanding the context of text is crucial for accurate sentiment analysis, as the same words can have different meanings in different contexts.
3. **Sarcasm and Irony:** Sentiment analysis algorithms can struggle to detect sarcasm and irony, which can lead to misinterpretation.
4. **Language Variations:** Sentiment analysis models trained on one language or dialect may not perform well on text written in a different language or dialect.
5. **Data Quality:** The quality of the training data used to train sentiment analysis models can significantly impact their performance.

What is a dataset?

A dataset simply means a collection of data. It consists of pieces of data that follow the same format.

In NLP, records in a dataset are usually some type of linguistic units, such as words, sentences, or documents. A dataset of natural language texts is called a corpus (plural: corpora).

As an example, let's think of a (hypothetical) dataset for spam filtering. Each record in this dataset is a pair of a piece of text and a label, where the text is a sentence or a paragraph (e.g., from an email) and the label specifies whether the text is spam. Both the text and the label are the fields of a record.

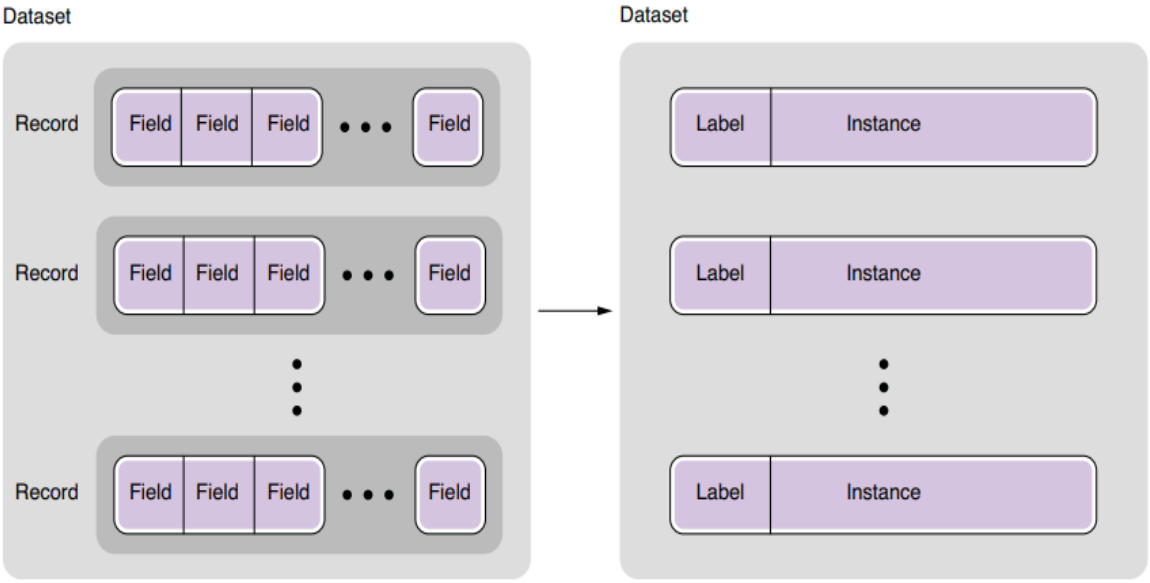
Some NLP datasets and corpora have more complex structures. For example, a dataset may contain a collection of sentences, where each sentence is annotated with detailed linguistic information, such as part-of-speech tags, parse trees, dependency structures, and semantic roles.

If a dataset contains a collection of sentences annotated with their parse trees, the dataset is called a treebank.

The most famous example of this is Penn Treebank (PTB) (http://realworldnlpbook.com/ch2.html#ptb), which has been serving as the defacto standard dataset for training and evaluating NLP tasks such as part-of-speech (POS) tagging and parsing.

- A closely related term to a record is an instance.
- In machine learning, an instance is a basic unit for which the prediction is made. For example, in the spam-filtering task mentioned earlier, an instance is one piece of text.
- An instance is usually created from a record in a dataset, as is the case for the spam-filtering task.
- Finally, a label is a piece of information attached to some linguistic unit in a dataset.
- A spam-filtering dataset has labels that correspond to whether each text is a spam.

- A treebank may have one label per word for its part of speech.



Using Word Embeddings

What are word embeddings?

Word embeddings are one of the most important concepts in modern NLP. Word embeddings are a type of representation for words in a continuous vector space where the positioning of words captures semantic relationships between them.

In simpler terms, word embeddings are numerical representations of words that allow computers to understand their meanings and relationships with other words. Word embeddings in natural language processing (NLP) are numerical representations of words that capture semantic relationships between words based on their usage in text.

Word embeddings are typically learned from large text corpora using neural network models, such as Word2Vec, GloVe, or FastText. These models map words to high-dimensional vectors in such a way that words with similar meanings are represented by vectors that are close to each other in the vector space. Word embeddings are useful in NLP tasks such as language modeling, sentiment analysis, and machine translation, as they allow models to capture the meaning of words and the relationships between them.

In the eyes of computers, “cat” is no closer to “dog” than it is to “pizza.”

One way to deal with discrete words programmatically is to assign indices to individual words as follows (here we simply assume that these indices are assigned alphabetically): ¡ index("cat") = 1

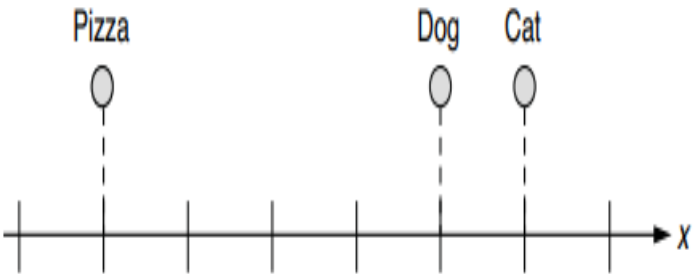
index("dog") = 2

index("pizza") = 3

The entire, finite set of words that one NLP application or task deals with is called vocabulary. Just because words are now represented by numbers doesn’t mean you can do arithmetic operations on them and conclude that “cat” is equally similar to “dog” (difference between 1 and 2), as “dog” is to “pizza” (difference between 2 and 3). Those indices are still discrete and arbitrary.

“What if we can represent them on a numerical scale?”

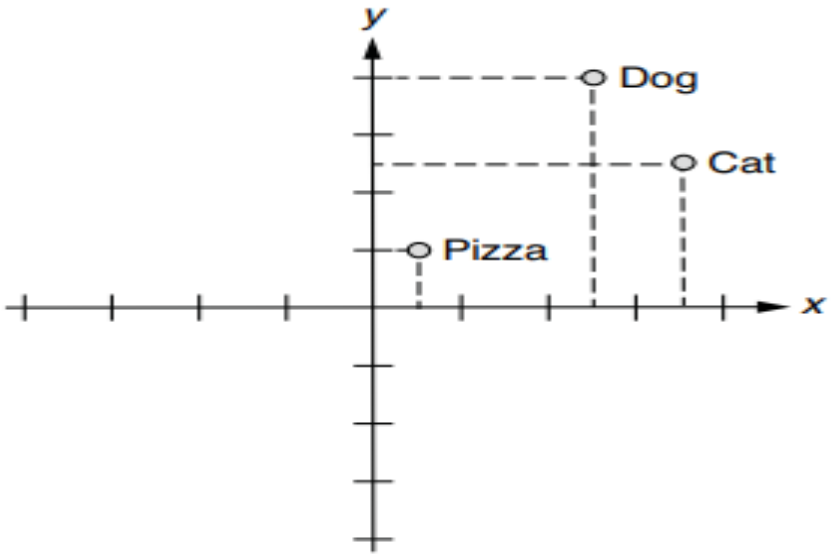
Conceptually, the numerical scale would look like the one shown in figure below



This is a step forward. Now we can represent the fact that “cat” and “dog” are more similar to each other than “pizza” is to those words.

But still, “pizza” is slightly closer to “dog” than it is to “cat.”

- What if we wanted to place it somewhere that is equally far from “cat” and “dog?”
- Maybe only one dimension is too limiting.
- How about adding another dimension to this, as shown in figure



Much better! Because computers are really good at dealing with multidimensional spaces you can simply keep doing this until you have a sufficient number of dimensions.

Let’s have three dimensions.

