Natural Language Processing (NLP) Unit-I

1. Natural Language Processing – Introduction

- Humans communicate through some form of language either by text or speech.
- To make interactions between computers and humans, computers need to understand natural languages used by humans.
- Natural language processing is all about making computers learn, understand, analyze, manipulate and interpret natural(human) languages.
- NLP stands for Natural Language Processing, which is a part of Computer Science, Human languages or Linguistics, and Artificial Intelligence.
- Processing of Natural Language is required when you want an intelligent system like robot to perform as per your instructions, when you want to hear decision from a dialogue based clinical expert system, etc.
- The ability of machines to interpret human language is now at the core of many applications that we use every day - chatbots, Email classification and spam filters, search engines, grammar checkers, voice assistants, and social language translators.
- The input and output of an NLP system can be Speech or Written Text.

2. Applications of NLP or Use cases of NLP

1. Sentiment analysis

- **Sentiment analysis**, also referred to as **opinion mining**, is an approach to natural language processing (NLP) that identifies the emotional tone behind a body of text.
- This is a popular way for organizations to determine and categorize opinions about a product, service or idea.
- Sentiment analysis systems help organizations gather insights into real-time customer sentiment, customer experience and brand reputation.
- Generally, these tools use text analytics to analyze online sources such as emails, blog posts, online reviews, news articles, survey responses, case studies, web chats, tweets, forums and comments.
- Sentiment analysis uses machine learning models to perform text analysis of human language. The metrics used are designed to detect whether the overall sentiment of a piece of text is positive, negative or neutral.

2. Machine Translation

- **Machine translation**, sometimes referred to by the abbreviation **MT**, is a sub-field of computational linguistics that investigates the use of software to translate text or speech from one language to another.
- On a basic level, MT performs mechanical substitution of words in one language for words in another, but that alone rarely produces a good translation because recognition of whole phrases and their closest counterparts in the target language is needed.
- Not all words in one language have equivalent words in another language, and many words have more than one meaning.

- Solving this problem with corpus statistical and neural techniques is a rapidly growing field that is leading to better translations, handling differences in linguistic typology, translation of idioms, and the isolation of anomalies.
- **Corpus:** A collection of written texts, especially the entire works of a particular author.

3. Text Extraction

- There are a number of natural language processing techniques that can be used to extract information from text or unstructured data.
- These techniques can be used to extract information such as entity names, locations, quantities, and more.
- With the help of natural language processing, computers can make sense of the vast amount of unstructured text data that is generated every day, and humans can reap the benefits of having this information readily available.
- Industries such as healthcare, finance, and e-commerce are already using natural language processing techniques to extract information and improve business processes.
- As the machine learning technology continues to develop, we will only see more and more information extraction use cases covered.

4. Text Classification

- Unstructured text is everywhere, such as emails, chat conversations, websites, and social media. Nevertheless, it's hard to extract value from this data unless it's organized in a certain way.
- Text classification also known as *text tagging* or *text categorization* is the process of categorizing text into organized groups. By using Natural Language Processing (NLP), text classifiers can automatically analyze text and then assign a set of pre-defined tags or categories based on its content.
- Text classification is becoming an increasingly important part of businesses as it allows to easily get insights from data and automate business processes.

5. Speech Recognition

- Speech recognition is an interdisciplinary subfield of computer science and computational linguistics that develops methodologies and technologies that enable the recognition and translation of spoken language into text by computers.
- It is also known as automatic speech recognition (ASR), computer speech recognition or speech to text (STT).
- It incorporates knowledge and research in the computer science, linguistics and computer engineering fields. The reverse process is speech synthesis.

Speech recognition use cases

- A wide number of industries are utilizing different applications of speech technology today, helping businesses and consumers save time and even lives. Some examples include:
- Automotive: Speech recognizers improves driver safety by enabling voice-activated navigation systems and search capabilities in car radios.
- Technology: Virtual agents are increasingly becoming integrated within our daily lives, particularly on our mobile devices. We use voice commands to access them through our smartphones, such as through Google Assistant or Apple's Siri, for tasks, such as voice search, or through our speakers, via Amazon's Alexa or Microsoft's Cortana, to play music. They'll only continue to integrate into the everyday products that we use, fueling the "Internet of Things" movement.
- Healthcare: Doctors and nurses leverage dictation applications to capture and log patient diagnoses and treatment notes.
- Sales: Speech recognition technology has a couple of applications in sales. It can help a call center transcribe thousands of phone calls between customers and agents to identify common call patterns and issues. AI chatbots can also talk to people via a webpage, answering common queries and solving basic requests without needing to wait for a contact center agent to be available. In both instances speech recognition systems help reduce time to resolution for consumer issues.

6. Chatbot

- Chatbots are computer programs that conduct automatic conversations with people. They are mainly used in customer service for information acquisition. As the name implies, these are bots designed with the purpose of chatting and are also simply referred to as "bots."

- You'll come across chatbots on business websites or messengers that give pre-scripted replies to your questions. As the entire process is automated, bots can provide quick assistance 24/7 without human intervention.

7. Email Filter

- One of the most fundamental and essential applications of NLP online is email filtering. It began with spam filters, which identified specific words or phrases that indicate a spam message. But, like early NLP adaptations, filtering has been improved.
- Gmail's email categorization is one of the more common, newer implementations of NLP. Based on the contents of emails, the algorithm determines whether they belong in one of three categories (main, social, or promotional).
- This maintains your inbox manageable for all Gmail users, with critical, relevant emails you want to see and reply to fast.

8. Search Autocorrect and Autocomplete

- When you type 2-3 letters into Google to search for anything, it displays a list of probable search keywords. Alternatively, if you search for anything with mistakes, it corrects them for you while still returning relevant results. Isn't it incredible?

- Everyone uses Google search autocorrect autocomplete on a regular basis but seldom gives it any thought. It's a fantastic illustration of how natural language processing is touching millions of people across the world, including you and me.
- Both, search autocomplete and autocorrect make it much easier to locate accurate results.

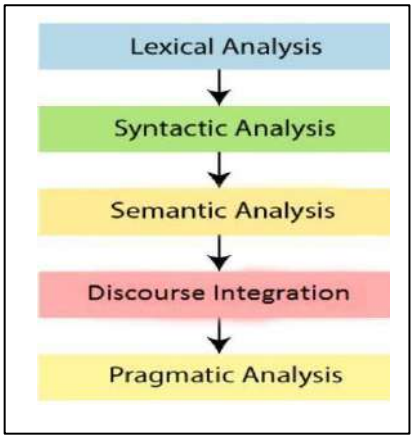
3. Components of NLP

- There are two components of NLP, Natural Language Understanding (NLU)and Natural Language Generation (NLG).
- Natural Language Understanding (NLU) which involves transforming humanlanguage into a machine-readable format.It helps the machine to understand and analyze human language by extracting the text from large data such as keywords, emotions, relations, and semantics.
- Natural Language Generation (NLG) acts as a translator that converts thecomputerized data into natural language representation.
- It mainly involves Text planning, Sentence planning, and Text realization.
- The NLU is harder than NLG.

4. Steps in NLP

There are general five steps :

- 1. Lexical Analysis
- 2. Syntactic Analysis (Parsing)
- 3. Semantic Analysis
- 4. Discourse Integration
- 5. Pragmatic Analysis



Lexical Analysis:

- The first phase of NLP is the Lexical Analysis.
- This phase scans the source code as a stream of characters and converts it into meaningful lexemes.
- It divides the whole text into paragraphs, sentences, and words.
- Lexeme: A lexeme is a basic unit of meaning. In linguistics, the abstract unit of morphological analysis that corresponds to a set of forms taken by a single word is called lexeme.
- The way in which a lexeme is used in a sentence is determined by its grammatical category.

- Lexeme can be individual word or multiword.
- For example, the word talk is an example of an individual word lexeme, which mayhave many grammatical variants like talks, talked and talking.
- Multiword lexeme can be made up of more than one orthographic word. For example,speak up, pull through, etc. are the examples of multiword lexemes.

Syntax Analysis (Parsing)

- Syntactic Analysis is used to check grammar, word arrangements, and shows therelationship among the words.
- The sentence such as "The school goes to boy" is rejected by English syntactic analyzer.

Semantic Analysis

- Semantic analysis is concerned with the meaning representation.
- It mainly focuses on the literal meaning of words, phrases, and sentences.
- The semantic analyzer disregards sentence such as "hot ice-cream".
- Another Example is "Manhattan calls out to Dave" passes a syntactic analysis because it's a grammatically correct sentence. However, it fails a semantic analysis. Because Manhattan is a place (and can't literally call out to people), the sentence's meaning doesn't make sense.

Discourse Integration

- Discourse Integration depends upon the sentences that precedes it and also invokesthe meaning of the sentences that follow it.
- For instance, if one sentence reads, "Manhattan speaks to all its people," and the following sentence reads, "It calls out to Dave," discourse integration checks the first sentence for context to understand that "It" in the latter sentence refers to Manhattan.

Pragmatic Analysis

- During this, what was said is re-interpreted on what it actually meant.
- It involves deriving those aspects of language which require real world knowledge.
- For instance, a pragmatic analysis can uncover the intended meaning of "Manhattan speaks to all its people." Methods like neural networks assess the context to understand that the sentence isn't literal, and most people won't interpret it as such. A pragmatic analysis deduces that this sentence is a metaphor for how people emotionally connect with place.

5. Finding the structure of Words

Words and Their Components

- Words are defined in most languages as the smallest linguistic units that can form acomplete utterance by themselves.
- The minimal parts of words that deliver aspects of meaning to them are called **morphemes**.

Tokens:

Suppose, for a moment, that words in English are delimited only by whitespace and punctuation (the marks, such as full stop, comma, and brackets)

- Example: Will you read the newspaper? Will you read it? I won't read it. If we confront our assumption with insights from syntax, we notice twowords here: words ***newspaper*** and ***won't***.

Being a compound word, ***newspaper*** has an interesting **derivational structure**.

In writing, *newspaper* and the associated concept is distinguished from the isolated *news* and *paper*.

For reasons of generality, linguists prefer to analyze *won't* as two syntactic words, or tokens, each of which has its independent role and can be reverted to its normalized form.

- The structure of ***won't*** could be parsed as ***will*** followed by ***not***.
- In English, this kind of tokenization and **normalization** may apply to just a limited set of cases, but in other languages, these phenomena have to be treated different way.

Lexemes

- By the term word, we often denote not just the one linguistic form in the given context but also the concept behind the form and the set of alternative forms that can express it.
- Such sets are called lexemes or lexical items, and they constitute the lexicon of a language.
- Lexemes can be divided by their behaviour into the lexical categories of verbs, nouns, adjectives, conjunctions or other parts of speech.
- The citation form of a lexeme, by which it is commonly identified, is also called its lemma.
- When we convert a word into its other forms, such as turning the **singular *mouse*** into the **plural *mice* or *mouses***, we say we **inflect** the lexeme.
- When we transform a lexeme into another one that is morphologically related, regardless of its lexical category, we say we derive the lexeme: for instance, the nouns ***receiver* and *reception*** are derived from the verb *receive*.
- Example: Did you see him? I didn't see him. I didn't see anyone
  Example presents the problem of tokenization of didn't and the investigation of the internal structure of anyone.
- The difficulty with the definition of what counts as a word need not pose a problem for the syntactic description if we understand no one as two closely connected tokens treated as one fixed element.

Morphemes

These components are usually called segments or morphs.

Morphology

Morphology is the domain of linguistics that analyses the internal structure of words.

- Morphological analysis – exploring the structure of words
- Words are built up of minimal meaningful elements called morphemes:
  played = play-ed
  cats = cat-s
  unfriendly = un-friend-ly

Two types of morphemes:
i Stems: play, cat, friend
ii Affixes: -ed, -s, un-, -ly
Two main types of affixes:
i Prefixes precede the stem: un
ii Suffixes follow the stem: -ed, -s, un-, -ly

Stemming = find the stem by stripping off affixes

play = play
replayed = re-play-ed
computerized = comput-er-ize-d

Problems in morphological processing

Inflectional morphology: inflected forms are constructed from base forms and inflectional
Affixes.
Inflection relates different forms of the same word

Lemma	Singular	Plural
Cat	cat	Cats
Mouse	mouse	mice

Derivational morphology: words are constructed from roots (or stems) and derivational
affixes:
inter+national = international
international+ize = internationalize
internationalize+ation = internationalization

- The simplest morphological process concatenates morphs one by one, as in disagree-ment-s, where agree is a free lexical morpheme and the other elements are bound grammatical morphemes contributing some partial meaning to the whole word.
- In a more complex scheme, morphs can interact with each other, and their forms may become subject to additional phonological and orthographic changes denoted as morphophonemic.
- The alternative forms of a morpheme are termed allomorphs.
- The ending *-s,* indicating plural in "cats," "dogs," the *-es* in "dishes," and the *-en* of "oxen" are all allomorphs of the plural morpheme.

Typology

- Morphological typology divides languages into groups by characterizing the prevalent morphological phenomena in those languages.
- It can consider various criteria, and during the history of linguistics, different classifications have been proposed.
- Let us outline the typology that is based on quantitative relations between words, their morphemes, and their features:

- Isolating, or analytic, languages include no or relatively few words that would comprise more than one morpheme (typical members are Chinese, Vietnamese, and Thai; analytic tendencies are also found in English).
- Synthetic languages can combine more morphemes in one word and are further divided into agglutinative and fusional languages.
- Agglutinative languages have morphemes associated with only a single function at a time (as in Korean, Japanese, Finnish, and Tamil, etc.)
- Fusional languages are defined by their feature-per-morpheme ratio higher than one (as in Arabic, Czech, Latin, Sanskrit, German, etc.).
- In accordance with the notions about word formation processes mentioned earlier, we can also find out using concatenative and nonlinear:
- Concatenative languages linking morphs and morphemes one after another.
- Nonlinear languages allowing structural components to merge nonsequentially to apply tonal morphemes or change the consonantal or vocalic templates of words.

Morphological Typology

- Morphological typology is a way of classifying the languages of the world that groups languages according to their common morphological structures.
- The field organizes languages on the basis of how those languages form words by combining morphemes.
- The morphological typology classifies languages into two broad classes like synthetic languages and analytical languages.
- The synthetic class is then further sub classified as either agglutinative languages or fusional languages.
- Analytic languages contain very little inflection, instead relying on features like word order and auxiliary words to convey meaning.
- Synthetic languages, ones that are not analytic, are divided into two categories: agglutinative and fusional languages.
- Agglutinative languages rely primarily on discrete particles(prefixes, suffixes, and infixes) for inflection, ex: inter+national = international, international+ize = internationalize.
- While fusional languages "fuse" inflectional categories together, often allowing one word ending to contain several categories, such that the original root can be difficult to extract (anybody, newspaper).

6. Natural Language Processing With Python's NLTK Package

- **NLTK**, or Natural Language Toolkit, is a Python package that you can use for NLP.
- A lot of the data that you could be analyzing is unstructured data and contains human-readable text.
- Before you can analyze that data programmatically, you first need to preprocess it.
- Now we are going to see kinds of **text preprocessing** tasks you can do with NLTK so that you'll be ready to apply them in future projects.

1. **Tokenizing**

- By **tokenizing**, you can conveniently split up text by word or by sentence.
- This will allow you to work with smaller pieces of text that are still relatively coherent and meaningful even outside of the context of the rest of the text.
- It's your first step in turning unstructured data into structured data, which is easier to analyze.
- When you're analyzing text, you'll be tokenizing by word and tokenizing by sentence.

Tokenizing by word

- Words are like the atoms of natural language. They're the smallest unit of meaning that still makes sense on its own.
- Tokenizing your text by word allows you to identify words that come up particularly often.
- For example, if you were analyzing a group of job ads, then you might find that the word "Python" comes up often.
- That could suggest high demand for Python knowledge, but you'd need to look deeper to know more.

Tokenizing by sentence

- When you tokenize by sentence, you can analyze how those words relate to one another and see more context.
- Are there a lot of negative words around the word "Python" because the hiring manager doesn't like Python?
- Are there more terms from the domain of herpetology than the domain of software development, suggesting that you may be dealing with an entirely different kind of python than you were expecting?

Python Program for Tokenizing by Sentence

```
from nltk.tokenize import sent_tokenize, word_tokenize

example_string = """

    Muad'Dib learned rapidly because his first training was in how to
    learn. And the first lesson of all was the basic trust that he could
    learn.It's shocking to find how many people do not believe
    they can learn,and how many more believe learning to be
    difficult."""

sent_tokenize(example_string)
```

Output

```
["\n Muad'Dib learned rapidly because his first training was in how to learn.",
```

'And the first lesson of all was the basic trust that he could learn.',

"It's shocking to find how many people do not believe they can learn,\n and how many more believe learning to be difficult."]