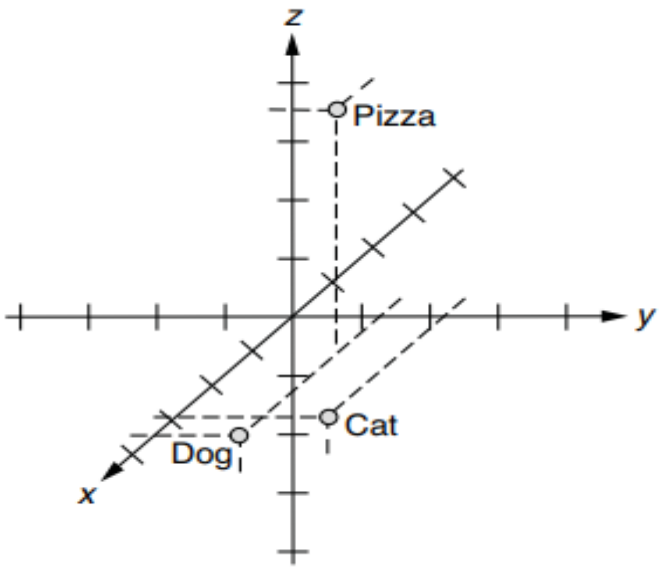
In this 3-D space, you can represent those three words as follows:

¡ vec("cat") = [0.7, 0.5, 0.1]

¡ vec("dog") = [0.8, 0.3, 0.1]

¡ vec("pizza") = [0.1, 0.2, 0.8]

Figure below illustrates this three-dimensional space



The x -axis (the first element) here represents some concept of “animal-ness” and the z-axis (the third dimension) corresponds to “food-ness.

This is essentially what word embeddings are.

Think of a multidimensional space that has as many dimensions as there are words.

Then, give to each word a vector that is filled with zeros but just one 1, as shown

¡ vec("cat") = [1, 0, 0]

¡ vec("dog") = [0, 1, 0]

¡ vec("pizza") = [0, 0, 1]

Notice that each vector has only one 1 at the position corresponding to the word’s index. These special vectors are called one-hot vectors.

Need for Word Embedding?

Word embeddings are crucial in natural language processing (NLP) for several reasons:

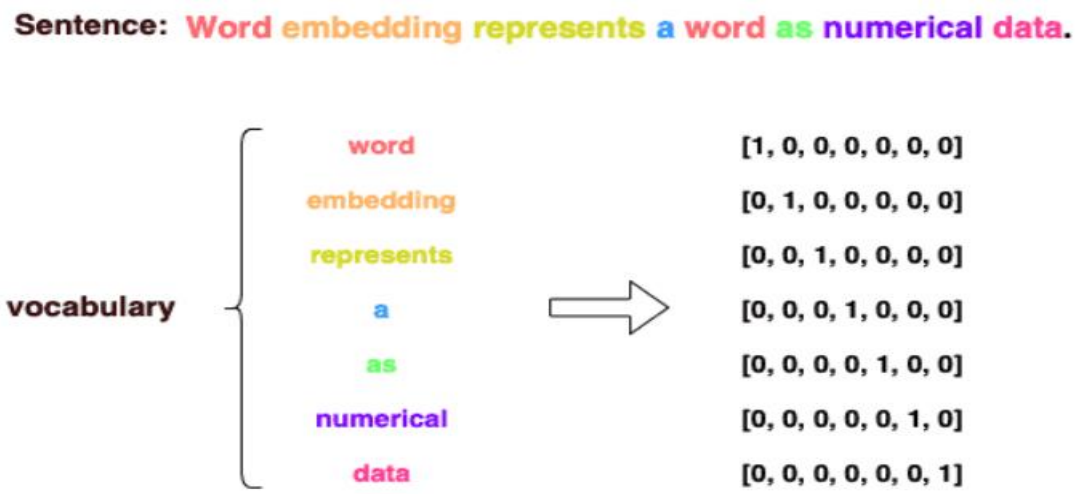
1. **Semantic Representation:** Word embeddings provide a way to represent words in a continuous vector space, where words with similar meanings are closer together. This allows NLP models to capture semantic relationships between words and understand their meanings in context.
2. **Dimensionality Reduction:** Word embeddings reduce the dimensionality of the input space, making it easier for NLP models to process and learn from text data. This can lead to more efficient and effective models.
3. **Generalization:** Word embeddings can generalize to unseen words based on their similarity to words in the training data. This is particularly useful in NLP tasks where the vocabulary is large and constantly evolving.
4. **Improved Performance:** NLP models that use word embeddings often achieve better performance compared to models that use traditional sparse representations of words, such as one-hot encoding. Word embeddings capture more nuanced relationships between words, leading to improved performance on tasks like text classification, sentiment analysis, and machine translation.
5. **Transfer Learning:** Pre-trained word embeddings can be used as a starting point for training NLP models on specific tasks. This allows models to leverage knowledge learned from large text corpora and achieve better performance with less training data.

How are Word Embeddings used?

They are used as input to machine learning models.
Take the words —-> Give their numeric representation —-> Use in training or inference.
To represent or visualize any underlying patterns of usage in the corpus that was used to train them. There are multiple ways to generate word representation.

One of the classic methods is one-hot encoding, which represents each word in a vocabulary as a binary vector. The dimensionality of the embedding is equal to the size of the vocabulary, and each element of the vector corresponds to a word in the vocabulary.

For example, in the sentence “Word embedding represents a word as numerical data.”, there are 7 unique words. Thus, the dimensionality of the word embedding is 7:



With the development of neural networks, scientists introduced more advanced methods that generate distributed word representations, such as TF-IDF, Word2vec, GloVe, etc.

Here are some examples of word embeddings and their applications:

1. **Word Similarity**: Word embeddings can capture semantic similarity between words. For instance, in a word embedding space, the vectors for "king" and "queen" are expected to be close together because they are both related to royalty.
2. **Word Analogies**: Word embeddings can solve analogies like "king is to queen as man is to woman." If you subtract the vector for "man" from "king" and add the vector for "woman," you should get a vector close to "queen" in the embedding space.
3. **Sentiment Analysis**: Word embeddings can be used in sentiment analysis tasks. By representing words in a continuous vector space, models can capture the sentiment of words and phrases more effectively.
4. **Named Entity Recognition**: In natural language processing tasks like named entity recognition, word embeddings can help identify entities more accurately by capturing contextual information about words.
5. **Machine Translation**: Word embeddings are useful in machine translation tasks. By representing words in a continuous space, it becomes easier for models to learn mappings between words in different languages.
6. **Clustering and Classification**: Word embeddings can be used in clustering tasks to group similar words together or in classification tasks to classify text documents into different categories based on the semantic meaning of words.

What are embeddings?

A word embedding is a real-valued vector representation of a word. If you find the concept of vectors intimidating, think of them as single-dimensional arrays of float numbers, like the following:

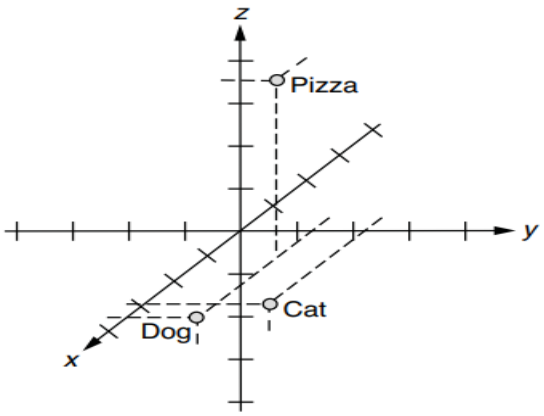
vec("cat") = [0.7, 0.5, 0.1]

vec("dog") = [0.8, 0.3, 0.1]

vec("pizza") = [0.1, 0.2, 0.8]

Because each array contains three elements, you can plot them as points in a 3-D space as in figure below.

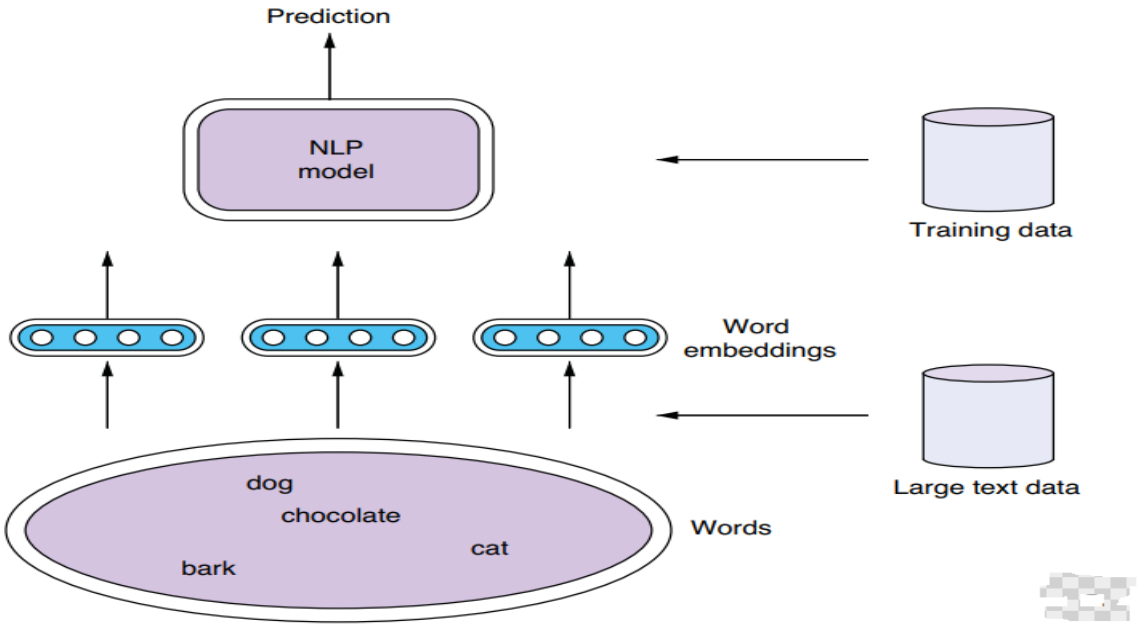
Notice that semantically-related words (“cat” and “dog”) are placed close to each other



Why are embeddings important?

- Word embeddings are not just important but essential for using neural networks to solve NLP tasks.
- Neural networks are pure mathematical computation models that can deal only with numbers. They can’t do symbolic operations, such as concatenating two strings or conjugating a verb to past tense, unless these items are all represented by numbers and arithmetic operations.
- On the other hand, almost everything in NLP, such as words and labels, is symbolic and discrete.

This is why you need to bridge these two worlds, and using embeddings is a way to do it. See figure below for an overview on how to use word embeddings for an NLP application.



Word embeddings, just like any other neural network models, can be trained, because they are simply a collection of parameters.

Embeddings are used with your NLP model in the following three scenarios:

Scenario 1: Train word embeddings and your model at the same time using the train set for your task. Scenario 2: First, train word embeddings independently using a larger text dataset. Alternatively, obtain pretrained word embeddings from somewhere else. Then initialize your model using the pretrained word embeddings, and train them and your model at the same time using the train set for your task.

Scenario 3: Same as scenario 2, except you fix word embeddings while you train your model.

Building blocks of language: Characters, words, and phrases

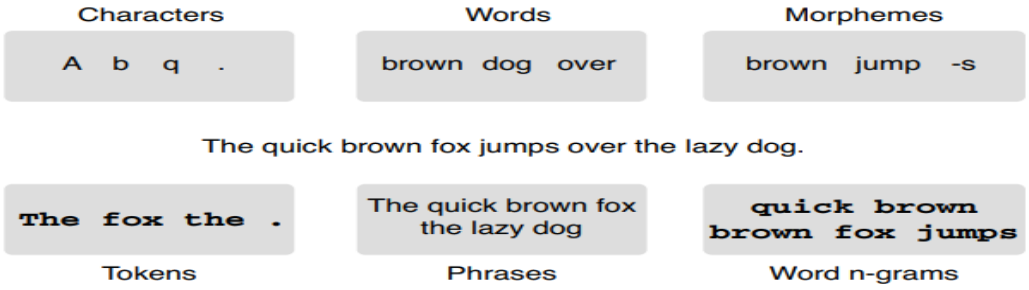
Characters

A character (also called a grapheme in linguistics) is the smallest unit of a writing system.

In written English, “a,” “b,” and “z” are characters.

Characters do not necessarily carry meaning by themselves or represent any fixed sound when spoken, although in some languages (e.g., Chinese), most do.

A typical character in many languages can be represented by a single Unicode codepoint (by string literals such as "\uXXXX" in Python), but this is not always the case. Many languages use a combination of more than one Unicode codepoint (e.g., accent marks) to represent a single character. Punctuation marks, such as “.” (period), “,” (comma), and “?” (question mark), are also characters.



Words

Words, tokens, morphemes, and phrases

A word is the smallest unit in a language that can be uttered independently and that usually carries some meaning.

In English, “apple,” “banana,” and “zebra” are words.

In most written languages that use alphabetic scripts, words are usually separated by spaces or punctuation marks. In some languages, like Chinese, Japanese, and Thai, however, words are not explicitly delimited by spaces and require a preprocessing step called word segmentation to identify words in a sentence.

Tokens

A closely related concept to a word in NLP is a token.

A token is a string of contiguous characters that play a certain role in a written language. Most words (“apple,” “banana,” “zebra”) are also tokens when written.

Punctuation marks such as the exclamation mark (“!”) are tokens but not words, because you can’t utter them in isolation.

Word and token are often used interchangeably in NLP. In fact, when you see “word” in NLP text (including this book), it often means “token,” because most NLP tasks deal only with written text that is processed in an automatic way. Tokens are the output of a process called tokenization.

Morphemes

Another closely related concept is morpheme.

A morpheme is the smallest unit of meaning in a language.

A typical word consists of one or more morphemes. For example, “apple” is a word and also a morpheme. “Apples” is a word comprised of two morphemes, “apple” and “-s,” which is used to signify the noun is plural.

English contains many other morphemes, including “-ing,” “-ly,” “-ness,” and “un-.”

The process for identifying morphemes in a word or a sentence is called morphological analysis, and it has a wide range of NLP/linguistics applications.

Phrase

A phrase is a group of words that play a certain grammatical role.

For example, “the quick brown fox” is a noun phrase (a group of words that behaves like a noun), whereas “jumps over the lazy dog” is a verb phrase.

The concept of phrase may be used somewhat liberally in NLP to simply mean any group of words.

N-grams

An n-gram is a contiguous sequence of one or more occurrences of linguistic units, such as characters and words.

For example, a word n-gram is a contiguous sequence of words, such as

- “the” (one word) ,
- “quick brown” (two words),
- “brown fox jumps” (three words).

Similarly, a character n-gram is composed of characters, such as

- “b” (one character),
- “br” (two characters),
- “row” (three characters), and so on,

which are all character n-grams made from “brown.”

An n-gram of size 1 (when n = 1) is called a unigram. N-grams of size 2 and 3 are called a bigram and a trigram, respectively.

Tokenization, stemming, and lemmatization

Steps where linguistic units are processed in a typical NLP pipeline.

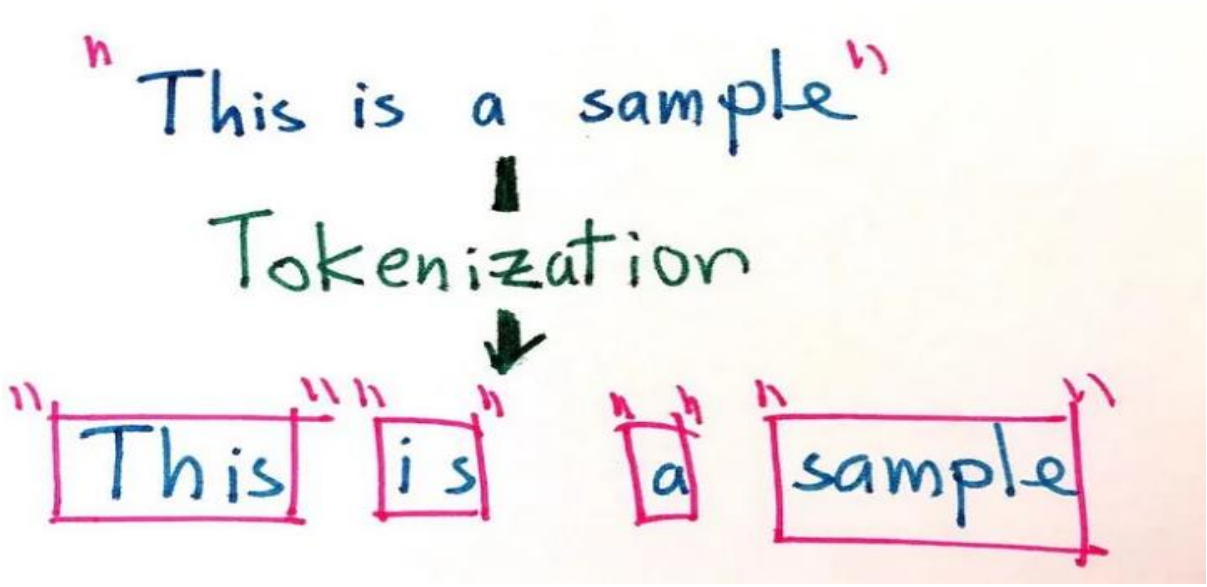
1. **Tokenization**

Tokenization is a process where the input text is split into smaller units.

There are two types of tokenization:

word and sentence tokenization.

- Word tokenization splits a sentence into tokens (rough equivalent to words and punctuation), which I mentioned earlier.
- Sentence tokenization, on the other hand, splits a piece of text that may include more than one sentence into individual sentences. If you say tokenization, it usually means word tokenization in NLP.