Note:

import nltk

nltk.download('punkt')

Python Program for Tokenizing by Word

from nltk.tokenize import sent_tokenize, word_tokenize

example_string = """

Muad'Dib learned rapidly because his first training was in how to learn. And the first lesson of all was the basic trust that he could learn.It's shocking to find how many people do not believe they can learn,and how many more believe learning to be difficult."""

word_tokenize(example_string)

Output:

["Muad'Dib", 'learned', 'rapidly', 'because', 'his', 'first', 'training', 'was', 'in', 'how', 'to', 'learn', '.', 'And', 'the', 'first', 'lesson', 'of', 'all', 'was', 'the', 'basic', 'trust', 'that', 'he', 'could', 'learn', '.', 'It', "'s", 'shocking', 'to', 'find', 'how', 'many', 'people', 'do', 'not', 'believe', 'they', 'can', 'learn', ',', 'and', 'how', 'many', 'more', 'believe', 'learning', 'to', 'be', 'difficult', '.']

2. 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.
- Note: nltk.download("stopwords")

Python program to eliminate stopwords

from nltk.corpus import stopwords

from nltk.tokenize import word_tokenize

worf_quote = "Sir, I protest. I am not a merry man!"

words_in_quote = word_tokenize(worf_quote)

print(words_in_quote)

stop_words = set(stopwords.words("english"))

filtered_list = []

```
for word in words_in_quote:
    if word.casefold() not in stop_words:
        filtered_list.append(word)
print(filtered_list)
```

Output:

- ['Sir', ',', 'I', 'protest', '.', 'I', 'am', 'not', 'a', 'merry', 'man', '!']
- ['Sir', ',', 'protest', '.', 'merry', 'man', '!']
- 'I' is pronoun and it is context word
- **Content words** give you information about the topics covered in the text or the sentiment that the author has about those topics.
- **Context words** give you information about writing style. You can observe patterns in how authors use context words in order to quantify their writing style.
- Once you've quantified their writing style, you can analyze a text written by an unknown author to see how closely it follows a particular writing style so you can try to identify who the author is.

3. Stemming

- **Stemming** is a text processing task in which you reduce words to their root, which is the core part of a word.
- For example, the words "helping" and "helper" share the root "help."
- Stemming allows you to zero in on the basic meaning of a word rather than all the details of how it's being used.
- NLTK has more than one stemmer, but we'll be using the Porter stemmer.

Python program for Stemming

```
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize
stemmer = PorterStemmer()
string_for_stemming = "The crew of the USS Discovery discovered many discoveries. Discovering is what explorers do."
words = word_tokenize(string_for_stemming)
print(words)
stemmed_words = [stemmer.stem(word) for word in words]
print(stemmed_words)
```

Output

- ['The', 'crew', 'of', 'the', 'USS', 'Discovery', 'discovered', 'many', 'discoveries', '.', 'Discovering', 'is', 'what', 'explorers', 'do', '.']
- ['the', 'crew', 'of', 'the', 'uss', 'discoveri', 'discov', 'mani', 'discoveri', '.', 'discov', 'is', 'what', 'explor', 'do', '.']

Original word	Stemmed version
'Discovery'	'discoveri'
'discovered'	'discov'
'discoveries'	'discoveri'
'Discovering'	'discov'

4. Tagging Parts of Speech

Part of speech is a grammatical term that deals with the roles words play when you use them together in sentences. Tagging parts of speech, or **POS tagging**, is the task of labeling the words in your text according to their part of speech.

Part of speech	**Role**	**Examples**
Noun	Is a person, place, or thing	mountain, bagel, Poland
Pronoun	Replaces a noun	you, she, we
Adjective	Gives information about what a noun is like	efficient, windy, colorful
Verb	Is an action or a state of being	learn, is, go
Adverb	Gives information about a verb, an adjective, or another adverb	efficiently, always, very
Preposition	Gives information about how a noun or pronoun is connected to another word	from, about, at
Conjunction	Connects two other words or phrases	so, because, and

Interjection	Is an exclamation	yay, ow, wow

- Some sources also include the category **articles** (like “a” or “the”) in the list of parts of speech, but other sources consider them to be adjectives. NLTK uses the word **determiner** to refer to articles.

Python program for Tagging Parts of Speech

```
import nltk

nltk.download('punkt')

nltk.download('averaged_perceptron_tagger')

from nltk.tokenize import word_tokenize

sagan_quote = """

  If you wish to make an apple pie from scratch,

  you must first invent the universe."""

words_in_sagan_quote = word_tokenize(sagan_quote)

nltk.pos_tag(words_in_sagan_quote)
```

Output:

- [('If', 'IN'), ('you', 'PRP'), ('wish', 'VBP'), ('to', 'TO'), ('make', 'VB'), ('an', 'DT'), ('apple', 'NN'), ('pie', 'NN'), ('from', 'IN'), ('scratch', 'NN'), (',', ','), ('you', 'PRP'), ('must', 'MD'), ('first', 'VB'), ('invent', 'VB'), ('the', 'DT'), ('universe', 'NN'), ('.', '.')]

POS Tag information

- nltk uses The Penn Treebank's POS tags

```
nltk.download('tagsets')

nltk.help.upenn_tagset()
```

5. Lemmatizing

- Like stemming, **lemmatizing** reduces words to their core meaning, but it will give you a complete English word that makes sense on its own instead of just a fragment of a word like 'discoveri'.
- A **lemma** is a word that represents a whole group of words, and that group of words is called a **lexeme**.
- For example, if you were to look up the word “blending” in a dictionary, then you’d need to look at the entry for “blend,” but you would find “blending” listed in that entry.
- In this example, “blend” is the **lemma**, and “blending” is part of the **lexeme**. So when you lemmatize a word, you are reducing it to its lemma.

5. Python Program for Lemmatization

```
import nltk
nltk.download('punkt')
nltk.download('wordnet')
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
lemmatizer = WordNetLemmatizer()
string_for_lemmatizing = "The friends of DeSoto love scarves."
words = word_tokenize(string_for_lemmatizing)
lemmatized_words = [lemmatizer.lemmatize(word) for word in words]
print(lemmatized_words)
```

Output:

- lemmatizer.lemmatize("worst")
  o/p: 'worst'
- lemmatizer.lemmatize("worst", pos="a")
  o/p: 'bad'

6. Chunking

- **chunking** allows you to identify **phrases**.
- A **phrase** is a word or group of words that works as a single unit to perform a grammatical function. **Noun phrases** are built around a noun.
- Here are some examples:
- "A planet"
- "A tilting planet"
- "A swiftly tilting planet"
- Chunking makes use of POS tags to group words and apply chunk tags to those groups. Chunks don't overlap, so one instance of a word can be in only one chunk at a time.
- After getting a list of tuples of all the words in the quote, along with their POS tag. In order to chunk, you first need to define a chunk grammar.
- **Note:** A **chunk grammar** is a combination of rules on how sentences should be chunked. It often uses regular expressions, or **regexes**.
- Create a chunk grammar with one regular expression rule:
- grammar = "NP: {<DT>?<JJ>*<NN>}"
- Create a chunk parser with this grammar:

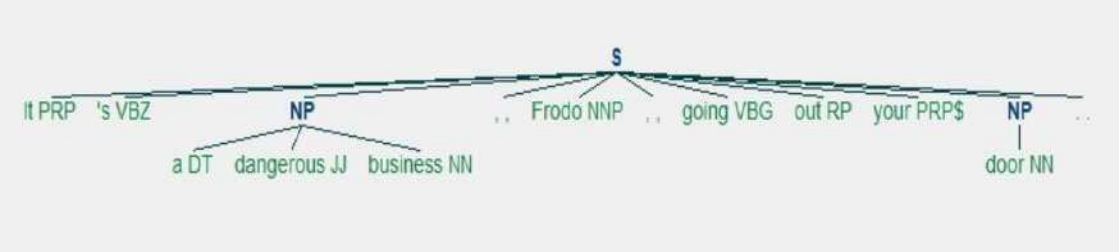
Python program for chuncking

```
import nltk
nltk.download('puckt')
from nltk.tokenize import word_tokenize
quote = "It's a dangerous business, Frodo, going out your door."
words_quote = word_tokenize(quote)
print(words_quote)
nltk.download("averaged_perceptron_tagger")
tags = nltk.pos_tag(words_quote)
print(tags)
#Regular expression for Noun Phrase
grammar = "NP: {<DT>?<JJ>*<NN>}"
#Create a chunk parser with this grammar:
chunk_parser = nltk.RegexpParser(grammar)
tree = chunk_parser.parse(tags)
print(tree)
```

Output:

- ['It', "'s", 'a', 'dangerous', 'business', ',', 'Frodo', ',', 'going', 'out', 'your', 'door', '.']
- [('It', 'PRP'), ("'s", 'VBZ'), ('a', 'DT'), ('dangerous', 'JJ'), ('business', 'NN'), (',', ','), ('Frodo', 'NNP'), (',', ','), ('going', 'VBG'), ('out', 'RP'), ('your', 'PRP$'), ('door', 'NN'), ('.', '.')]
- (S
- It/PRP
- 's/VBZ
- (NP a/DT dangerous/JJ business/NN)
- ,/, Frodo/NNP
- ,/, going/VBG
- out/RP
- your/PRP$
- (NP door/NN)
- ./.)

Tree Representation



7. Chinking

- Chinking is used together with chunking, but while chunking is used to include a pattern, **chinking** is used to exclude a pattern.

Python program to perform chinking

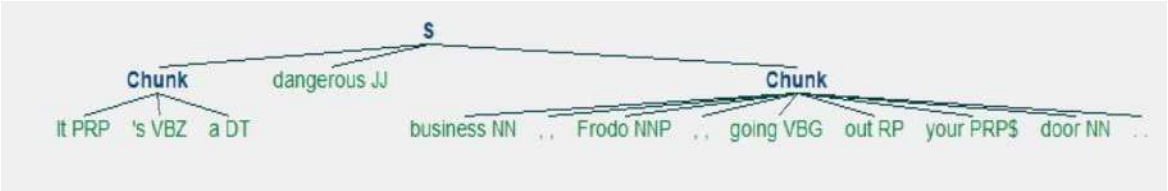
```
import nltk
nltk.download('puckt')
from nltk.tokenize import word_tokenize
quote = "It's a dangerous business, Frodo, going out your door."
words_quote = word_tokenize(quote)
print(words_quote)
nltk.download("averaged_perceptron_tagger")
tags = nltk.pos_tag(words_quote)
print(tags)
#Regular expression
grammar = """
    Chunk: {<.*>+}
    }<JJ>{""“
chunk_parser = nltk.RegexpParser(grammar)
tree = chunk_parser.parse(tags)
print(tree)
```

Output:

- ['It', "'s", 'a', 'dangerous', 'business', ',', 'Frodo', ',', 'going', 'out', 'your', 'door', '.']
- [('It', 'PRP'), ("'s", 'VBZ'), ('a', 'DT'), ('dangerous', 'JJ'), ('business', 'NN'), (',', ','), ('Frodo', 'NNP'), (',', ','), ('going', 'VBG'), ('out', 'RP'), ('your', 'PRP$'), ('door', 'NN'), ('.', '.')]

- (S
- (Chunk It/PRP 's/VBZ a/DT)
- dangerous/JJ
- (Chunk business/NN ,/, Frodo/NNP ,/, going/VBG out/RP your/PRP$ door/NN ./.))

Tree Representation



8. **Using Named Entity Recognition (NER)**

Some Examples of Named Entity Recognition (NER)

NE type	Examples
ORGANIZATION	Georgia-Pacific Corp., WHO
PERSON	Eddy Bonte, President Obama
LOCATION	Murray River, Mount Everest
DATE	June, 2008-06-29
TIME	two fifty a m, 1:30 p.m.
MONEY	175 million Canadian dollars, GBP 10.40
PERCENT	twenty pct, 18.75 %
FACILITY	Washington Monument, Stonehenge
GPE	South East Asia, Midlothian

Python Program to Name Entity Recognition

```
import nltk
nltk.download('punkt')
from nltk.tokenize import word_tokenize
quote = "It's a dangerous business, Frodo, going out your door."
words_quote = word_tokenize(quote)
print(words_quote)
nltk.download("averaged_perceptron_tagger")
tags = nltk.pos_tag(words_quote)
nltk.download("maxent_ne_chunker")
nltk.download("words")
tree = nltk.ne_chunk(tags)
print(tree)
```

Output

```
['It', "'s", 'a', 'dangerous', 'business', ',', 'Frodo', ',', 'going', 'out', 'your', 'door', '.']
```

(S
It/PRP
's/VBZ
a/DT
dangerous/JJ
business/NN
,/,
(PERSON Frodo/NNP)
,/,
going/VBG
out/RP
your/PRP$
door/NN
./.)

Note: If we use this code it simply specifies that it is a Named Entity with out giving the specification.

- tree = nltk.ne_chunk(tags, binary=True)
- print(tree)

Output

```
(S
  It/PRP
  's/VBZ
  a/DT
  dangerous/JJ
  business/NN
  ,/,
  (NE Frodo/NNP)
  ,/,
  going/VBG
  out/RP
  your/PRP$
  door/NN
  ./.)
```

.

Natural Language Processing
Unit-II

2.1 Parsing Natural Language
2.2 Treebanks: A Data-Driven Approach to Syntax 2.3Representation of Syntactic Structure
2.3 Parsing Algorithms
2.4 Models for Ambiguity Resolution in Parsing

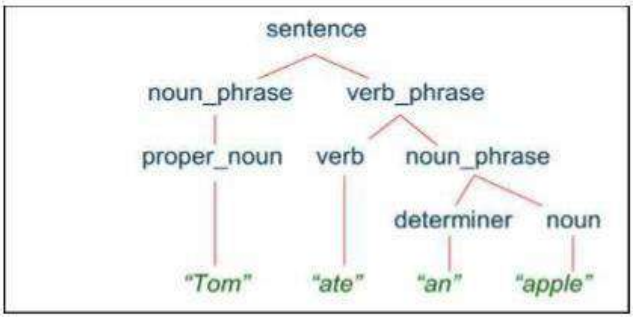
The parsing in NLP is the process of determining the syntactic structure of a text by analysing its constituent words based on an underlying grammar.

Example Grammar:

```
sentence -> noun_phrase, verb_phrase
noun_phrase -> proper_noun
noun_phrase -> determiner, noun
verb_phrase -> verb, noun_phrase
proper_noun -> [Tom]
noun -> [apple]
verb -> [ate]
determiner -> [an]
```

Then, the outcome of the parsing process would be a parse tree, where sentence is the root, intermediate nodes such as noun_phrase, verb_phrase etc. have children - hence they are called non-terminals and finally, the leaves of the tree 'Tom', 'ate', 'an', 'apple' are called terminals.

Parse Tree:



- A sentence is parsed by relating each word to other words in the sentence which depend on it.
- The syntactic parsing of a sentence consists of finding the correct syntactic structure of that sentence in the given formalism/grammar.
- Dependency grammar (DG) and phrase structure grammar (PSG) are two such formalisms.
- PSG breaks sentence into constituents (phrases), which are then broken into smaller constituents.
- Describe phrase, clause structure Example: NP, PP, VP etc.,
- DG: syntactic structure consists of lexical items, linked by binary asymmetric relations called dependencies.
- Interested in grammatical relations between individual words.
- Does propose a recursive structure rather a network of relations

- These relations can also have labels.
- treebank can be defined as a linguistically annotated corpus that includes some kind of syntactic analysis over and above part-of-speech tagging.

Constituency tree vs Dependency tree

- Dependency structures explicitly represent

- Head-dependent relations (directed arcs)
- Functional categories (arc labels)
- Possibly some structural categories (POS)

- Phrase structure explicitly represent

- Phrases (non-terminal nodes)
- Structural categories (non-terminal labels)
- Possible some functional categories (grammatical functions)

Defining candidate dependency trees for an input sentence

- **Learning:** scoring possible dependency graphs for a given sentence, usually by factoring the graphs into their component arcs
- **Parsing:** searching for the highest scoring graph for a given sentence

Syntax:

- In NLP, the syntactic analysis of natural language input can vary from being very low-level, such as simply tagging each word in the sentence with a part of speech (POS), or very high level, such as full parsing.
- In syntactic parsing, ambiguity is a particularly difficult problem because the most possible analysis has to be chosen from an exponentially large number of alternative analyses.
- From tagging to full parsing, algorithms that can handle such ambiguity have to be carefully chosen.
- Here we explore the syntactic analysis methods from tagging to full parsing and the use of supervised machine learning to deal with ambiguity.

2.1 Parsing Natural Language

- In a text-to-speech application, input sentences are to be converted to a spoken output that should sound like it was spoken by a native speaker of the language.
- Example: He wanted to go a drive in the country.
- There is a natural pause between the words derive and In in sentence that reflects an underlying hidden structure to the sentence.
- Parsing can provide a structural description that identifies such a break in the intonation.
- A simpler case: The cat who lives dangerously had nine lives.
- In this case, a text-to-speech system needs to know that the first instance of the word lives is a verb and the second instance is a noun before it can begin to produce the natural intonation for this sentence.
- This is an instance of the part-of-speech (POS) tagging problem where each word in the sentence is assigned a most likely part of speech.
- Another motivation for parsing comes from the natural language task of summarization, in which several documents about the same topic should be condensed down to a small digest of information.
- Such a summary may be in response to a question that is answered in the set of

documents.

- In this case, a useful subtask is to compress an individual sentence so that only the relevant portions of a sentence is included in the summary.
- For example: Beyond the basic level, the operations of the three products vary widely. The operations of the products vary.
- The elegant way to approach this task is to first parse the sentence to find the various constituents: where we recursively partition the words in the sentence into individual phrases such as a verb phrase or a noun phrase.

2.2 Treebanks: A Data-Driven Approach to Syntax

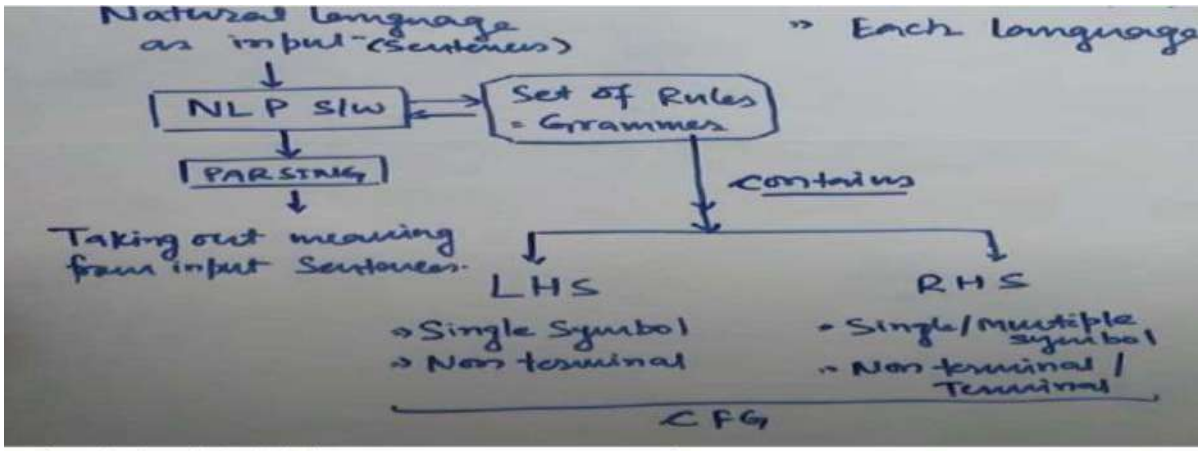
- Parsing recovers information that is not explicit in the input sentence.
- This implies that a parser requires some knowledge (syntactic rules) in addition to the input sentence about the kind of syntactic analysis that should be produced as output.
- One method to provide such knowledge to the parser is to write down a grammar of the language – a set of rules of syntactic analysis as a CFGs.
- In natural language, it is far too complex to simply list all the syntactic rules in terms of a CFG.
- The second knowledge acquisition problem- not only do we need to know the syntactic rules for a particular language, but we also need to know which analysis is the most plausible(probably) for a given input sentence.
- The construction of treebank is a data driven approach to syntax analysis that allows us to address both of these knowledge acquisition bottlenecks in one stroke.
- A treebank is simply a collection of sentences (also called a corpus of text), where each sentence is provided a complete syntax analysis.
- The syntactic analysis for each sentence has been judged by a human expert as the most possible analysis for that sentence.
- A lot of care is taken during the human annotation process to ensure that a consistent treatment is provided across the treebank for related grammatical phenomena.
- There is no set of syntactic rules or linguistic grammar explicitly provided by a treebank, and typically there is no list of syntactic constructions provided explicitly in a treebank.
- A detailed set of assumptions about the syntax is typically used as an annotation guideline to help the human experts produce the single-most plausible syntactic analysis for each sentence in the corpus.
- Treebanks provide a solution to the two kinds of knowledge acquisition bottlenecks.
- Treebanks solve the first knowledge acquisition problem of finding the grammar underlying the syntax analysis because the syntactic analysis is directly given instead of a grammar.
- In fact, the parser does not necessarily need any explicit grammar rules as long as it can faithfully produce a syntax analysis for an input sentence.
- Treebank solve the second knowledge acquisition problem as well.
- Because each sentence in a treebank has been given its most plausible(probable) syntactic analysis, supervised machine learning methods can be used to learn a scoring function over all possible syntax analyses.
- Two main approaches to syntax analysis are used to construct treebanks: dependency graph and phrase structure trees.
- These two representations are very closely related to each other and under some

assumptions, one representation can be converted to another.

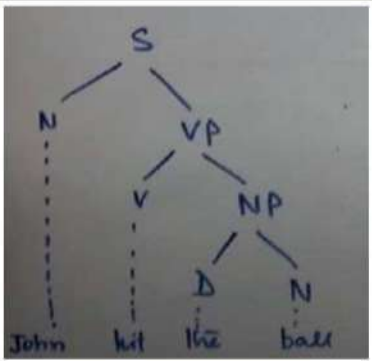
- Dependence analysis is typically favoured for languages such as Czech and Turkish, that have free word order.
- Phrase structure analysis is often used to provide additional information about long-distance dependencies and mostly languages like English and French.
- NLP: is the capability of the computer software to understand the natural language.
- There are variety of languages in the world.
- Each language has its own structure (SVO or SOV)->called grammar ->has certain set of rules->determines: what is allowed, what is not allowed.
- English: S O V Other languages: S V O or O S V
  I eat mango
- Grammar is defined as the rules for forming well-structured sentences.
- belongs to VN
- Different Types of Grammar in NLP
  1. Context-Free Grammar (CFG)
  2. Constituency Grammar (CG) or Phrase structure grammar
  3. Dependency Grammar (DG)

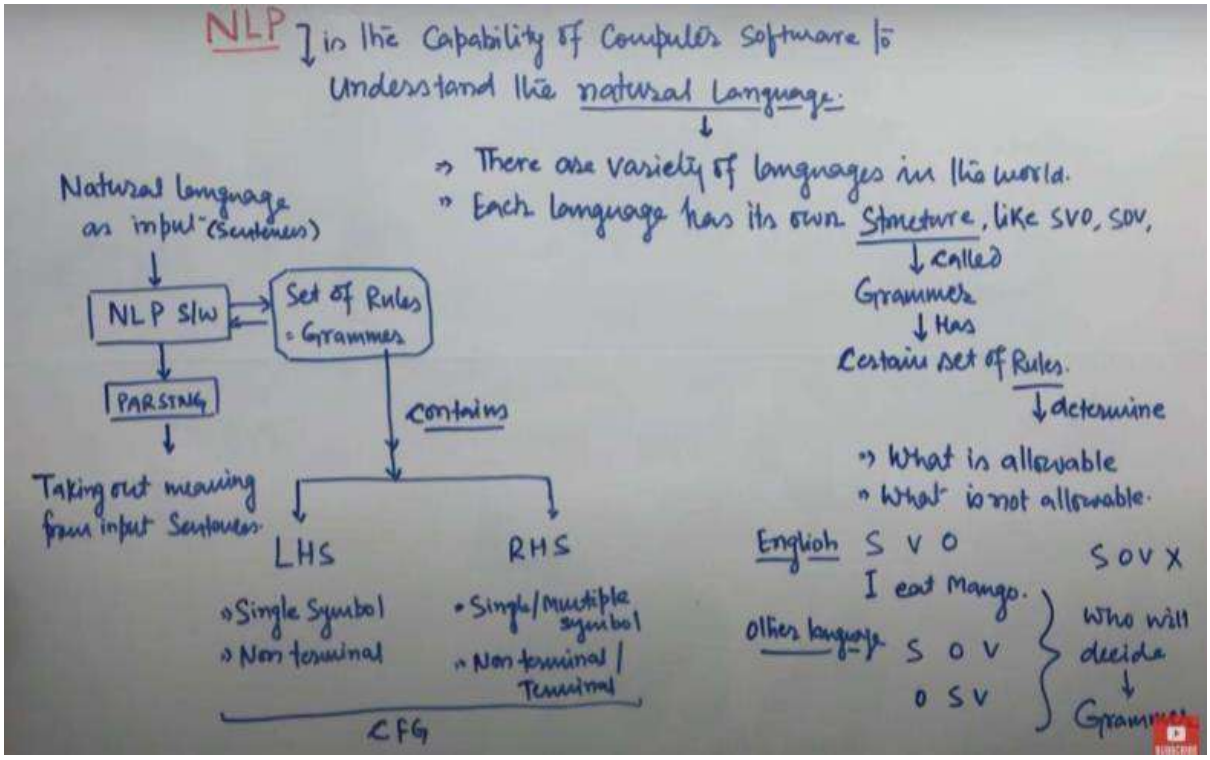
Context-Free Grammar (CFG)

- Mathematically, a grammar G can be written as a 4-tuple (N, T, S, P)
- N or V_N = set of non-terminal symbols, or variables.
- T or $\sum$ = set of terminal symbols.
- S = Start symbol where S ∈ N
- P = Production rules for Terminals as well as Non-terminals.
- It has the form $\alpha \rightarrow \beta$, where α and β are strings on $V_N \cup \sum$ at least one symbol of α.



- Example: Jogn hit the ball
  S -> NP VP
  VP -> V NP
  V ->hit
  NP-> DN
  D->the
  N->John|ball



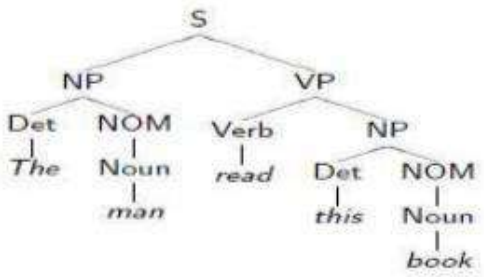


Application of grammar rewrite rules

S → NP VP	Det → *that* | *this* | *a* | *the*
S → Aux NP VP	Noun → *book* | *flight* | *meal* | *man*
S → VP	Verb → *book* | *include* | *read*
NP → Det NOM	Aux → *does*
NOM → Noun	
NOM → Noun NOM	
VP → Verb	
VP → Verb NP	

S → NP VP
→ Det NOM VP
→ *The* NOM VP
→ *The* Noun VP
→ *The man* VP
→ *The man* Verb NP
→ *The man read* NP
→ *The man read* Det NOM
→ *The man read this* NOM
→ *The man read this* Noun
→ *The man read this book*

Parse tree

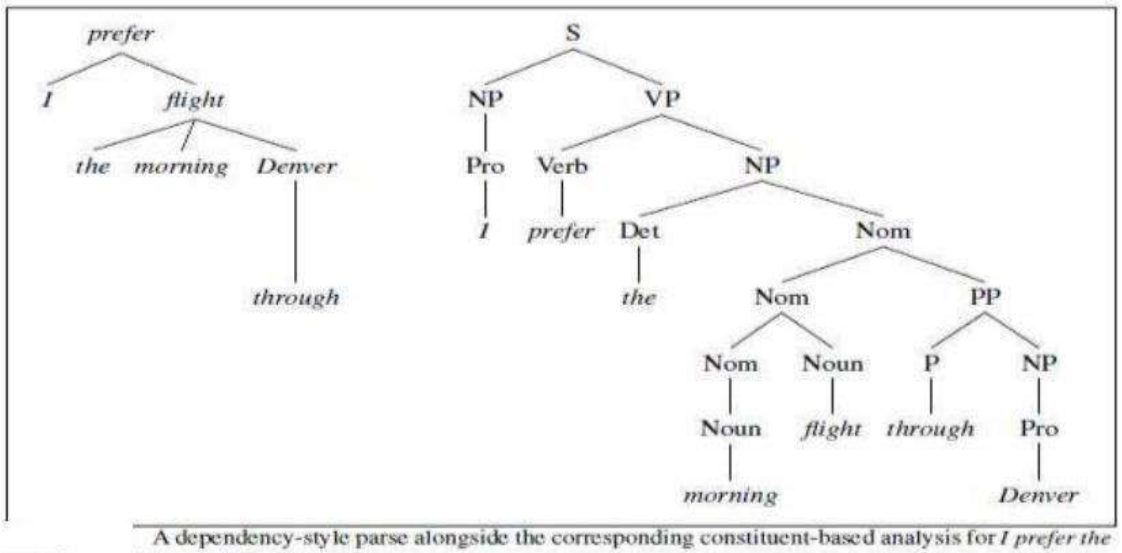


2.3 Representation of Syntactic Structure

2.3.1 Syntax Analysis Using Dependency Graphs

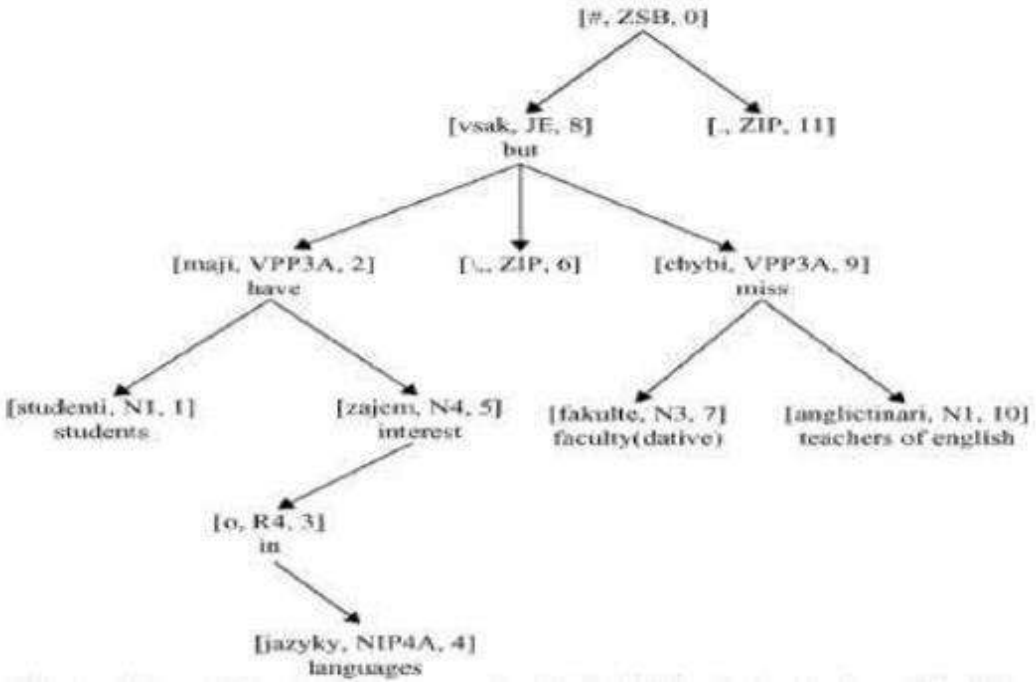
- The main philosophy behind dependency graphs is to connect a word- the head of a phrase- with the dependents in that phrase.
- The notation connects a head with its dependent using a directed (asymmetric) connections.
- Dependency graphs, just like phrase structures trees, is a representation that is consistent with many different linguistic frameworks.
- The words in the input sentence are treated as the only vertices in the graph, which are linked together by directed arcs representing syntactic dependencies.

- In dependency-based syntactic parsing, the task is to derive a syntactic structure for an input sentence by identifying the syntactic head of each word in the sentence.
- This defines a dependency graph, where the nodes are the words of the input sentence and arcs are the binary relations from head to dependent.



A dependency-style parse alongside the corresponding constituent-based analysis for *I prefer the morning flight through Denver.*

- The dependency tree analyses, where each word depends on exactly one parent, either another word or a dummy root symbol.
- By convention, in dependency tree 0 index is used to indicate the root symbol and the directed arcs are drawn from the head word to the dependent word.
- In the Fig shows a dependency tree for Czech sentence taken from the Prague dependency treebank.

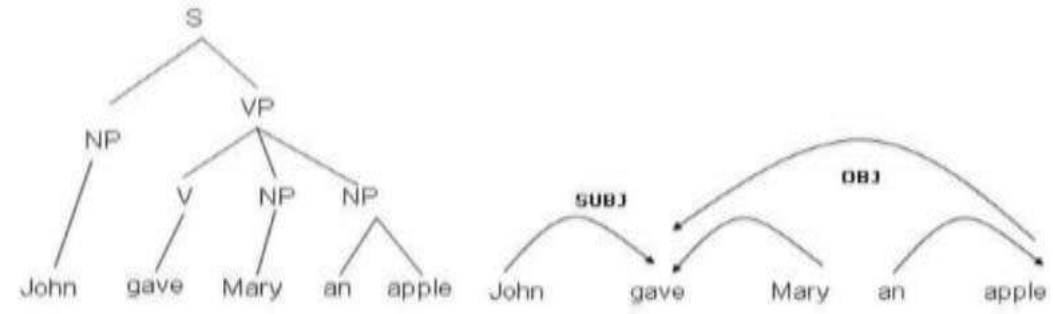


- Each node in the graph is a word, its part of speech and the position of the word in the sentence. • For example [fakulte, N3,7] is the seventh word in the sentence with POS tag N3.
- The node [#, ZSB,0] is the root node of the dependency tree.