2. Stemming

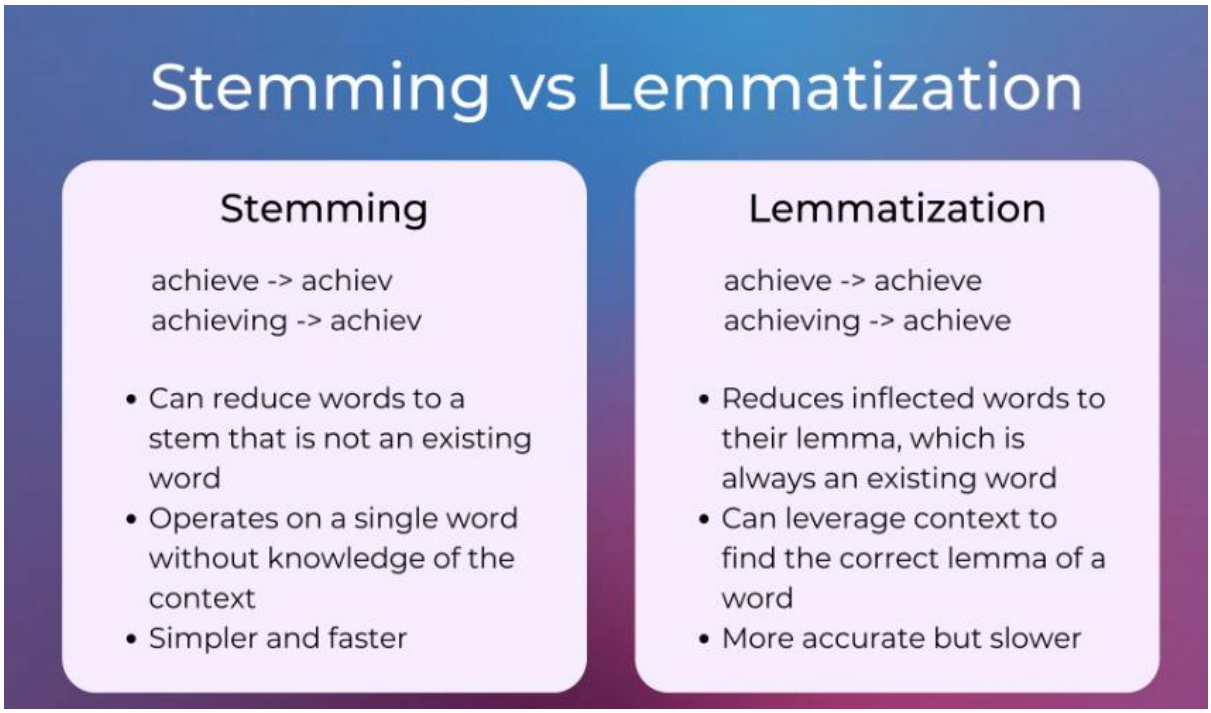
Stemming is a process for identifying word stems.

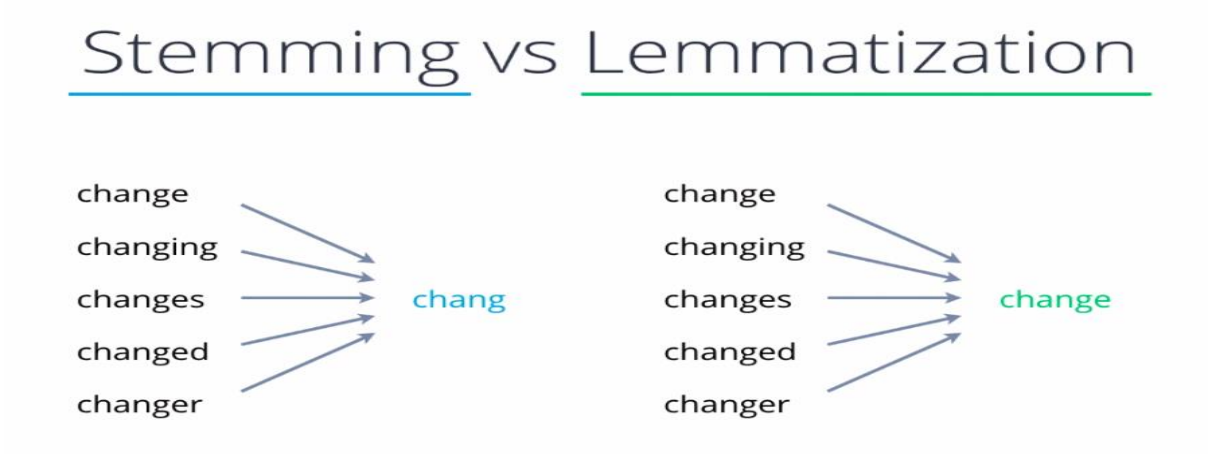
- A word stem is the main part of a word after stripping off its affixes (prefixes and suffixes).
- For example, the word stem of "apples" (plural) is "apple." The stem of "meets" (with a third-person singular s) is "meet."
- It is often a part that remains unchanged after inflection.
- Stemming—that is, normalizing words to something closer to their original forms—has great benefits in many NLP applications.
- In search, for example, you can improve the chance of retrieving relevant documents if you index documents using word stems instead of words.
- The most popular algorithm used for stemming English words is called the Porter stemming algorithm, originally written by Martin Porter. It consists of a number of rules for rewriting affixes (e.g., if a word ends with "-ization," change it to "-ize").

3.Lemmatization

- A lemma is the original form of a word that you often find as a head word in a dictionary.

- It is also the base form of the word before inflection.
- For example, the lemma of “meetings” (as a plural noun) is “meeting.” The lemma of “met” (a verb past form) is “meet.”
- Notice that it differs from stemming, which simply strips off affixes from a word and cannot deal with such irregular verbs and nouns.





Representation Of Word Embeddings

One-hot Encoding

One-hot encoding is a typical representation.

Each word is encoded using a distinct vector in this manner. The number of words is equal to the size of the vectors. As a result, if there are 1.000 words, the vectors are 1x1.000 in size. Except for a value one that distinguishes each word representation, all values in the vectors are zeros. Each word is given its vector, but this representation has its problems.

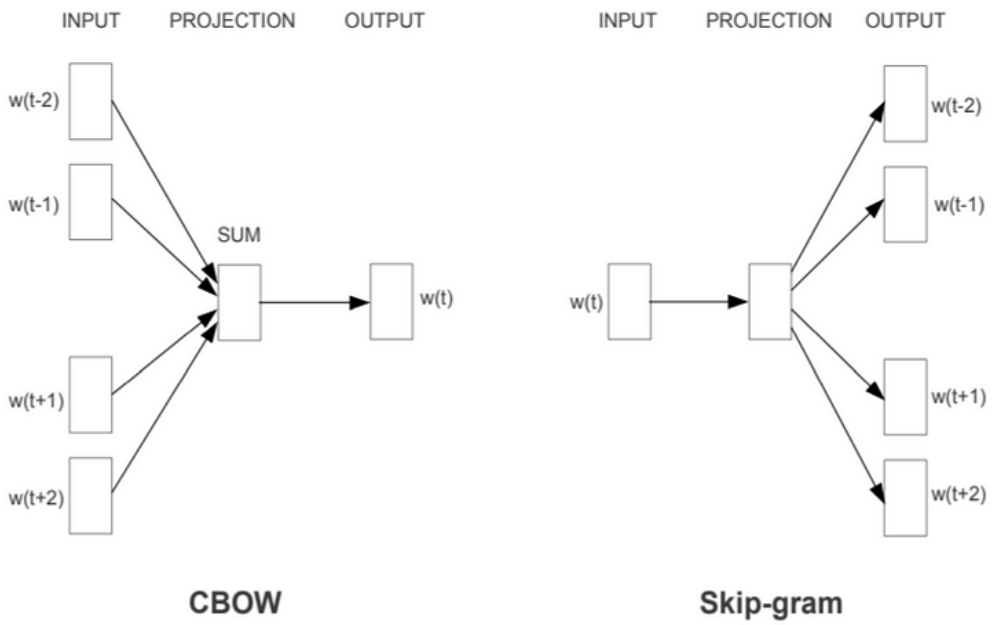
To begin with, if the vocabulary is extensive, the vectors will be enormous. Employing a model with this encoding would result in the curse of dimensionality. Adding or removing terms from the vocabulary will also affect the display of all words.

Word2Vec

- Word2Vec is a neural network model for word embeddings.
- Word2Vec generates word vectors, which are distributed numerical representations of word features - these word features could be words that indicate the context of particular words in our vocabulary.
- Through the produced vectors, word embeddings eventually assist in forming the relationship of a term with another word with a similar meaning. Context is used in these models. This means that it looks at neighboring words to learn the embedding; if a set of words is always found close to the exact words, their embeddings will be similar.

Skip-gram and continuous bag of words (CBOW)

- The Skip-Gram and Continuous Bag of Words models are two distinct architectures that Word2Vec can build word embeddings.
- To calculate word embeddings from large textual data using two popular algorithms—Skip-gram and CBOW.



In the **CBOW** model, the distributed representations of context (or surrounding words) are combined to **predict the word in the middle**. While in the **Skip-gram** model, the distributed representation of the input word is used to **predict the context**.

CBOW

The **Continuous Bag of Words (CBOW)** is a Word2Vec model that predicts a target word based on the surrounding context words. It takes a fixed-sized context window of words and tries to predict the target word in the middle of the window. The model learns by maximizing the probability of predicting the target word correctly given the context words.

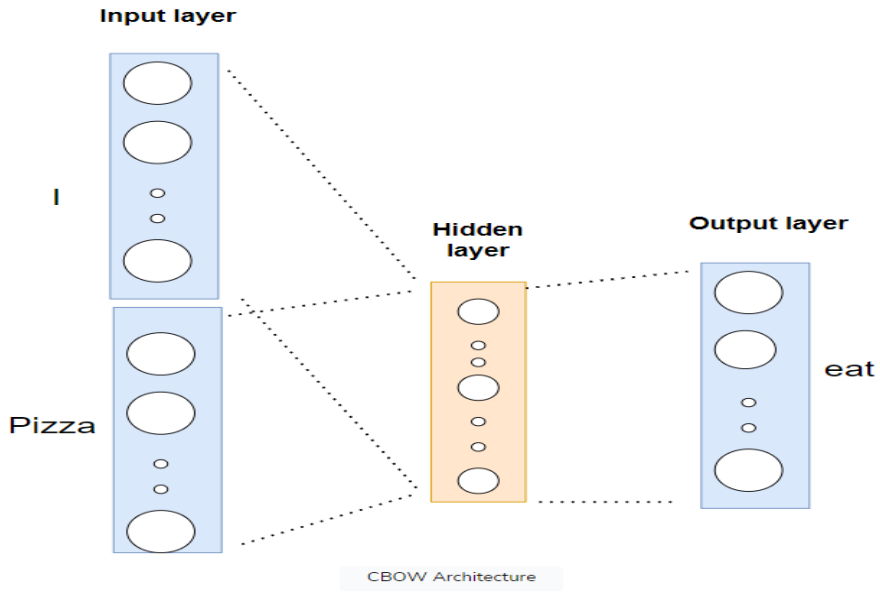
Let’s take a look at a simple example.

Example

Let's say we have the sentence, "I eat Pizza on Friday". First, we will tokenize the sentence: ["I", "eat", "pizza","on", "Friday"]. Now, let's create the training examples for this sentence for the CBOW model, considering a window size of 2.

- **Training example 1**: Input: ["I", "pizza"], Target: "eat".
- **Training example 2:** Input: ["eat", "on"], Target: "pizza".
- **Training example 3:** Input: ["pizza", "Friday"], Target:"on".

In CBOW, there are typically three main layers involved: the input layer, the hidden layer, and the output layer.



CBOW Architecture

Skip-gram

Skip-gram is another neural network architecture used in Word2Vec that predicts the context of a word, given a target word.

The input to the skip-gram model is a target word, while the output is a set of context words. The goal of the skip-gram model is to learn the probability distribution of the context words, given the target word.

During training, the skip-gram model is fed with a set of target words and their corresponding context words.

The model learns to adjust the weights of the hidden layer to maximize the probability of predicting the correct context words, given the target word.

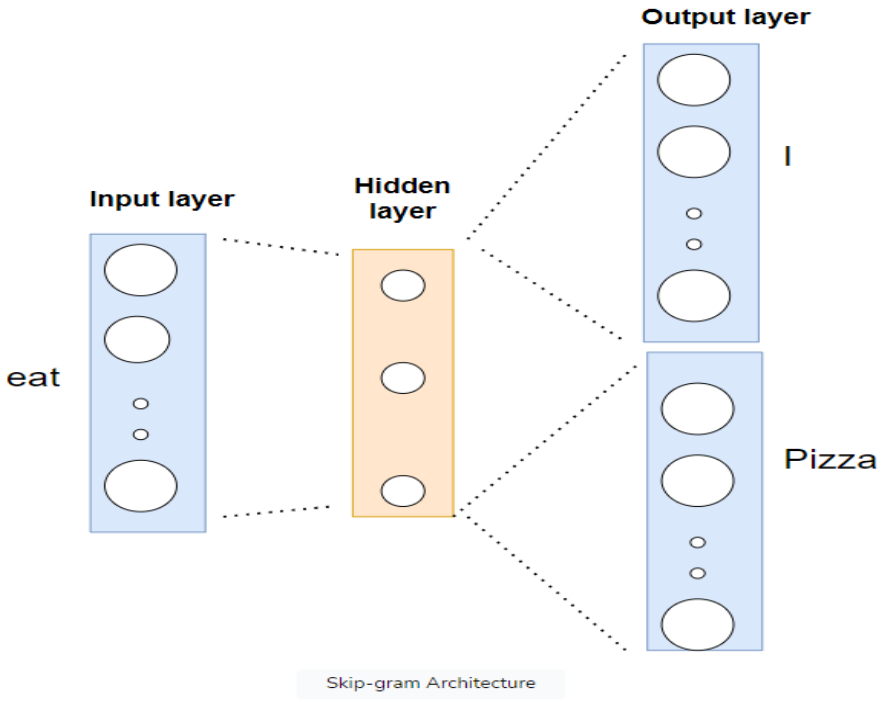
Let’s take a look at the same example discussed above.

Example

- The sentence was, "I eat Pizza on Friday". First, we will tokenize the sentence: ["I", "eat", "pizza", "on", "Friday"]. Now, let's create the training examples for this sentence

for the skip-gram model, considering a window size of 2. **Training example 1**: Input: "eat", Target: ["I", "pizza"].

- **Training example 2:** Input: "pizza", Target: ["eat", "on"].
- **Training example 3:** Input: "on", Target: ["pizza", "Friday"].
- **Training example 4:** Input: "Friday", Target: ["on"].



Skip-gram Architecture

CBOW vs Skip-gram

	Skip-gram	CBOW
Architecture	Predicts context words given a target word	Predicts a target word given context words
Context Size	Handles large context windows (e.g., 5-20 words)	Handles smaller context windows (e.g., 2-5 words)
Training	Slower training time due to multiple target predictions	Faster training time due to single target prediction
Performance	Performs well with rare words and captures word diversity	Performs well with frequent words and captures word similarity
Word Vector	Dense word vectors with high dimensionality (100-300)	Dense word vectors with high dimensionality (100-300)
Model Size	Larger model size due to more parameters	Smaller model size due to fewer parameters

Both CBOW and skip-gram provide different approaches to word embeddings, offering a trade-off between training efficiency, semantic capture, and the ability to handle different dataset characteristics. Therefore, choosing the appropriate algorithm among them requires careful consideration of the specific requirements of your task.

Both CBOW and skip-gram provide different approaches to word embeddings, offering a trade-off between training efficiency, semantic capture, and the ability to handle different dataset characteristics. Therefore, choosing the appropriate algorithm among them requires careful consideration of the specific requirements of your task.

Summary

Word embeddings are the results of learning from deep learning algorithms, which can learn characters from data through feature extraction. One implementation of word embedding is word2vec.

Word2vec has two models, namely

- Continuous Bag of Word (CBOW) and
- Skip Gram Model.

Both of these methods use the concept of a neural network that maps words to target variables, which are also words. In these techniques, "weights" are used as word vector representations. CBOW tries to predict a word on the basis of its neighbors, while Skip Gram tries to predict the neighbors of a word.

In simpler words, CBOW tends to find the probability of a word occurring in a context. So, it generalizes over all the different contexts in which a word can be used. Whereas SkipGram tends to study different contexts separately. Skip Gram needs more data to be trained contains more knowledge about the context.

- **Glove and FastText**
- **Document level Embeddings**
- **Visualizing Embeddings**

Another popular word-embedding model—GloVe, named after Global Vectors. Pretrained word embeddings generated by GloVe are probably the most widely used embeddings in NLP applications today

Skip-gram uses a prediction task where a context word ("bark") is predicted from the target word ("dog").

CBOW basically does the opposite of this.