- There are many variations of dependency syntactic analysis, but the basic textual format for a dependency tree can be written in the following form.
- Where each dependent word specifies the head Word in the sentence, and exactly one word is dependent to the root of the sentence.

Index	Word	Part of Speech	Head	Label
1	They	PRP	2	SBJ
2	persuaded	VBD	0	ROOT
3	Mr.	NNP	4	NMOD
4	Trotter	NNP	2	IOBJ
5	to	TO	6	VMOD
6	take	VB	2	OBJ
7	it	PRP	6	OBJ
8	back	RB	6	PRT
9	.	.	2	P

- An important notation in dependency analysis is the notation of projectivity, which is a constraint imposed by the linear order of words on the dependencies between words.
- A projective dependency tree is one where if we put the words in a linear order based on the sentence with the root symbol in the first position, the dependency arcs can be drawn above the words without any crossing dependencies.



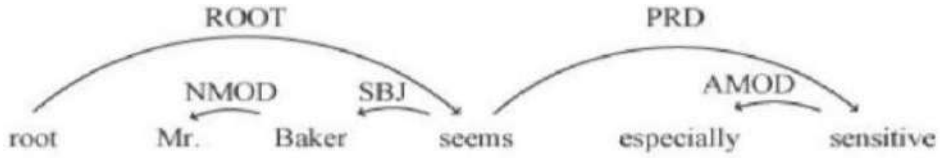
Phrase structure tree and the corresponding Dependency structure tree

2.3.2 Syntax Analysis Using Phrase Structures Trees

- A Phrase Structure syntax analysis of a sentence derives from the traditional sentence diagrams that partition a sentence into constituents, and larger constituents are formed by meaning smaller ones.
- Phrase structure analysis also typically incorporate ideas from generative grammar (from linguistics) to deal with displaced constituents or apparent long-distance relationships between heads and constituents.
- A phrase structure tree can be viewed as implicitly having a predicate-argument structure associated with it.
- Sentence includes a subject and a predicate. The subject is a noun phrase (NP) and the predicate is a verb phrase.
- For example, the phrase structure analysis: Mr. Baker seems especially sensitive, taken from the Penn Treebank.
- The subject of the sentence is marked with the SBJ marker and predicate of the sentence is marked with the PRD marker.

```
(S (NP-SBJ (NNP Mr.)
           (NNP Baker))
   (VP (VBZ seems)
       (ADJP-PRD (RB especially)
                 (JJ sensitive))))
```

- NNP: proper noun, singular VBZ: verb, third person singular present ADJP: adjective phrase RB: adverb JJ: adjective
- The same sentence gets the following dependency tree analysis: some of the information from the bracketing labels from the phrase structure analysis gets mapped onto the labelled arcs of the dependency analysis.



- To explain some details of phrase structure analysis in treebank, which was a project that annotated 40,000 sentences from the wall street journal with phrase structure tree.

```
(SBARQ (WHNP-1 What)
       (SQ is (NP-SBJ Tim)
              (VP eating (NP *T*-1)))
       ?)
```

  - The SBARQ label marks what questions ie those that contain a gap and therefore require a trace.
  - Wh- moved noun phrases are labelled WHNP and put inside SBARQ. They bear an identity index that matches the reference index on the *T* in the position of the gap.
  - However, questions that are missing both subject and auxiliary are label SQ
  - NP-SBJ noun phrases cab be subjects.
  - *T* traces for wh- movement and this empty trace has an index (here it is 1) and associated with the WHNP constituent with the same index.

Parsing Algorithms

- Given an input sentence, a parser produces an output analysis of that sentence.
- Treebank parsers do not need to have an explicit grammar, but to discuss the parsing algorithms simpler, we use CFG.
- The simple CFG G that can be used to derive string such as a and b or c from the start symbol N.

```
N -> N 'and' N
N -> N 'or' N
N -> 'a' | 'b' | 'c'
```

- An important concept for parsing is a derivation.
- For the input string a and b or c, the following sequence of actions separated by symbol represents a sequence of steps called derivation.

```
N
=> N 'or' N
=> N 'or c'
=> N 'and' N 'or c'
=> N 'and b or c'
=> 'a and b or c'
```

```
N -> N 'and' N
N -> N 'or' N
N -> 'a' | 'b' | 'c'
```

- In this derivation, each line is called a sentential form.
- In the above derivation, we restricted ourselves to only expanded on the rightmost nonterminal in each sentential form.
- This method is called the rightmost derivation of the input using a CFG.
- This derivation sequence exactly corresponds to the construction of the following parse tree from left to right, one symbol at a time.

```
(N (N (N a)
      and
      (N b))
   or
   (N c))
```

- However, a unique derivation sequence is not guaranteed.
- There can be many different derivations.
- For example, one more rightmost derivation that results following parse tree.

```
(N (N a)
   and
   (N (N b)
      or
      (N c)))
```

```
'a and b or c'
=> N 'and b or c'          # use rule N -> a
=> N 'and' N 'or c'        # use rule N -> b
=> N 'and' N 'or' N        # use rule N -> c
=> N 'and' N               # use rule N -> N or N
=> N                       # use rule N -> N and N
```

Shift Reduce Parsing

- To build a parser, we need an algorithm that can perform the steps in the above rightmost derivation for any grammar and for any input string.
- Every CFG turns out to have an automaton that is equivalent to it, called pushdown automata (just like regular expression can be converted to finite-state automata).
- An algorithm for parsing that is general for any given CFG and input string.
- The algorithm is called shift-reduce parsing which uses two data structures: a buffer for input symbols and a stack for storing CFG symbols.

Shift Reduce Parser

Start with the sentence to be parsed in an input buffer.

- a "shift" action correponds to pushing the next input symbol from the buffer onto the stack
- a "reduce" action occurrs when we have a rule's RHS on top of the stack. To perform the reduction, we pop the rule's RHS off the stack and replace it with the terminal on the LHS of the corresponding rule.

(When either "shift" or "reduce" is possible, choose one arbitrarily.)

If you end up with only the START symbol on the stack, then success!
If you don't, and you cannot and no "shift" or "reduce" actions are possible, backtrack.

		a and b or c	Init
a	a	and b or c	shift a
(N a)	N	and b or c	reduce N -> a
(N a) and	N and	b or c	shift and
(N a) and b	N and b	or c	shift b
(N a) and (N b)	N and N	or c	reduce N -> b
(N (N a) and (N b))	N	or c	reduce N -> a
(N (N a) and (N b)) or	N or	c	shift or
(N (N a) and (N b)) or c	N or c		shift c
(N (N a) and (N b)) or (N c)	N or N		reduce N -> c
(N (N (N a) and (N b)) or (N c))	N		reduce N -> N or N
(N (N (N a) and (N b)) or (N c))	N		Accept!

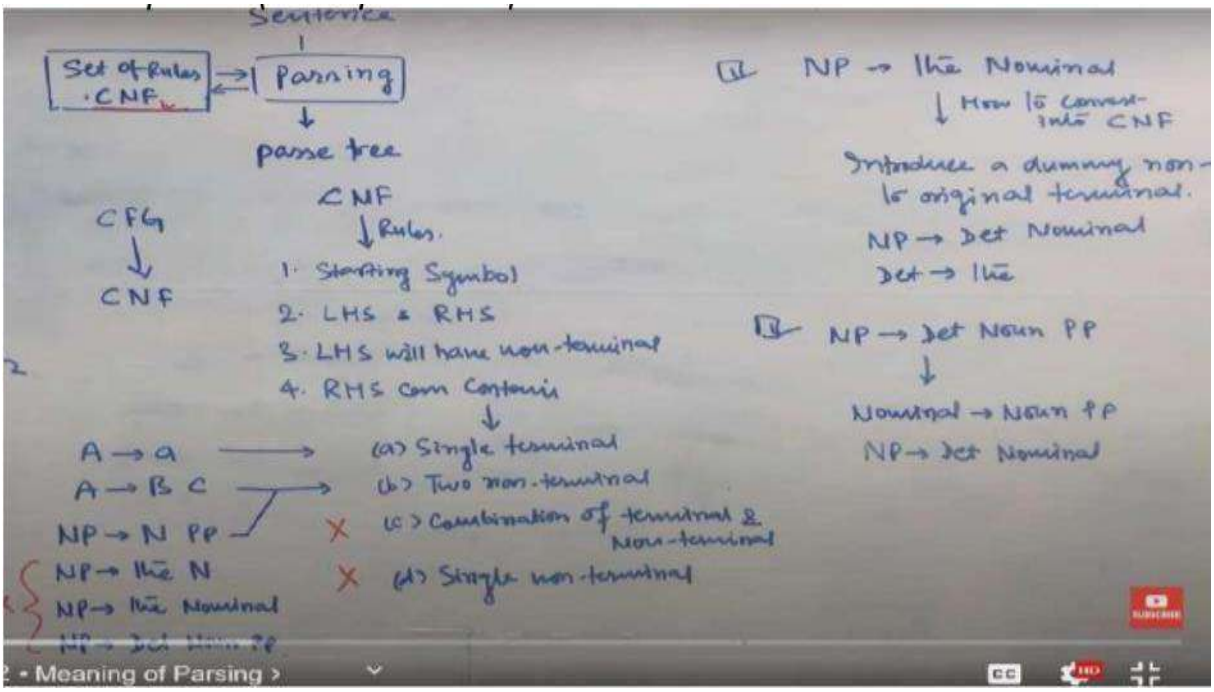
```
N -> N 'and' N
N -> N 'or' N
N -> 'a' | 'b' | 'c'
```

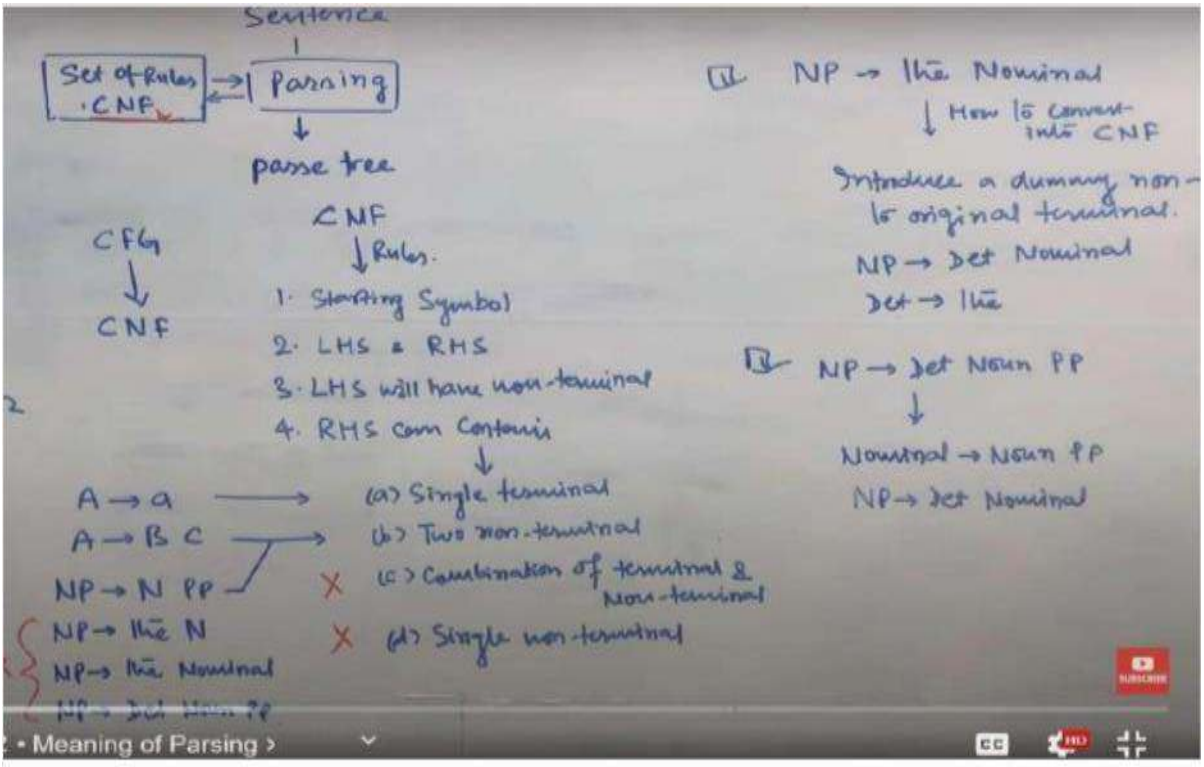
S.No	Parse Tree	Stack	Input	Action
1			a and b or c	Init
2	a	a	and b or c	Shift a
3	(N a)	N	and b or c	Reduce N->a
4	(N a) and	N and	b or c	Shift and
	(N a) and b	N and b	or c	Shift b
	(N a) and (N b)	N and N	or c	Reduce N->b
	(N (N a) and (N b))	N	or c	Reduce N->N and N
	(N (N a) and (N b)) or	N or	c	Shift or
9	(N (N a) and (N b)) or c	N or c		Shift c
10	(N (N a) and (N b)) or (N c)	N or N		Reduce N->c
11	(N (N (N a) and (N b)) or (N c))	N		Reduce N->N or N

1. Start with an empty stack and the buffer contains the input string.
2. Exit with success if the top of the stack contains the start symbol of the grammar and if the buffer is empty.
3. Choose between the following two steps (if the choice is ambiguous, choose one based on an oracle):
   - Shift a symbol from the buffer onto the stack.
   - If the top k symbols of the stack are $\alpha_1 \ldots \alpha_k$ which corresponds to the right-hand side of a CFG rule $A \rightarrow \alpha_1 \ldots \alpha_k$ then replace the top k symbols with the left-hand side non-terminal A.
4. Exit with failure if no action can be taken in previous step.
5. Else, go to Step 2.

Hypergraphs and Chart Parsing (CYK Parsing)

- CFG s in the worst case such a parser might have to resort to backtracking, which means re-parsing the input which leads to a time that is exponential in the grammar size in the worst case.
- Variants of this algorithm (CYK) are often used in statistical parsers that attempt to search the space of possible parse trees without the limitation of purely left to right parsing.
- One of the earliest recognition parsing algorithm is CYK (Cocke, Kasami and younger) parsing algorithm and It works only with CNF(Chomsky normal form).



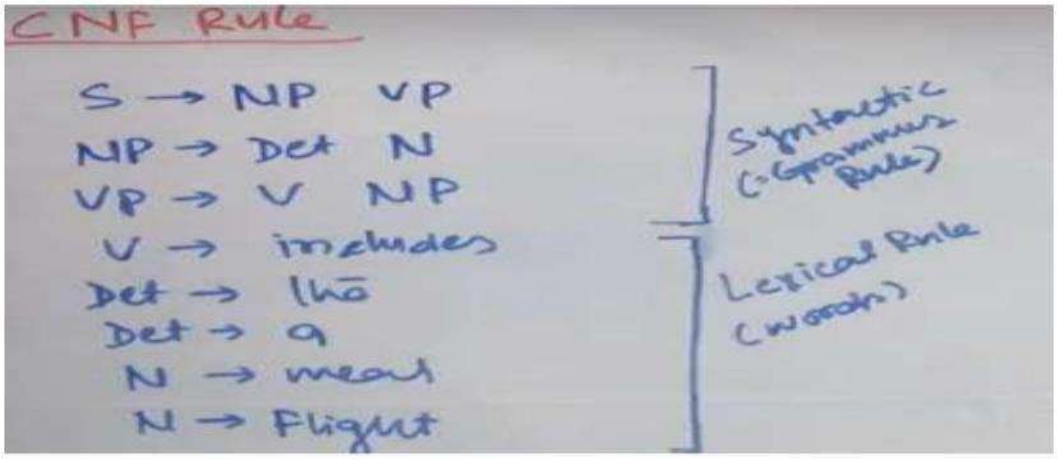


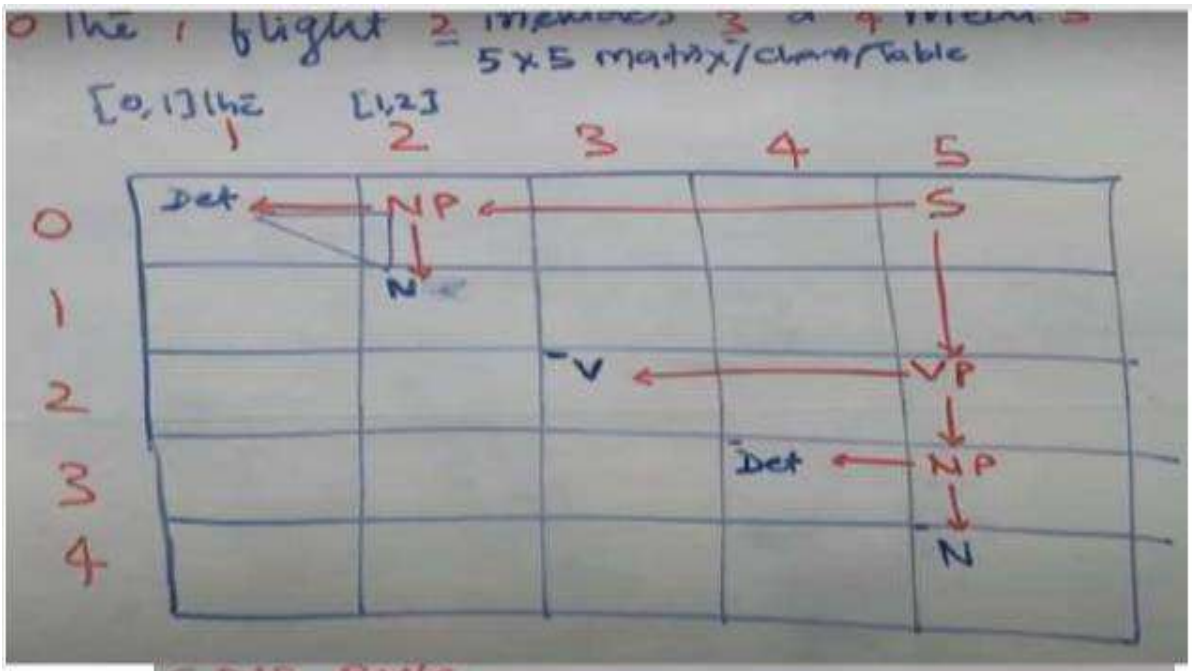
```
N -> N 'and
N -> N 'or' N
N -> 'a' | 'b' | 'c'
```

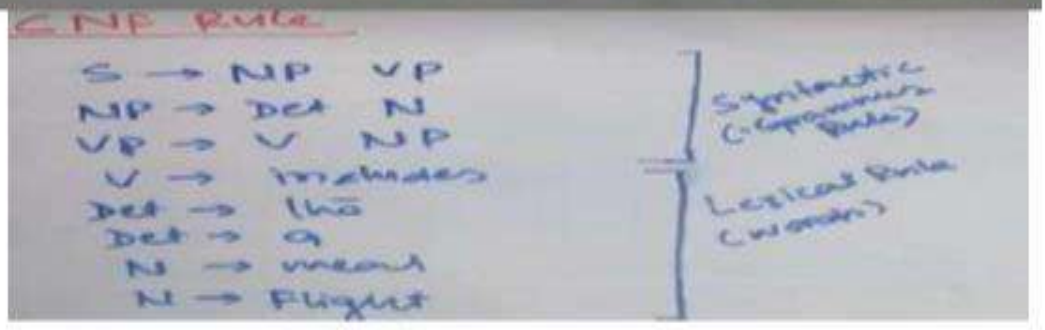
is re-written into a new CFG G_c .
two non-terminals. This is done by introduc..

```
N -> N N^
N^ -> 'and' N
N -> N Nv
Nv -> 'or' N
N -> 'a' | 'b' | 'c'
```

CYK example:







G_f that represents the *forest* of parse trees is shown below. Imagine that the input string is broken up into spans *0 a 1 and 2 b 3 or 4 c 5* so that *a* is span 0,1 and the string *b or c* is the span 2,5 in this string. The non-terminals in this forest grammar G_c include the span information. The different parse trees that can be generated using this grammar are the valid parse trees for the input sentence.

```
N[0,5] -> N[0,1] N^[1,5]
N[0,3] -> N[0,1] N^[1,3]
N^[1,3] -> 'and'[1,2] N[2,3]
N^[1,5] -> 'and'[1,2] N[2,5]
N[0,5] -> N[0,3] Nv[3,5]
N[2,5] -> N[2,3] Nv[3,5]
Nv[3,5] -> 'or'[3,4] N[4,5]
N[0,1] -> 'a'[0,1]
N[2,3] -> 'b'[2,3]
N[4,5] -> 'c'[4,5]
```

```
N -> N 'and' N
N -> N 'or' N
N -> 'a' | 'b' | 'c'
```

is re-written into a new CFG G_c where the right hand side only contains up to two non-terminals. This is done by introducing two new non-terminals N^ and Nv:

```
N -> N N^
N^ -> 'and' N
N -> N Nv
Nv -> 'or' N
N -> 'a' | 'b' | 'c'
```

In this view, a parsing algorithm is defined as taking as input a CFG and an input string and producing a specialized CFG that is a compact representation of all legal parses for the input. A parser has to create all the valid specialized rules or alternatively create a path from the start symbol non-terminal that spans the entire string to the leaf nodes that are the input tokens.

Let us examine the steps the parser has to take to construct a specialized CFG. First let us consider the rules that generate only lexical items:

```
N[0,1] -> 'a'[0,1]
N[2,3] -> 'b'[2,3]
N[4,5] -> 'c'[4,5]
```

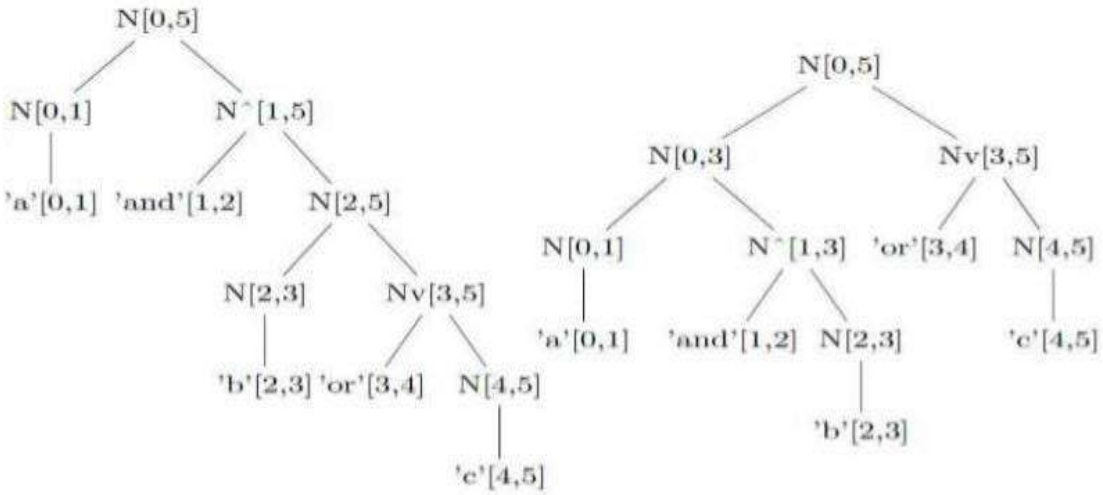


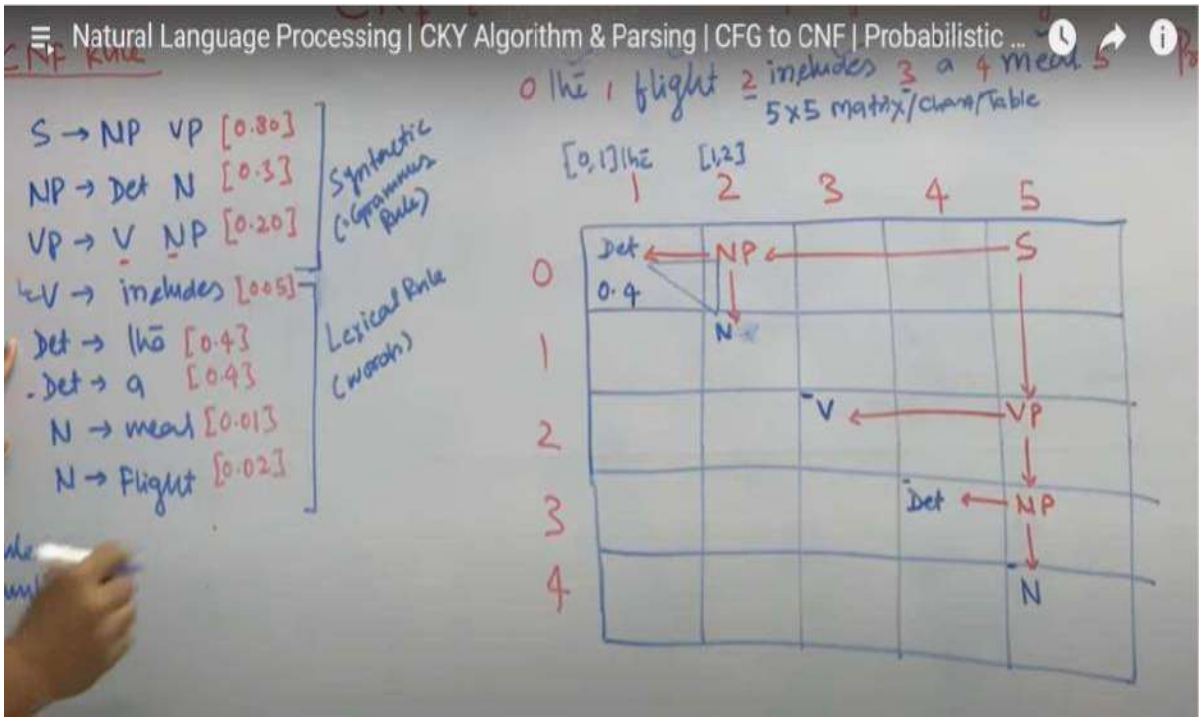
Figure 1.8. Parse trees embedded in the *specialized* CFG for a particular input string. The nodes with the same label, e.g. *N[0,5]* can be merged to form a hypergraph representation of all parses for the input.

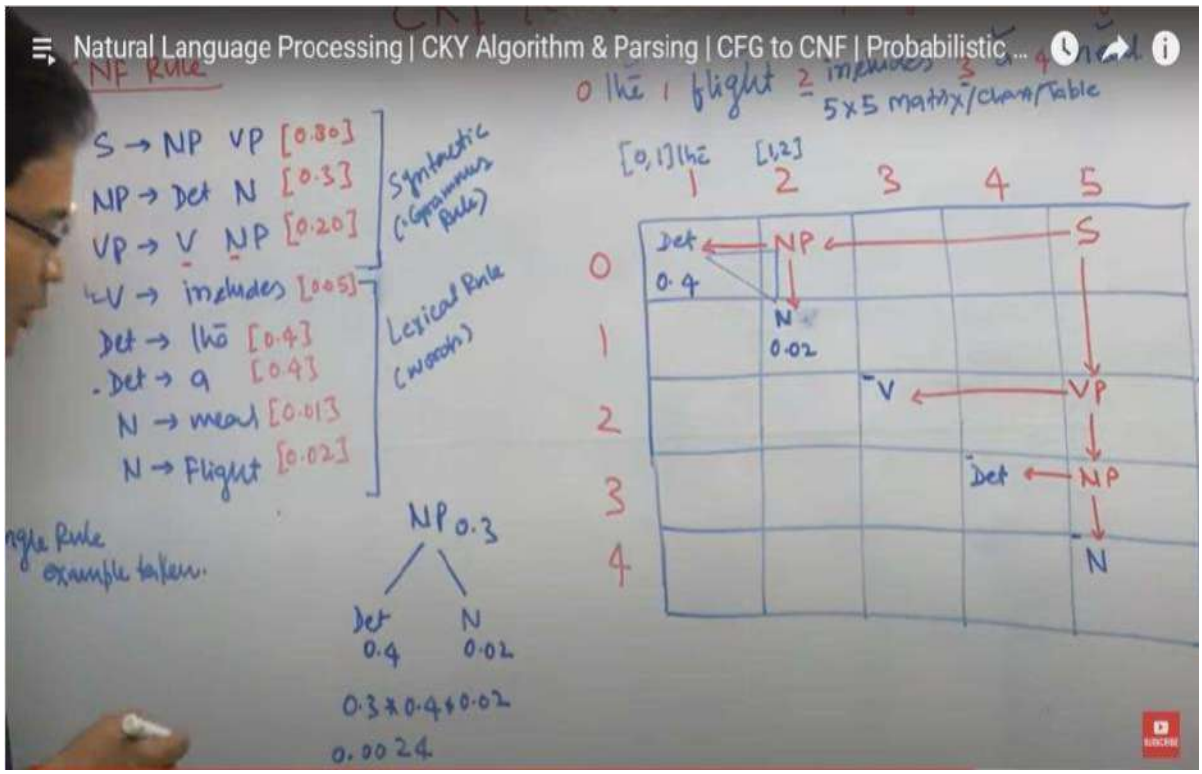
```
N[0,5] -> N[0,1] N^[1,5]
N[0,3] -> N[0,1] N^[1,3]
N^[1,3] -> 'and'[1,2] N[2,3]
N^[1,5] -> 'and'[1,2] N[2,5]
N[0,5] -> N[0,3] Nv[3,5]
N[2,5] -> N[2,3] Nv[3,5]
Nv[3,5] -> 'or'[3,4] N[4,5]
N[0,1] -> 'a'[0,1]
N[2,3] -> 'b'[2,3]
N[4,5] -> 'c'[4,5]
```

```
N -> N 'and' N
N -> N 'or' N
N -> 'a' | 'b' | 'c'
```

is re-written into a new CFG G_c where the right hand side only contains up to two non-terminals. This is done by introducing two new non-terminals N^ and Nv:

```
N -> N N^
N^ -> 'and' N
N -> N Nv
Nv -> 'or' N
N -> 'a' | 'b' | 'c'
```





Models for Ambiguity Resolution in Parsing

Here we discuss on modelling aspects of parsing: how to design features and ways to resolve ambiguity in parsing.

Probabilistic context-free grammar

• Ex: John bought a shirt with pockets

```
(S (NP John)
   (VP (VP (V bought)
           (NP (D a)
               (N shirt)))
       (PP (P with)
           (NP pockets))))
```

```
(S (NP John)
   (VP (V bought)
       (NP (NP (D a)
               (N shirt))
           (PP (P with)
               (NP pockets)))))
```

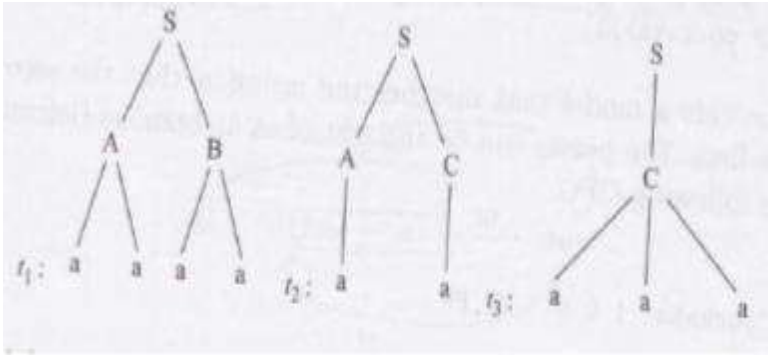
- Here we want to provide a model that matches the intuition that the second tree above is preferred over the first.
- The parses can be thought of as ambiguous (leftmost to rightmost) derivation of the following CFG:

```
S -> NP VP
NP -> 'John' | 'pockets' | D N | NP PP
VP -> V NP | VP PP
V -> 'bought'
D -> 'a'
N -> 'shirt'
PP -> P NP
P -> 'with'
```

- We assign scores or probabilities to the rules in CGF in order to provide a score or probability for each derivation.

```
S -> NP VP (1.0)
NP -> 'John' (0.1) | 'pockets' (0.1) | D N (0.3) | NP PP (0.5)
VP -> V NP (0.9) | VP PP (0.1)
V -> 'bought' (1.0)
D -> 'a' (1.0)
N -> 'shirt' (1.0)
PP -> P NP (1.0)
P -> 'with' (1.0)
```

- From these rule probabilities, the only deciding factor for choosing between the two parses for John brought a shirt with pockets in the two rules NP->NP PP and VP-> VP PP. The probability for NP -> NP PP is set higher in the preceding PCFG.
- The rule probabilities can be derived from a treebank, consider a treebank with three tress t1, t2, t3

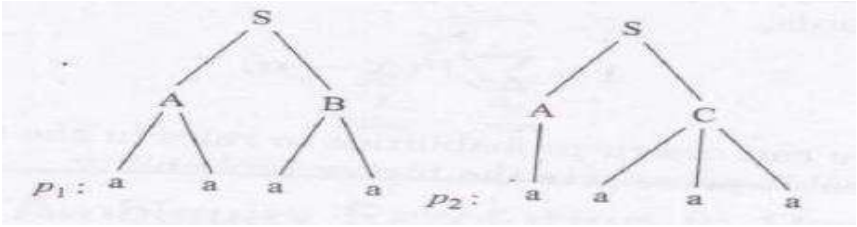


- If we assume that tree t1 occurred 10 times in the treebank, t2 occurred 20 times and t3 occurred 50 times, then the PCFG we obtain from this treebank is:

$\frac{10}{10+20+50} = 0.125 \quad S \rightarrow A\ B$
$\frac{20}{10+20+50} = 0.25 \quad S \rightarrow A\ C$
$\frac{50}{10+20+50} = 0.625 \quad S \rightarrow C$
$\frac{10}{10+20} = 0.334 \quad A \rightarrow a\ a$
$\frac{20}{10+20} = 0.667 \quad A \rightarrow a$
$\frac{20}{20+50} = 0.285 \quad B \rightarrow a\ a$
$\frac{50}{20+50} = 0.714 \quad C \rightarrow a\ a\ a$

$\frac{10}{10+20+50} = 0.125 \quad S \rightarrow A\ B$
$\frac{20}{10+20+50} = 0.25 \quad S \rightarrow A\ C$
$\frac{50}{10+20+50} = 0.625 \quad S \rightarrow C$
$\frac{10}{10+20} = 0.334 \quad A \rightarrow a\ a$
$\frac{20}{10+20} = 0.667 \quad A \rightarrow a$
$\frac{20}{20+50} = 0.285 \quad B \rightarrow a\ a$
$\frac{50}{20+50} = 0.714 \quad C \rightarrow a\ a\ a$

- For input a a a there are two parses using the above PCFG: the probability P1 =0.125 0.334 0.285 = 0.01189 p2=0.25 0.667 0.714 =0.119.
- The parse tree p2 is the most likely tree for that input.