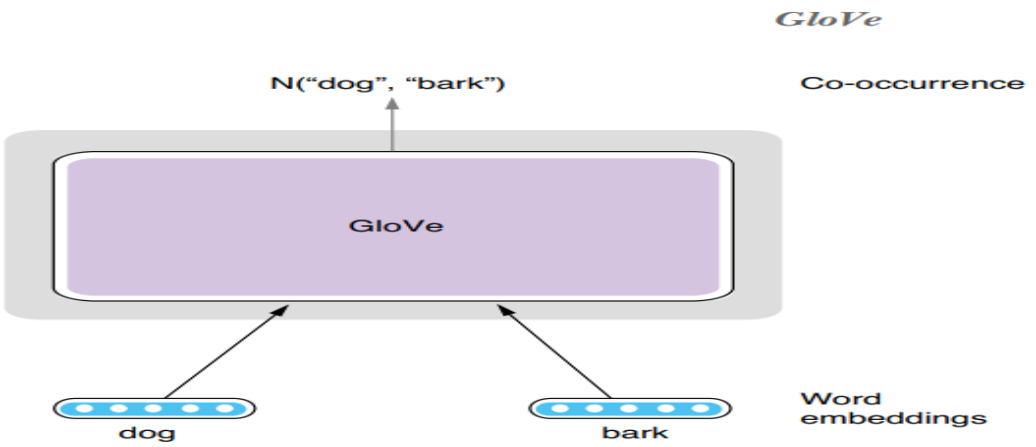
This process is repeated as many times as there are word tokens in the dataset. It basically scans the entire dataset and asks the question, "Can this word be predicted from this other word?" for every single occurrence of words in the dataset.

What if there were two or more identical sentences in the dataset? Or very similar sentences?

In that case, Skip-gram would repeat the exact same set of updates multiple times. "Can 'bark' be predicted from 'dog'?" you might ask. But chances are :you already asked that exact same question a couple of hundred sentences ago. If you know that the words "dog" and "bark" appear together in the context N times in the entire dataset, why repeat this N times? It's as if you were adding "1" N times to something else (x + 1 + 1 + 1 + ... + 1) when you could simply add N to it (x + N).

How Glove learns word embeddings

Could we somehow use the global information directly? The design of Glove is motivated by this insight. Instead of using local word co-occurrences, it uses aggregated word co-occurrence statistics in the entire dataset. Let's say "dog" and "bark" co-occur N times in the dataset. I'm not going into the details of the model, but roughly speaking, the GloVe model tries to predict this number N from the embeddings of both words. It still makes some predictions about word relations, but notice that it makes one prediction per a combination of word types, but Skip-gram does so for every combination of word tokens!



A token is an occurrence of a word in text. There may be multiple occurrences of the same word in a corpus. A type, on the other hand, is a distinctive, unique word.

For example, in the sentence **"A rose is a rose is a rose,"** there are eight tokens but only three types ("a," "rose," and "is").

Using pretrained Glove vectors

- More often, we download and use word embeddings, which are pretrained using large text corpora. This is not only quick but usually beneficial in making your NLP applications more accurate, because those pretrained word embeddings are usually trained using larger datasets and more computational power than most of us can afford.
- By using pretrained word embeddings, you can "stand on the shoulders of giants" and quickly leverage high-quality linguistic knowledge distilled from large text corpora.
- The official GloVe website (https://nlp.stanford.edu/projects/glove/) provides multiple word-embedding files trained using different datasets and vector sizes.

FastText

- Lets see how to train word embeddings using your own text data using fastText, a popular word-embedding toolkit. This is handy when your textual data is not in a general domain (e.g., medical, financial, legal, and so on) and/or is not in English

Making use of subword information

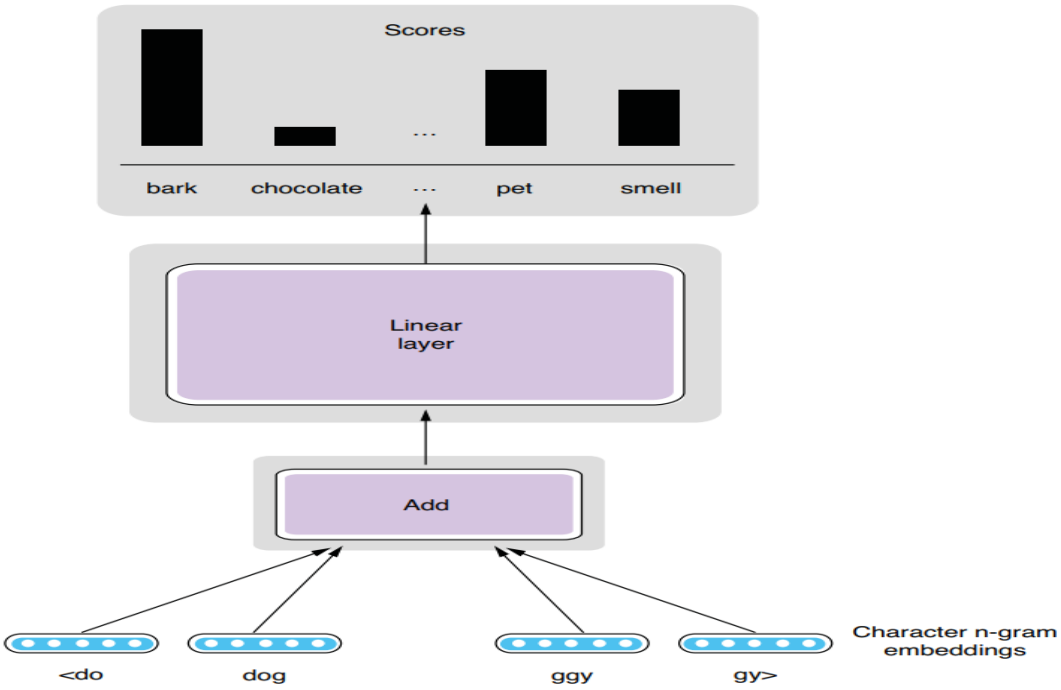
- All the word-embedding methods we've seen so far in this chapter assign a distinct word vector for each word.

- For example, word vectors for “dog” and “cat” are treated distinctly and are independently trained at the training time.
- But what if the words were, say, “dog” and “doggy?”
- Because “-y” is an English suffix that denotes some familiarity and affection (other examples include “grandma” and “granny” and “kitten” and “kitty”), these pairs of words have some semantic connection. However, word-embedding algorithms that treat words as distinct cannot make this connection.
- **This is obviously limiting.**
- In most languages, there’s a strong connection between word orthography (how you write words) and word semantics (what they mean).
- For example, words that share the same stems (e.g., “study” and “studied,” “repeat” and “repeatedly,” and “legal” and “illegal”) are often related. By treating them as separate words, word-embedding algorithms are losing a lot of information.
- How can they leverage word structures and reflect the similarities in the learned word embeddings?

FastText, an algorithm and a word-embedding library developed by Facebook, is one such model.

It uses subword information, which means information about linguistic units that are smaller than words, to train higher-quality word embeddings.

Specifically, fastText breaks words down to character n-grams and learns embeddings for them.



- Another benefit in leveraging subword information is that it can alleviate the **out-ofvocabulary (OOV) problem.**
- Many NLP applications and models assume a fixed vocabulary. For example, a typical word-embedding algorithm such as Skip-gram learns word vectors only for the words that were encountered in the train set. However, if a test set contains words that did not appear in the train set (which are called OOV words), the model would be unable to assign any vectors to them.
- For example, if you train Skipgram word embeddings from books published in the 1980s and apply them to modern social media text, how would it know what vectors to assign to "Instagram"? It won't.
- On the other hand, because **fastText uses subword information (character n-grams), it can assign word vectors to any OOV words, as long as they contain character n-grams**

Using the fastText toolkit

- Facebook provides the open source for the fastText toolkit, a library for training the word-embedding model.
- (http://realworldnlpbook.com/ch3.html #fasttext) and follow the instruction to download and compile the library.