Generative models

- **To find the most plausible parse tree, the parser has to choose between the possible derivations each of which can be represented as a sequence of decisions.**
- **Let each derivation D = d1,d2,…..,dn, which is the sequence of decisions used to build the parse tree.**
- **Then for input sentence x, the output parse tree y is defined by the sequence of steps in the derivation.**
- **The probability for each derivation:**

$$P(x,y) = P(d_1,\ldots,d_n) = \prod_{i=1}^{n} P(d_i \mid d_1,\ldots,d_{i-1})$$

- **The conditioning context in the probability P(di|d1,…….,di-1) is called the history and corresponds to a partially built parse tree (as defined by the derived sequence).**
- **We make a simplifying assumption that keeps the conditioning context to a finite set by grouping the histories into equivalence classes using a function:**

$$P(d_1,\ldots,d_n) = \prod_{i=1}^{n} P(d_i \mid \Phi(d_1,\ldots,d_{i-1}))$$

Using Φ, each history $H_i = d_1,\ldots,d_{i-1}$ for all x, y is mapped to some fixed finite set of feature functions of the history $\phi_1(H_i),\ldots,\phi_k(H_i)$. In terms of these k feature functions:

$$P(d_1,\ldots,d_n) = \prod_{i=1}^{n} P(d_i \mid \phi_1(H_i),\ldots,\phi_k(H_i))$$

Discriminative models for Parsing

• Colins created a simple notation and framework that describes various discriminative approaches to learning for parsing.

• This framework is called global linear model.
• Let x be a set of inputs and y be a set of possible outputs that can be a sequence of POS tags or a parse tree or a dependency analysis.
• Each xƐx and yƐy is mapped to a d-dimensional feature vector ø(x,y), with each dimension being a real number.
• A weight parameter vector wƐRd assigns a weight to each feature in ø(x,y), representing the importance of that feature.

- The value of ø(x,y).w is the score of (x,y) . The height the score, the more possible it is that y is the output of x.
- The function GEN(x) generates the set of possible outputs y for a given x.
- Having ø(x,y).w and GEN(x) specified, we would like to choose the height scoring candidate $y*$ from GEN(x) as the most possible output

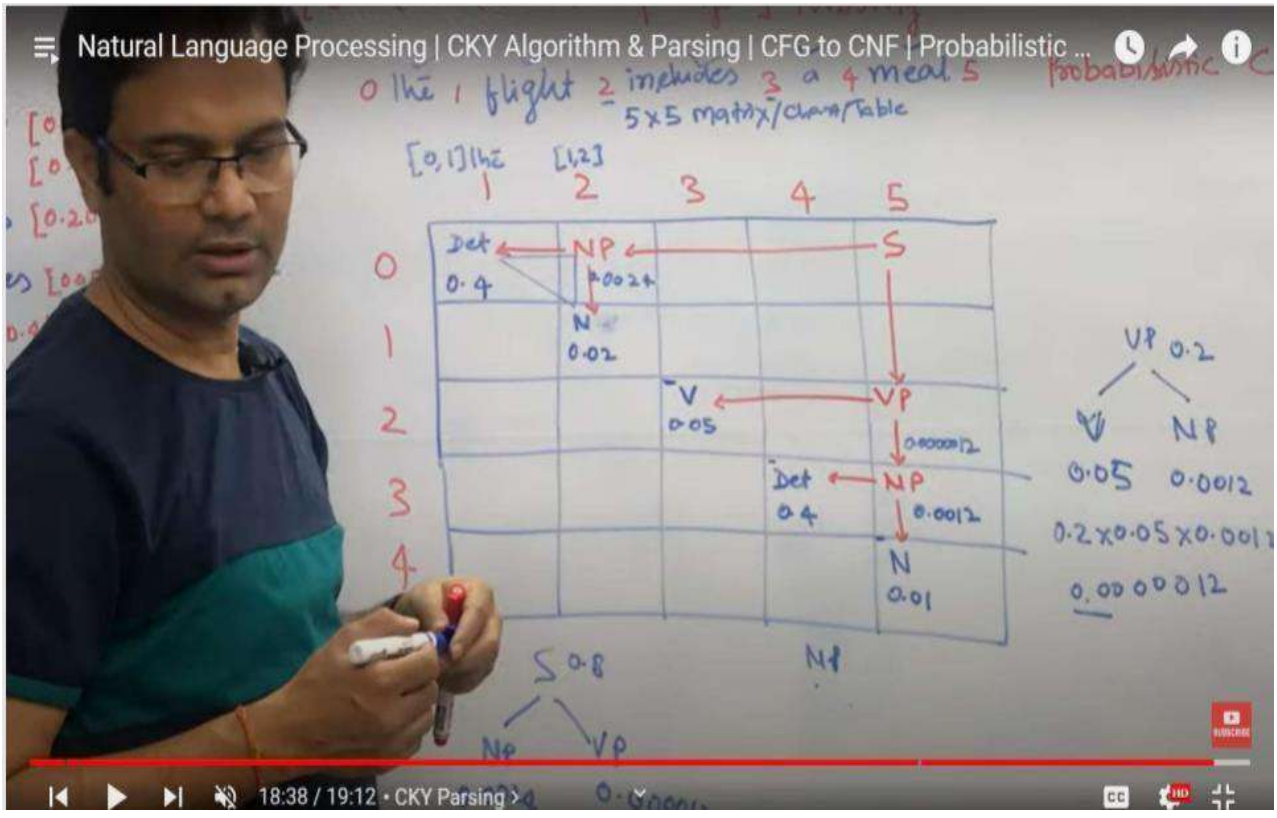
$$F(x) = \underset{y \in GEN(x)}{\operatorname{argmax}} p(y \mid x, \mathbf{w})$$

where F(x) returns the highest scoring output $y*$ from GEN(x)
• A conditional random field (CRF) defines the conditional probability as a linear score for each candidate y and a global normalization term:

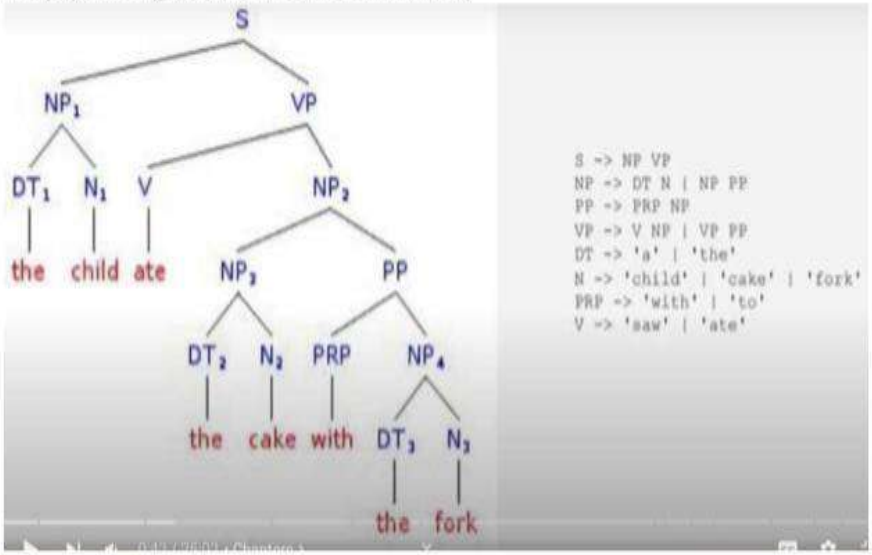
$$\log p(y \mid x, \mathbf{w}) = \Phi(x, y) \cdot \mathbf{w} - \log \sum_{y' \in GEN(x)} \exp(\Phi(x, y') \cdot \mathbf{w})$$

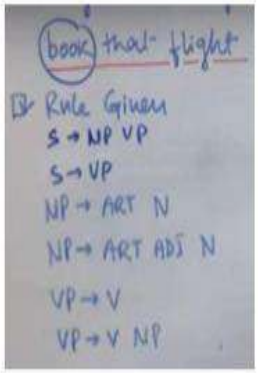
- A simple linear model that ignores the normalization term is:

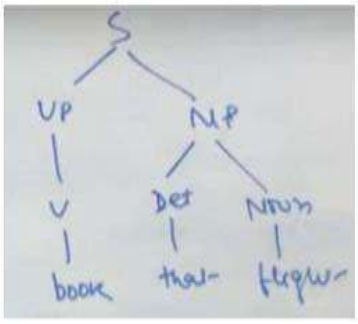
$$F(x) = \underset{y \in GEN(x)}{\operatorname{argmax}} \Phi(x, y) \cdot \mathbf{w}$$

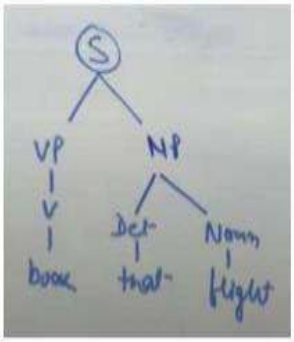


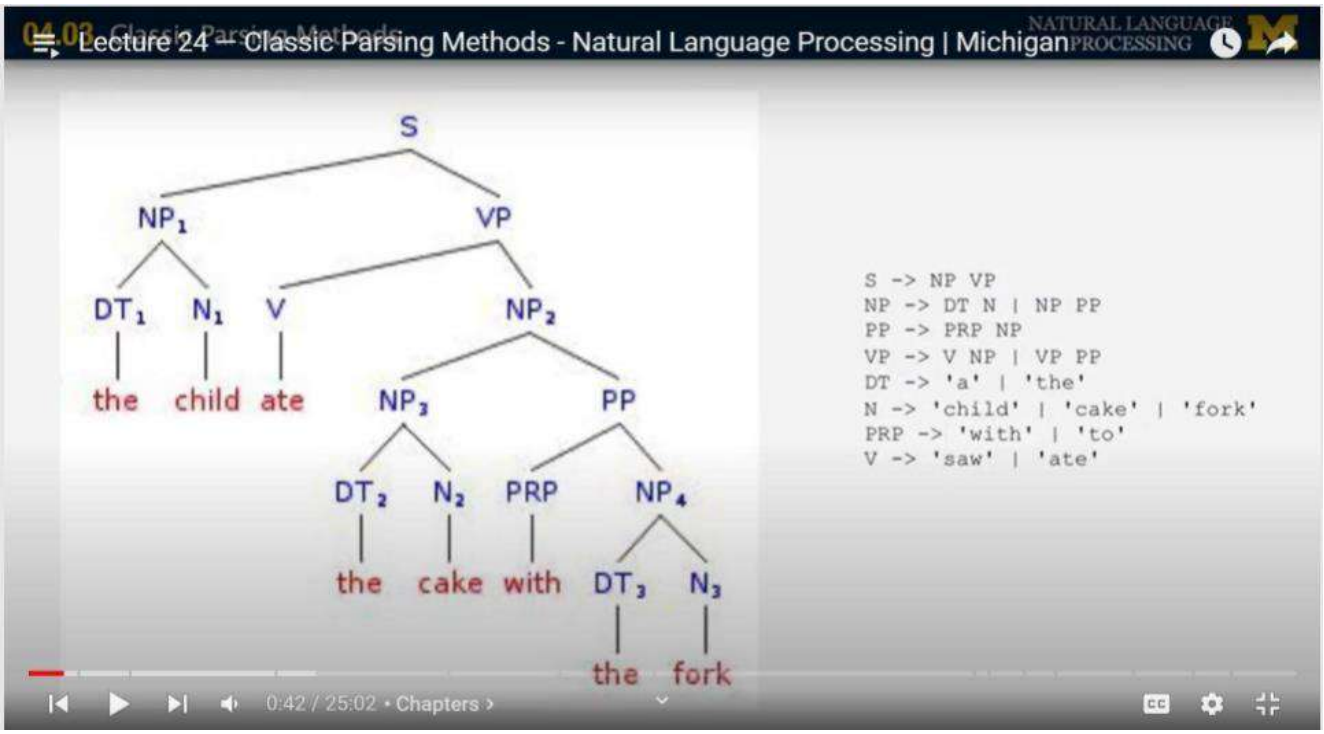
- There are two general approaches to parsing 1.Top down parsing (start with start symbol) 2.Botttom up parsing (start from terminals)











Unit-3: N-gram Language Models (Part-I)

Uses of Language Modelling:

1. Predicting is difficult—especially about the future What word, for example, is likely to follow

Please turn your homework ...

Hopefully, most of you concluded that a very likely word is in, or possibly over, but probably not refrigerator or the.

We will formalize this intuition by introducing models that assign a probability to each possible next word. The same models will also serve to assign a probability to an entire sentence. Such a model, for example, could predict that the following sequence has a much higher probability of appearing in a text:

all of a sudden I notice three guys standing on the sidewalk

than does this same set of words in a different order:

on guys all I of notice sidewalk three a sudden standing the

2. Why would you want to predict upcoming words, or assign probabilities to sentences? Probabilities are essential in any task in which we have to identify words in noisy, ambiguous input, like **speech recognition**. For a speech recognizer to realize that you said

I will be back soonish and not **I will be bassoon dish**, it helps to know that back soonish is a much more probable sequence than bassoon dish.

3. For writing tools like **spelling correction or grammatical error correction**, we need to find and correct errors in writing like **Their are two midterms**, in which **There was mistyped as Their**, or **Everything has improve**, in which improve should have been improved. The phrase **There are** will be much more probable than **Their are**, and has improved than has improve, allowing us to help users by detecting and correcting these errors.

4. Assigning probabilities to sequences of words is also essential in **machine translation.** Suppose we are translating a Chinese source sentence:

他 向 记者 介绍了 主要 内容

He to reporters introduced main content

As part of the process we might have built the following set of potential rough English translations:

he introduced reporters to the main contents of the statement

he briefed to reporters the main contents of the statement

he briefed reporters on the main contents of the statement

5. Probabilities are also important for **augmentative and alternative communication AAC systems.** People often use such AAC devices if they are physically unable to speak or sign but can instead use **eye gaze or other specific movements** to select words from a menu to be spoken by the system. Word prediction can be used to suggest likely words for the menu.

Language Models: Models that assign probabilities to sequences of words are called language models or LMs. The simplest model that assigns probabilities to sentences and sequences of words are the **n-gram**. An n-gram is a sequence of n words: a **2-gram** (which we'll call bigram) is a two-word sequence of words like "please turn", "turn your", or "your homework", and a **3-gram** (a trigram) is a three-word sequence of words like "please turn your", or "turn your homework".

We'll see how to use n-gram models to estimate the probability of the last word of an n-gram given the previous words, and also to assign probabilities to entire sequences. The n-gram models are much simpler than state-of-the art neural language models based on the RNNs and transformers.