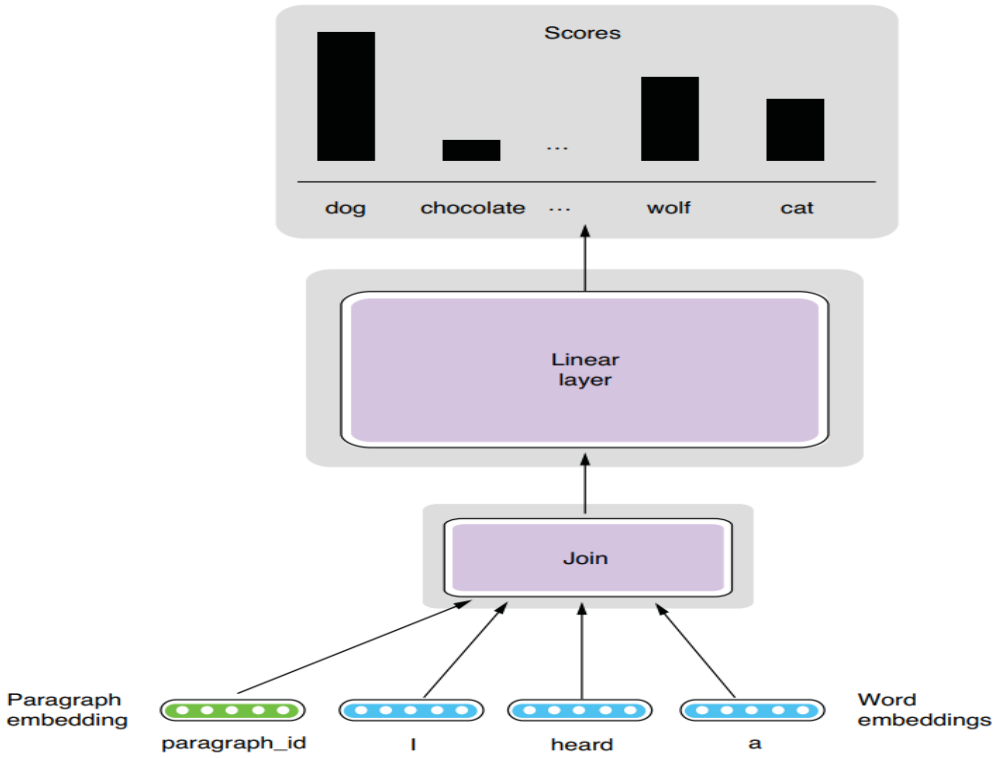
Document-level embeddings

- All the models described so far learn embeddings for individual words.
- However, if you wish to solve NLP tasks that are concerned with larger linguistic structures such as sentences and documents using word embeddings and traditional machine learning tools such as logistic regression and support vector machines (SVMs), word-level embedding methods are still limited.

How can you represent larger linguistic units such as sentences using vector representations?

How can you use word embeddings for sentiment analysis, for example?

- One way to achieve this is to simply use the average of all word vectors in a sentence. You can average vectors by taking the average of first elements, second elements, and so on and make a new vector by combining these averaged numbers.
- You can use this new vector as an input to traditional machine learning models. Although this method is simple and can be effective, it is also very limiting.
- The biggest issue is that it cannot take word order into consideration.
- For example, both sentences **"Mary loves John." and "John loves Mary."** would have exactly the same vectors if you simply averaged word vectors for each word in the sentence.
- NLP researchers have proposed models and algorithms that can specifically address this issue.
- One of the most popular is Doc2Vec, originally proposed by Le and Mikolov in 2014 (https://cs.stanford.edu/~quocle/paragraph_vector.pdf).
- This model, as its name suggests, learns vector representations for documents. In fact, "document" here simply means any variable-length piece of text that contains multiple words. Similar models are also called under many similar names such as Sentence2Vec, Paragraph2Vec, paragraph vectors (this is what the authors of the original paper used), but in essence, they all refer to the variations of the same model.



Visualizing embeddings

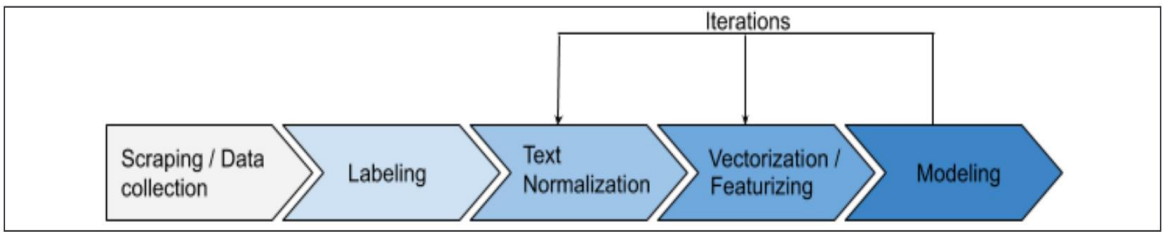
- Retrieving similar words given a query word is a great way to quickly check if word embeddings are trained correctly.
- But it gets tiring and time-consuming if you need to check a number of words to see if the word embeddings are capturing semantic relationships between words as a whole.
- word embeddings are simply N-dimensional vectors, which are also “points” in an N-dimensional space. We were able to see those points visually in a 3-D space because N was 3. But N is typically a couple of hundred in most word embeddings, and we cannot simply plot them on an N-dimensional space.
- A solution is to reduce the dimension down to something that we can see (two or three dimensions) while preserving relative distances between points.
- This technique is called dimensionality reduction. We have a number of ways to reduce dimensionality, including PCA (principal component analysis) and ICA (independent component analysis), but by far the most widely used visualization technique for word embeddings is called t-SNE (t-distributed Stochastic Neighbor Embedding, pronounced “tee-snee).

Summary

- Word embeddings are numeric representations of words, and they help convert discrete units (words and sentences) to continuous mathematical objects (vectors).
- The Skip-gram model uses a neural network with a linear layer and SoftMax to learn word embeddings as a by-product of the “fake” word-association task.
- GloVe makes use of global statistics of word co-occurrence to train word embeddings efficiently.
- Doc2Vec and fastText learn document-level embeddings and word embeddings with subword information, respectively.
- You can use t-SNE to visualize word embeddings.

A typical text processing workflow

To understand how to process text, it is important to understand the general workflow for NLP. The following diagram illustrates the basic steps:



The first two steps of the process in the preceding diagram involve collecting labeled data. A supervised model or even a semi-supervised model needs data to operate. The next step is usually normalizing and featurizing the data. Models have a hard time processing text data as is. There is a lot of hidden structure in a given text that needs to be processed and exposed. These two steps focus on that. The last step is building a model with the processed inputs.

Data collection and labelling

The first step of any Machine Learning (ML) project is to obtain a dataset. Fortunately, in the text domain, there is plenty of data to be found. A common approach is to use libraries such as scrapy or Beautiful Soup to scrape data from the web. However, data is usually unlabeled, and as such can't be used in supervised models directly. This data is quite useful though. Through the use of transfer learning, a language model can be trained using unsupervised or semi-supervised methods and can be further used with a small training dataset specific to the task at hand.

In the labeling step, textual data sourced in the data collection step is labeled with the right classes. Let's take some examples. If the task is to build a spam classifier for emails, then the previous step would involve collecting lots of emails. This labeling step would be to attach a spam or not spam label to each email. Another example could be sentiment detection on tweets. The data collection step would involve gathering a number of tweets. This step would label each tweet with a label that acts as a ground truth. A more involved example would involve collecting news articles, where the labels would be summaries of the articles. Yet another example of such a case would be an email auto-reply functionality. Like the spam case, a number of emails with their replies would need to be collected. The labels in this case would be short pieces of text that would approximate replies.

Text normalization

Text normalization is a pre-processing step aimed at improving the quality of the text and making it suitable for machines to process. Four main steps in text normalization are case normalization, tokenization and stop word removal, Parts-of-Speech (POS) tagging, and stemming. Case normalization applies to languages that use uppercase and lowercase letters. In case normalization, all letters are converted to the same case. It is quite helpful in semantic use cases. However, in other cases, this may hinder performance.

Another common normalization step removes punctuation in the text. Again, this may or may not be useful given the problem at hand. In most cases, this should give good results. However, in some cases, such as spam or grammar models, it may hinder performance. It is more likely for spam messages to use more exclamation marks or other punctuation for emphasis.

Vectorizing text

While building models for the SMS message spam detection thus far, only aggregate features based on counts or distributions of lexical or grammatical features have been considered. The actual words in the messages have not been used thus far. There are a couple of challenges in using the text content of messages. The first is that text can be of arbitrary lengths. Comparing this to image data, we know that each image has a fixed width and height. Even if the corpus of images has a mixture of sizes, images can be resized to a common size with minimal loss of information by using a variety of compression mechanisms. In NLP, this is a bigger problem compared to computer vision. A common approach to handle this is to truncate the text.

The second issue is that of the representation of words with a numerical quantity or feature. In computer vision, the smallest unit is a pixel. Each pixel has a set of numerical values indicating color or intensity. In a text, the smallest unit could be a word. Aggregating the Unicode values of the characters does not convey or embody the meaning of the word. In fact, these character codes embody no information at all about the character, such as its prevalence, whether it is a consonant or a vowel, and so on. However, averaging the pixels in a section of an image could be a reasonable approximation of that region of the image. It may represent how that region would look if seen from a large distance. A core problem then is to construct a numerical representation of words. Vectorization is the process of converting a word to a vector of numbers that embodies the information contained in the word. Depending on the vectorization technique, this vector may have additional properties that may allow comparison with other words, as will be shown in the Word vectors.

Modeling

modeling is the last part of the text processing pipeline described .

The Natural Language Toolkit (NLTK)

It is a platform used for building Python programs that work with human language in natural language processing (NLP). Almost every Natural Language Processing (NLP) task requires text to be preprocessed before training a model NLTK contains text processing libraries for tokenization, parsing, classification, stemming, tagging etc NLTK includes more than 50 corpora and lexical sources such as the Penn Treebank Corpus, Open Multilingual Wordnet, Problem Report Corpus, and Lin's Dependency Thesaurus.

Why NLTK?

Popularity: NLTK is one of the leading platforms for dealing with language data.

Simplicity: Provides easy-to-use APIs for a wide variety of text preprocessing methods

Community: It has a large and active community that supports the library and improves it

Open Source: Free and open-source available for Windows, Mac OSX, and Linux.

Text Processing steps

- Tokenization
- Lower case conversion
- Stop Words removal
- Stemming
- Lemmatization
- POS Tagging

Before preprocessing, we need to first import the NLTK library.

```
1. import nltk
```

```
# list the functions of nltk
```

```
2. dir(nltk)
```

```
3. nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data] Unzipping tokenizers/punkt.zip.
```

True

to download all packages

4. nltk.download()

NLTK Downloader

---

d) Download l) List u) Update c) Config h) Help q) Quit

Downloader> d all

1.Tokenization

It is used in natural language processing to split paragraphs and sentences into smaller units that can be more easily assigned meaning.

```
from nltk.tokenize import word_tokenize
from nltk.tokenize import sent_tokenize
'
This is my first sentence. This is my second sentence. Is this third one?
'
text1 = 'This is my first sentence. This is my second sentence. Is this third one?'
text1
sent_tokenize(text1)
```

['This is my first sentence.',

'This is my second sentence.',

'Is this third one?']

```
#Punctuation following sentences is also included by default.
PunkToken = nltk.tokenize.punkt.PunktSentenceTokenizer()
PunkToken.tokenize(text1)
```

['This is my first sentence.',

'This is my second sentence.',

'Is this third one?']

```
len(sent_tokenize(text1))
```

3

```
#Word Tokenization
word_tokens = word_tokenize(text1)
word_tokens
```

['This', 'is', 'my', 'first', 'sentence', '.', 'This', 'is', 'my', 'second', 'sentence', '.', 'Is', 'this', 'third', 'one', '?']

```
len(word_tokenize(text1))
```

17

The word_tokenize() function is a wrapper function that calls tokenize() on an instance of the TreebankWordTokenizer class. It's equivalent to the following code:

```
from nltk.tokenize import TreebankWordTokenizer

treetokenizer = TreebankWordTokenizer()

treetokenizer.tokenize(text1)
```

['This',

'is',

'my',

'first',

'sentence.',

'This',

'is',

'my',

'second',

'sentence.',

'Is',

'this',

'third',

'one',

'?']

```
len(treetokenizer.tokenize(text1))
```

15

```
# word tokenization using regular expressions

import re

words = re.findall('\w+', text1)

print(words)

print('count : ',len(words))
```

['This', 'is', 'my', 'first', 'sentence', 'This', 'is', 'my', 'second', 'sentence', 'Is', 'this', 'third', 'one']

count : 14

One of the most significant conventions of a tokenizer is to separate contractions. For example word_tokenize("I don't like it.")

['I', 'do', "n't", 'like', 'it', '.']

use treebank tokenizer

treetokenizer.tokenize("I don't like it.")

['I', 'do', "n't", 'like', 'it', '.']

An alternative word tokenizer is WordPunctTokenizer. It splits all punctuation into separate tokens:

from nltk.tokenize import WordPunctTokenizer

tokenizer = WordPunctTokenizer()

tokenizer.tokenize("Can't is a contraction.")

['Can', "'", 't', 'is', 'a', 'contraction', '.']

2.Lower casing

The lowercasing is an important text preprocessing step in which we convert the text into the same casing, preferably all in lowercase

text_lower = text1.lower()

print(text1)

print(text_lower)

This is my first sentence. This is my second sentence. Is this third one?

this is my first sentence. this is my second sentence. is this third one?

3.Filtering Stop Words

Stop words are words that you want to ignore, so you filter them out of your text when you're processing it. Very common words like 'in', 'is', and 'an' are often used as stop words since they don't add a lot of meaning to a text in and of themselves.

download the stopwords from NLTK

nltk.download('stopwords')

[nltk_data] Downloading package stopwords to /root/nltk_data...

[nltk_data] Package stopwords is already up-to-date!

True

#Import the english stop words list from NLTK

```
from nltk.corpus import stopwords # module for stop words that come with NLTK

stopwords_english = stopwords.words('english')

print('Stop words\n')

print(stopwords_english)
```

Stop words

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

```
len(stopwords_english)
```

179

```
import string # for string operations

print('\nPunctuation marks\n')

print(string.punctuation)
```

Punctuation marks

!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~

```
print(stopwords.words('spanish'))
```

['de', 'la', 'que', 'el', 'en', 'y', 'a', 'los', 'del', 'se', 'las', 'por', 'un', 'para', 'con', 'no', 'una', 'su', 'al',…]

```
print(stopwords.words('chinese'))
```

['一', '一下', '一些', '一切', '一则', '一天', '一定', '一方面', '一旦', '一时', '一来', '一样', '一次'……..]

```
from nltk.corpus import stopwords
```

```
print(stopwords.fileids())
```

['arabic', 'azerbaijani', 'basque', 'bengali', 'catalan', 'chinese', 'danish', 'dutch', 'english', 'finnish', 'french', 'german',…….]

Frequency Distrubution of words

how many times a word occured in the sentence

```
text1 = 'This is my first sentence. This is my second sentence. Is this third one ?'
fwords = tokenizer.tokenize(text1)
fwords
```

['This',

'is',

'my',

'first',

'sentence',

'.',

'This',

'is',

'my',

'second',

'sentence',

'.',

'Is',

'this',

'third',

'one',

'?']

```
# create vocab including word count
counts_a = dict()
for w in fwords:
```

```
counts_a[w] = counts_a.get(w,0)+1

print(counts_a)

print('count : ',len(counts_a))
```

{'This': 2, 'is': 2, 'my': 2, 'first': 1, 'sentence': 2, '.': 2, 'second': 1, 'Is': 1, 'this': 1, 'third': 1, 'one': 1, '?': 1}

count : 12

create vocab including word count using collections.Counter

from collections import Counter # collections library; counter: dict subclass for counting hashable objects

```
counts_b = dict()

counts_b = Counter(words)

print(counts_b)

print('count : ',len(counts_b))
```

Counter({'sunflowers': 2, 'friday': 2, 'off': 2, 'my': 1, 'beautiful': 1, 'on': 1, 'a': 1, 'sunny': 1, 'morning': 1, ':)': 1, '$': 1

count : 14

5 Stemming

Stemming is the process of converting a word to its most general form, or stem. This helps in reducing the size of our vocabulary. Stemming just removes or stems the last few characters of a word, often leading to incorrect meanings and spelling.

Consider the words: play playing

rises the different forms of happy:

happy

happiness

happier

We can see that the prefix happi is more commonly used. We cannot choose happ because it is the stem of unrelated words like happen.

NLTK has different modules for stemming and we will be using the PorterStemmer module which uses the Porter Stemming Algorithm. Let's

see how we can use it in the cell below.

'

beauti

'

```
from nltk.stem import PorterStemmer
text = 'beautifully'
stemmer = PorterStemmer()
stem_text = stemmer.stem(text)
stem_text
word_list = ['playing', 'plays', 'played']
stem_list = []
for word in word_list:
stem_word = stemmer.stem(word)
stem_list.append(stem_word)
stem_list
```

['play', 'play', 'play']

6 Lemmatization

Lemmatization is converting words into their root word using vocabulary mapping. Lemmatization is done with the help of part of speech and its meaning; hence it doesn't generate meaningless root words. But lemmatization is slower than stemming.

Lemmatization considers the context and converts the word to its meaningful base form, which is called Lemma. Sometimes, the same word

can have multiple different Lemmas. We should identify the Part of Speech (POS) tag for the word in that specific context.

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
lemmatizer.lemmatize("scarves")
```

'

scarf

'

lemmatizer.lemmatize("plays")

'

'

scarf

'

lemmatizer.lemmatize("worst")

worst

lemmatizer.lemmatize("worst", pos="a")

bad

```
from nltk.stem import WordNetLemmatizer
lm= WordNetLemmatizer()
nltk.download('wordnet')
```

[nltk_data] Downloading package wordnet to /root/nltk_data...

[nltk_data] Package wordnet is already up-to-date!

True

```
wordlist = ["playing", "played","plays", "was",'am']
print('Word list: \n', wordlist)
WordSetLem = []
for word in wordlist:
WordSetLem.append(lm.lemmatize(word))
print('Lemmatized list: \n', WordSetLem)
```

Word list:

['playing', 'played', 'plays', 'was', 'am']

Lemmatized list:

['playing', 'played', 'play', 'wa', 'am']

In the above output, you must be wondering that no actual root form has been given for any word, this is because they are given without context. we need to provide the context in which we want to lemmatize that is the parts-of-speech (POS). This is done by giving the value forpos parameter in wordnet_lemmatizer.lemmatize.

If nothing is mentioned, the default is ‘noun’.