N-Grams

P(w|h), the probability of a word w given some history h. Suppose the history h is **"its water** is so transparent that" and we want to know the probability that the next word is **the**:

$$P(the|its\ water\ is\ so\ transparent\ that).$$

One way to estimate this probability is from relative frequency counts: take a very large corpus, count the number of times we see its water is so transparent that, and count the number of times this is followed by the. This would be answering the question "Out of the times we saw the history h, how many times was it followed by the word w", as follows:

$$P(the|its\ water\ is\ so\ transparent\ that) = \frac{C(its\ water\ is\ so\ transparent\ that\ the)}{C(its\ water\ is\ so\ transparent\ that)}$$

With a large enough corpus, such as the web, we can compute these counts and estimate the probability. While this method of estimating probabilities directly from counts works fine in

many cases, it turns out that even the web isn't big enough to give us good estimates in most cases. This is because language is creative; new sentences are created all the time, and we won't always be able to count entire sentences. Even simple extensions of the example sentence may have counts of zero on the web (such as "Walden Pond's water is so transparent that the"; well, used to have counts of zero). Similarly, if we wanted to know the joint probability of an entire sequence of words like its water is so transparent, we could do it by asking "out of all possible sequences of five words, how many of them are its water is so transparent?" We would have to get the count of its water is so transparent and divide by the sum of the counts of all possible five word sequences. That seems rather a lot to estimate!

For this reason, we'll need to introduce more clever ways of estimating the probability of a word w given a history h, or the probability of an entire word sequence W. Now, how can we compute probabilities of entire sequences like $P(w_1;w_2;\ldots;w_n)$? One thing we can do is decompose this probability using the chain rule of probability:

Applying the chain rule to words, we get

$$\begin{aligned} P(w_{1:n}) &= P(w_1)P(w_2|w_1)P(w_3|w_{1:2})\ldots P(w_n|w_{1:n-1}) \\ &= \prod_{k=1}^{n} P(w_k|w_{1:k-1}) \end{aligned}$$

The chain rule shows the link between computing the joint probability of a sequence and computing the conditional probability of a word given previous words. But using the chain rule doesn't really seem to help us! We don't know any way to compute the exact probability of a word given a long sequence of preceding words, $P(w_n|w_{1:n-1})$.

The intuition of the n-gram model is that instead of computing the probability of a word given its entire history, we can approximate the history by just the last few words. The bigram model, approximates the probability of a word given all the previous words $P(w_n|w_{1:n-1})$ by using only the conditional probability of the preceding word P(wn|wn-1). In other words, instead of computing the probability

$$P(\text{the}|\text{Walden Pond's water is so transparent that})$$

we approximate it with the probability $P(\text{the}|\text{that})$

When we use a bigram model to predict the conditional probability of the next word, we are thus making the following approximation:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-1})$$

The assumption that the probability of a word depends only on the previous word is Markov called a **Markov assumption.** Markov models are the class of probabilistic models that assume

we can predict the probability of some future unit without looking too far into the past. We can generalize the bigram (which looks one word into the past) to the trigram (which looks two words into the past) and thus to the n-gram (which looks n-1 words into the past).

Let's see a general equation for this n-gram approximation to the conditional probability of the next word in a sequence. We'll use N here to mean the n-gram size, so N = 2 means bigrams and N = 3 means trigrams. Then we approximate the probability of a word given its entire context as follows:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-N+1:n-1})$$

Given the bigram assumption for the probability of an individual word, we can compute the probability of a complete word sequence

$$P(w_{1:n}) \approx \prod_{k=1}^{n} P(w_k|w_{k-1})$$

An intuitive way to estimate probabilities is called maximum likelihood estimation or MLE. We get the MLE estimate for the parameters of an n-gram model by getting counts from a corpus, and normalizing the counts so that they lie between 0 and 1.

For example, to compute a particular bigram probability of a word w_n given a previous word w_{n-1}, we'll compute the count of the bigram $C(w_{n-1}w_n)$ and normalize by the sum of all the bigrams that share the same first word w_{n-1}:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})}$$

Let's work through an example using a mini-corpus of three sentences. We'll first need to augment each sentence with a special symbol <s> at the beginning of the sentence, to give us the bigram context of the first word. We'll also need a special end-symbol. </s>

<s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>

Here are the calculations for some of the bigram probabilities from this corpus.

$$P(\text{I}|\text{<s>}) = \tfrac{2}{3} = .67 \qquad P(\text{Sam}|\text{<s>}) = \tfrac{1}{3} = .33 \qquad P(\text{am}|\text{I}) = \tfrac{2}{3} = .67$$
$$P(\text{</s>}|\text{Sam}) = \tfrac{1}{2} = 0.5 \qquad P(\text{Sam}|\text{am}) = \tfrac{1}{2} = .5 \qquad P(\text{do}|\text{I}) = \tfrac{1}{3} = .33$$

Maximum Likelihood Estimate:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})}$$

The above equation estimates the n-gram probability by dividing the observed frequency of a particular sequence by the observed frequency of a prefix. This ratio is called a relative

frequency. We said above that this use of frequencies as a way to estimate probabilities is an example of maximum likelihood estimation or MLE. In MLE, the resulting parameter set maximizes the likelihood of the training set T given the model M (i.e., P(T|M)). For example, suppose the word Chinese occurs 400 times in a corpus of a million words like the Brown corpus. What is the probability that a random word selected from some other text of, say, a million words will be the word Chinese? The MLE of its probability is 400/1000000 or :0004. Now :0004 is not the best possible estimate of the probability of Chinese occurring in all situations; it might turn out that in some other corpus or context Chinese is a very unlikely word. But it is the probability that makes it most likely that Chinese will occur 400 times in a million-word corpus.

Let's move on to some examples from a slightly larger corpus than our 14-word example above. We'll use data from the now-defunct Berkeley Restaurant Project, a dialogue system from the last century that answered questions about a database of restaurants in Berkeley, California. Here are some text normalized sample user queries (a sample of 9332 sentences is on the website):

can you tell me about any good cantonese restaurants close by
mid priced thai food is what i'm looking for
tell me about chez panisse
can you give me a listing of the kinds of food that are available
i'm looking for a good place to eat breakfast
when is caffe venezia open during the day

Figure below shows the bigram counts from a piece of a bigram grammar from the Berkeley Restaurant Project. Note that the majority of the values are zero. In fact, we have chosen the sample words to cohere with each other; a matrix selected from a random set of eight words would be even more sparse.

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

Figure 3.1 Bigram counts for eight of the words (out of $V = 1446$) in the Berkeley Restaurant Project corpus of 9332 sentences. Zero counts are in gray.

Figure below shows the bigram probabilities after normalization (dividing each cell above Figure by the appropriate unigram for its row, taken from the following set of unigram probabilities):

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

Figure 3.2 Bigram probabilities for eight words in the Berkeley Restaurant Project corpus of 9332 sentences. Zero probabilities are in gray.

Here are a few other useful probabilities:

$P(\texttt{i}|\texttt{<s>}) = 0.25$ $P(\texttt{english}|\texttt{want}) = 0.0011$
$P(\texttt{food}|\texttt{english}) = 0.5$ $P(\texttt{</s>}|\texttt{food}) = 0.68$

Now we can compute the probability of sentences like I want English food or I want Chinese food by simply multiplying the appropriate bigram probabilities together, as follows:

$$\begin{aligned} P(\texttt{<s> i want english food </s>}) &= P(\texttt{i}|\texttt{<s>})P(\texttt{want}|\texttt{i})P(\texttt{english}|\texttt{want}) \\ &\quad P(\texttt{food}|\texttt{english})P(\texttt{</s>}|\texttt{food}) \\ &= .25 \times .33 \times .0011 \times 0.5 \times 0.68 \\ &= .000031 \end{aligned}$$

compute the probability of i want chinese food.

Some practical issues: Although for pedagogical purposes we have only described trigram bigram models, in practice it's more common to use trigram models, which condition on the previous two words rather than the previous word, or 4-gram or even 5-gram models, when there is sufficient training data. Note that for these larger ngrams, we'll need to assume extra contexts to the left and right of the sentence end.

For example, to compute trigram probabilities at the very beginning of the sentence, we use two pseudo-words for the first trigram (i.e., P(I|<s><s>).

We always represent and compute language model probabilities in log format as log probabilities. Since probabilities are (by definition) less than or equal to1, the more probabilities we multiply together, the smaller the product becomes. Multiplying enough n-

grams together would result in numerical underflow. By using log probabilities instead of raw probabilities, we get numbers that are not as small.

$$p_1 \times p_2 \times p_3 \times p_4 = \exp(\log p_1 + \log p_2 + \log p_3 + \log p_4)$$

Evaluating Language Models

The best way to evaluate the performance of a language model is to embed it in an application and measure how much the application improves. Such end-to-end evaluation is called extrinsic evaluation. Extrinsic evaluation is the only way to know if a particular improvement in a component is really going to help the task at hand. Thus, for speech recognition, we can compare the performance of two language models by running the speech recognizer twice, once with each language model, and seeing which gives the more accurate transcription. Unfortunately, running big NLP systems end-to-end is often very expensive. Instead, it would be nice to have a metric that can be used to quickly evaluate potential improvements in a language model. An intrinsic evaluation metric is one that measures the quality of a model independent of any application.

For an intrinsic evaluation of a language model we need a test set. As with many of the statistical models in our field, the probabilities of an n-gram model come from the corpus it is trained on, the training set or training corpus. We can then measure the quality of an n-gram model by its performance on some unseen data called the test set or test corpus. So if we are given a corpus of text and want to compare two different n-gram models, we divide the data into training and test sets, train the parameters of both models on the training set, and then compare how well the two trained models fit the test set. But what does it mean to "fit the test set"? The answer is simple: whichever model assigns a higher probability to the test set—meaning it more accurately predicts the test set, is a better model.

Perplexity

In practice we don't use raw probability as our metric for evaluating language models, but a variant called perplexity. The perplexity (sometimes called PPL for short) of a language model on a test set is the inverse probability of the test set, normalized by the number of words. For a test set W = w1w2…wN,:

$$\begin{aligned} \text{perplexity}(W) &= P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}} \end{aligned}$$

We can use the chain rule to expand the probability of W:

$$\text{perplexity}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots w_{i-1})}}$$

The perplexity of a test set W depends on which language model we use. Here's the perplexity of W with a unigram language model (just the geometric mean of the unigram probabilities):

$$\text{perplexity}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i)}}$$

The perplexity of W computed with a bigram language model is still a geometric mean, but now of the bigram probabilities:

$$\text{perplexity}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}}$$

Minimizing perplexity is equivalent to maximizing the test set probability according to the language model.

Given a text W, different language models will have different perplexities. Because of this, perplexity can be used to compare different n-gram models. Let's look at an example, in which we trained unigram, bigram, and trigram grammars on 38 million words (including start-of-sentence tokens) from the Wall Street Journal, using a 19,979 word vocabulary. We then computed the perplexity of each of these models on a test set of 1.5 million words, using Eq. for unigrams, for bigrams, and the corresponding equation for trigrams. The table below shows the perplexity of a 1.5 million word WSJ test set according to each of these grammars.

	Unigram	Bigram	Trigram
Perplexity	962	170	109

As we see above, the more information the n-gram gives us about the word sequence, the higher the probability the n-gram will assign to the string.

Sampling sentences from a language model

One important way to visualize what kind of knowledge a language model embodies is to sample from it. Sampling from a distribution means to choose random points according to their likelihood. Thus, sampling from a language model, which represents a distribution over sentences, means to generate some sentences, choosing each sentence according to its likelihood as defined by the model. Thus, we are more likely to generate sentences that the model thinks have a high probability and less likely to generate sentences that the model thinks have a low probability.

This technique of visualizing a language model by sampling was first suggested very early on by Shannon (1951) and Miller and Selfridge (1950). It's simplest to visualize how this works for the unigram case. Imagine all the words of the English language covering the probability space between 0 and 1, each word covering an interval proportional to its frequency. Figure shows a visualization, using a unigram LM computed from the text of this book. We choose a

random value between 0 and 1, find that point on the probability line, and print the word whose interval includes this chosen value. We continue choosing random numbers and generating words until we randomly generate the sentence-final token </s>.

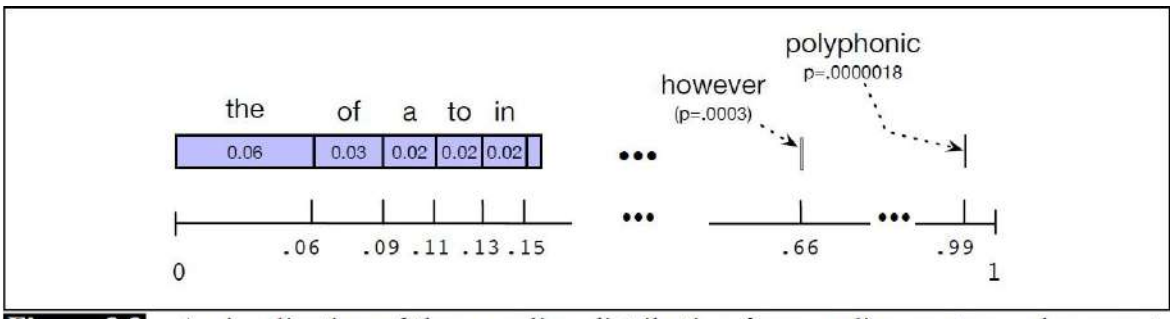


Figure 3.3 A visualization of the sampling distribution for sampling sentences by repeatedly sampling unigrams. The blue bar represents the relative frequency of each word (we've ordered them from most frequent to least frequent, but the choice of order is arbitrary). The number line shows the cumulative probabilities. If we choose a random number between 0 and 1, it will fall in an interval corresponding to some word. The expectation for the random number to fall in the larger intervals of one of the frequent words (*the*, *of*, *a*) is much higher than in the smaller interval of one of the rare words (*polyphonic*).

We can use the same technique to generate bigrams by first generating a random bigram that starts with <s> (according to its bigram probability). Let's say the second word of that bigram is w. We next choose a random bigram starting with w (again, drawn according to its bigram probability), and so on.

Generalization and Zeros

The n-gram model, like many statistical models, is dependent on the training corpus. One implication of this is that the probabilities often encode specific facts about a given training corpus. Another implication is that n-grams do a better and better job of modelling the training corpus as we increase the value of N. We can use the sampling method from the prior section to visualize both of these facts! To give an intuition for the increasing power of higher-order n-grams, Figure below shows random sentences generated from unigram, bigram, trigram, and 4-gram models trained on Shakespeare's works.

1 gram	–To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have –Hill he late speaks; or! a more to leg less first you enter
2 gram	–Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow. –What means, sir. I confess she? then all sorts, he is trim, captain.
3 gram	–Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done. –This shall forbid it should be branded, if renown made it empty.
4 gram	–King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in; –It cannot be but so.

Figure 3.4 Eight sentences randomly generated from four n-grams computed from Shakespeare's works. All characters were mapped to lower-case and punctuation marks were treated as words. Output is hand-corrected for capitalization to improve readability.

The longer the context on which we train the model, the more coherent the sentences. In the unigram sentences, there is no coherent relation between words or any sentence-final punctuation. The bigram sentences have some local word-to-word coherence (especially if we consider that punctuation counts as a word). The trigram and 4-gram sentences are beginning to look a lot like Shakespeare. Indeed, a careful investigation of the 4-gram sentences shows that they look a little too much like Shakespeare. The words ***It cannot be but so*** are directly from ***King John***. From Shakespeare (N =884,647, V=29,066), our n-gram probability matrices are ridiculously sparse. There are V2 =844,000,000 possible bigrams alone, and the number of possible 4-grams is V4 = $7X10^{17}$. Thus, once the generator has chosen the first 4-gram (It cannot be but), there are only five possible continuations (that, I, he, thou, and so); indeed, for many 4-grams, there is only one continuation.

To get an idea of the dependence of a grammar on its training set, let's look at an n-gram grammar trained on a completely different corpus: the Wall Street Journal (WSJ) newspaper. Shakespeare and the Wall Street Journal are both English, so we might expect some overlap between our n-grams for the two genres. Figure below shows sentences generated by unigram, bigram, and trigram grammars trained on 40 million words from WSJ.

Compare these examples to the pseudo-Shakespeare in above figure. While they both model "English-like sentences", there is clearly no overlap in generated sentences, and little overlap even in small phrases. Statistical models are likely to be pretty useless as predictors if the training sets and the test sets are as different as Shakespeare and WSJ.

1 gram	Months the my and issue of year foreign new exchange's september were recession exchange new endorsed a acquire to six executives
2 gram	Last December through the way to preserve the Hudson corporation N. B. E. C. Taylor would seem to complete the major central planners one point five percent of U. S. E. has already old M. X. corporation of living on information such as more frequently fishing to keep her
3 gram	They also point to ninety nine point six billion dollars from two hundred four oh six three percent of the rates of interest stores as Mexico and Brazil on market conditions

Figure 3.5 Three sentences randomly generated from three n-gram models computed from 40 million words of the *Wall Street Journal*, lower-casing all characters and treating punctuation as words. Output was then hand-corrected for capitalization to improve readability.

How should we deal with this problem when we build n-gram models? One step is to be sure to use a training corpus that has a similar genre to whatever task we are trying to accomplish. To build a language model for translating legal documents, we need a training corpus of legal documents. To build a language model for a question-answering system, we need a training corpus of questions. It is equally important to get training data in the appropriate dialect or variety, especially when processing social media posts or spoken transcripts.

Matching genres and dialects is still not sufficient. Our models may still be subject to the problem of sparsity. For any n-gram that occurred a sufficient number of times, we might have a good estimate of its probability. But because any corpus is limited, some perfectly acceptable English word sequences are bound to be missing from it. That is, we'll have many cases of putative "zero probability n-grams" that should really have some non-zero probability. Consider the words that follow the bigram **denied the** in the WSJ Treebank3 corpus, together with their counts:

denied the allegations: 5
denied the speculation: 2
denied the rumors: 1
denied the report: 1

But suppose our test set has phrases like:

denied the offer
denied the loan

Our model will incorrectly estimate that the $P(\text{offer}|\text{denied the})$ is 0!

These zeros—things that don't ever occur in the training set but do occur in the test set—are a problem for two reasons. First, their presence means we are underestimating the probability of all sorts of words that might occur, which will hurt the performance of any application we want to run on this data. Second, if the probability of any word in the test set is 0, the entire probability of the test set is 0. By definition, perplexity is based on the inverse probability of the test set. Thus, if some words have zero probability, we can't compute perplexity at all, since we can't divide by 0! There are two solutions, depending on the kind of zero. For words whose n-gram probability is zero because they occur in a novel test set context, like the example of **denied the** offer above, we'll introduce algorithms called smoothing or discounting. Smoothing algorithms shave off a bit of probability mass from some more frequent events and give it to these unseen events. But first, let's talk about an even more insidious form of zero: words that the model has never seen below at all (in any context): **unknown words!**

Unknown Words

What do we do about words we have never seen before? Perhaps the word **Jurafsky** simply did not occur in our training set, but pops up in the test set! We can choose to disallow this situation from occurring, by stipulating that we already know all the words that can occur. In such a closed vocabulary system the test set can only contain words from this known lexicon, and there will be no unknown words.

In most real situations, however, we have to deal with words we haven't seen before, which we'll call unknown words, or out of vocabulary (OOV) words. The percentage of OOV words that appear in the test set is called the OOV rate. One way to create an open vocabulary system

is to model these potential unknown words in the test set by adding a pseudo-word called <UNK>.

There are two common ways to train the probabilities of the unknown word model <UNK>. The first one is to turn the problem back into a closed vocabulary one by choosing a fixed vocabulary in advance:

1. Choose a vocabulary (word list) that is fixed in advance.
2. Convert in the training set any word that is not in this set (any OOV word) to the unknown word token <UNK> in a text normalization step.
3. Estimate the probabilities for <UNK> from its counts just like any other regular word in the training set.

The second alternative, in situations where we don't have a prior vocabulary in advance, is to create such a vocabulary implicitly, replacing words in the training data by <UNK> based on their frequency. For example, we can replace by <UNK> all words that occur fewer than n times in the training set, where n is some small number, or equivalently select a vocabulary size V in advance (say 50,000) and choose the top V words by frequency and replace the rest by <UNK>. In either case we then proceed to train the language model as before, treating <UNK> like a regular word.

Smoothing

What do we do with words that are in our vocabulary (they are not unknown words) but appear in a test set in an unseen context (for example they appear after a word they never appeared after in training)? To keep a language model from assigning zero probability to these unseen events, we'll have to shave off a bit of probability mass from some more frequent events and give it to the events we've never seen. This modification is called **smoothing** or **discounting.** Now we'll see a variety of ways to do smoothing: **Laplace (add-one) smoothing, add-k smoothing, stupid backoff, and Kneser-Ney smoothing.**

Laplace Smoothing

The simplest way to do smoothing is to add one to all the n-gram counts, before we normalize them into probabilities. All the counts that used to be zero will now have a count of 1, the counts of 1 will be 2, and so on. This algorithm is called Laplace smoothing. Laplace smoothing does not perform well enough to be used smoothing in modern n-gram models, but it usefully introduces many of the concepts that we see in other smoothing algorithms, gives a useful baseline, and is also a practical smoothing algorithm for other tasks like text classification. Let's start with the application of Laplace smoothing to unigram probabilities. Recall that the unsmoothed maximum likelihood estimate of the unigram probability of the word wi is its count ci normalized by the total number of word tokens N:

$$P(w_i) = \frac{c_i}{N}$$

Laplace smoothing merely adds one to each count (hence its alternate name **add one smoothing**). Since there are V words in the vocabulary and each one was incremented, we also need to adjust the denominator to take into account the extra V observations.

$$P_{\text{Laplace}}(w_i) = \frac{c_i + 1}{N + V}$$

Let's smooth our Berkeley Restaurant Project bigrams. Figure below shows the add-one smoothed counts for the bigrams in Berkeley Restaurant Project.

	i	want	to	eat	chinese	food	lunch	spend
i	6	828	1	10	1	1	1	3
want	3	1	609	2	7	7	6	2
to	3	1	5	687	3	1	7	212
eat	1	1	3	1	17	3	43	1
chinese	2	1	1	1	1	83	2	1
food	16	1	16	1	2	5	1	1
lunch	3	1	1	1	1	2	1	1
spend	2	1	2	1	1	1	1	1

Figure 3.6 Add-one smoothed bigram counts for eight of the words (out of $V = 1446$) in the Berkeley Restaurant Project corpus of 9332 sentences. Previously-zero counts are in gray.

Recall that normal bigram probabilities are computed by normalizing each row of counts by the unigram count:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})}$$

For add-one smoothed bigram counts, we need to augment the unigram count by the number of total word types in the vocabulary V:

$$P_{\text{Laplace}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n) + 1}{\sum_w (C(w_{n-1}w) + 1)} = \frac{C(w_{n-1}w_n) + 1}{C(w_{n-1}) + V}$$

Thus, each of the unigram counts given in the previous section will need to be augmented by V =1446. The result is the smoothed bigram probabilities in Figure below.

	i	want	to	eat	chinese	food	lunch	spend
i	0.0015	0.21	0.00025	0.0025	0.00025	0.00025	0.00025	0.00075
want	0.0013	0.00042	0.26	0.00084	0.0029	0.0029	0.0025	0.00084
to	0.00078	0.00026	0.0013	0.18	0.00078	0.00026	0.0018	0.055
eat	0.00046	0.00046	0.0014	0.00046	0.0078	0.0014	0.02	0.00046
chinese	0.0012	0.00062	0.00062	0.00062	0.00062	0.052	0.0012	0.00062
food	0.0063	0.00039	0.0063	0.00039	0.00079	0.002	0.00039	0.00039
lunch	0.0017	0.00056	0.00056	0.00056	0.00056	0.0011	0.00056	0.00056
spend	0.0012	0.00058	0.0012	0.00058	0.00058	0.00058	0.00058	0.00058

Figure 3.7 Add-one smoothed bigram probabilities for eight of the words (out of $V = 1446$) in the BeRP corpus of 9332 sentences. Previously-zero probabilities are in gray.

	i	want	to	eat	chinese	food	lunch	spend
i	3.8	527	0.64	6.4	0.64	0.64	0.64	1.9
want	1.2	0.39	238	0.78	2.7	2.7	2.3	0.78
to	1.9	0.63	3.1	430	1.9	0.63	4.4	133
eat	0.34	0.34	1	0.34	5.8	1	15	0.34
chinese	0.2	0.098	0.098	0.098	0.098	8.2	0.2	0.098
food	6.9	0.43	6.9	0.43	0.86	2.2	0.43	0.43
lunch	0.57	0.19	0.19	0.19	0.19	0.38	0.19	0.19
spend	0.32	0.16	0.32	0.16	0.16	0.16	0.16	0.16

Figure 3.8 Add-one reconstituted counts for eight words (of $V = 1446$) in the BeRP corpus of 9332 sentences. Previously-zero counts are in gray.

The sharp change in counts and probabilities occurs because too much probability mass is moved to all the zeros.

Add-k smoothing

One alternative to add-one smoothing is to move a bit less of the probability mass from the seen to the unseen events. Instead of adding 1 to each count, we add a fractional count k (.5? .05? .01?). This algorithm is therefore called add-k smoothing.

$$P^*_{\text{Add-k}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n) + k}{C(w_{n-1}) + kV}$$

Add-k smoothing requires that we have a method for choosing k; this can be done, for example, by optimizing on a devset. Although add-k is useful for some tasks (including text classification), it turns out that it still doesn't work well for language modelling, generating counts with poor variances and often inappropriate discounts.

Backoff and Interpolation

The discounting we have been discussing so far can help solve the problem of zero frequency n-grams. But there is an additional source of knowledge we can draw on. If we are trying to compute $P(w_n|w_{n-2}w_{n-1})$ but we have no examples of a particular trigram $w_{n-2}w_{n-1}w_n$, we can instead estimate its probability by using the bigram probability $P(w_n|w_{n-1})$. Similarly, if we don't have counts to compute $P(w_n|w_{n-1})$, we can look to the unigram $P(w_n)$. In other words, sometimes using less context is a good thing, helping to generalize more for contexts that the model hasn't learned much about. There are two ways to use this n-gram "hierarchy". In backoff, we use the trigram if the evidence is sufficient, otherwise we use the bigram, otherwise the unigram. In other words, we only "back off" to a lower-order n-gram if we have zero evidence for a higher-order n-gram.

By contrast, in interpolation, we always mix the probability estimates from all the n-gram estimators, weighting and combining the trigram, bigram, and unigram counts. In simple linear interpolation, we combine different order n-grams by linearly interpolating them. Thus, we

estimate the trigram probability $P(w_n|w_{n-2}w_{n-1})$ by mixing together the unigram, bigram, and trigram probabilities, each weighted by a

λ:

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) &= \lambda_1 P(w_n) \\ &\quad +\lambda_2 P(w_n|w_{n-1}) \\ &\quad +\lambda_3 P(w_n|w_{n-2}w_{n-1})\end{aligned}$$

The λs must sum to 1, making Equation equivalent to a weighted average:

$$\sum_i \lambda_i = 1$$

Word Embedding techniques: Bag of words (BOW)

Natural Language Processing technique of text modeling known as Bag of Words model. Whenever we apply any algorithm in NLP, it works on numbers. We cannot directly feed our text into that algorithm. Hence, Bag of Words model is used to preprocess the text by converting it into a bag of words, which keeps a count of the total occurrences of most frequently used words. This model can be visualized using a table, which contains the count of words corresponding to the word itself. Applying the Bag of Words model: Let us take this sample paragraph for our task

Beans. I was trying to explain to somebody as we were flying in, that's corn. That's beans. And they were very impressed at my agricultural knowledge. Please give it up for Amaury once again for that outstanding introduction. I have a bunch of good friends here today, including somebody who I served with, who is one of the finest senators in the country, and we're lucky to have him, your Senator, Dick Durbin is here. I also noticed, by the way, former Governor Edgar here, who I haven't seen in a long time, and somehow he has not aged and I have. And it's great to see you, Governor. I want to thank President Killeen and everybody at the U of I System for making it possible for me to be here today. And I am deeply honored at the Paul Douglas Award that is being given to me. He is somebody who set the path for so much outstanding public service here in Illinois. Now, I want to start by addressing the elephant in the room. I know people are still wondering why I didn't speak at the commencement.

Step #1 : We will first preprocess the data, in order to:

- Convert text to lower case.
- Remove all non-word characters.
- Remove all punctuations.

```
# Python3 code for preprocessing text
import nltk
import re
import numpy as np

# execute the text here as :
# text = """ # place text here  """
dataset = nltk.sent_tokenize(text)
for i in range(len(dataset)):
    dataset[i] = dataset[i].lower()
    dataset[i] = re.sub(r'\W', ' ', dataset[i])
    dataset[i] = re.sub(r'\s+', ' ', dataset[i])
```

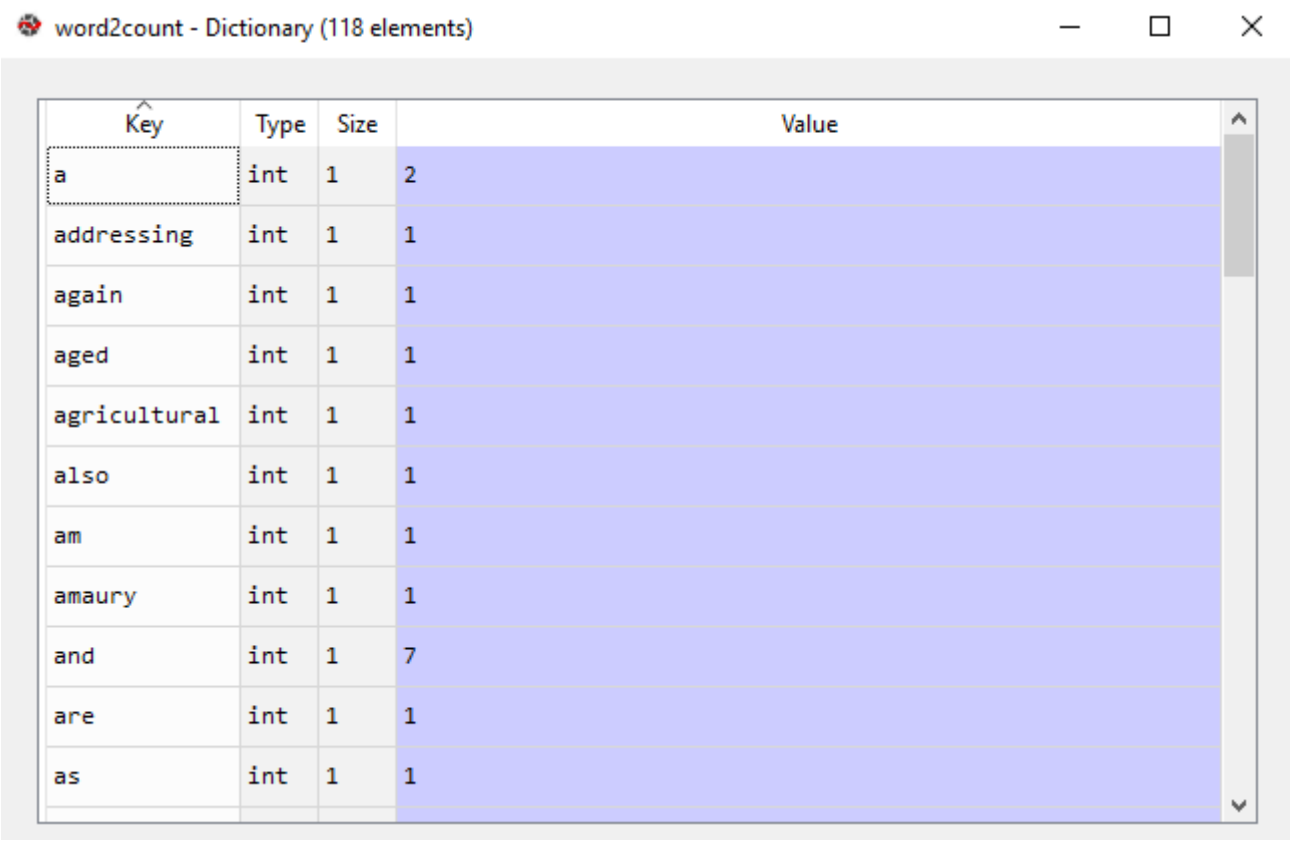
Index	Type	Size	Value
0	str	1	beans
1	str	1	i was trying to explain to somebody as we were flying in that s corn
2	str	1	that s beans
3	str	1	and they were very impressed at my agricultural knowledge
4	str	1	please give it up for amaury once again for that outstanding introduct ...
5	str	1	i have a bunch of good friends here today including somebody who i ser ...
6	str	1	i also noticed by the way former governor edgar here who i haven t see ...
7	str	1	and it s great to see you governor
8	str	1	i want to thank president killeen and everybody at the u of i system f ...
9	str	1	and i am deeply honored at the paul douglas award that is being given ...
10	str	1	he is somebody who set the path for so much outstanding public service ...
11	str	1	now i want to start by addressing the elephant in the room
12	str	1	i know people are still wondering why i didn t speak at the commenceme ...

You can further preprocess the text to suit you needs.

Step #2 : Obtaining most frequent words in our text. We will apply the following steps to generate our model.

- we declare a dictionary to hold our bag of words.
- Next we tokenize each sentence to words.
- Now for each word in sentence, we check if the word exists in our dictionary.
- If it does, then we increment its count by 1. If it doesn't, we add it to our dictionary and set its count as 1.

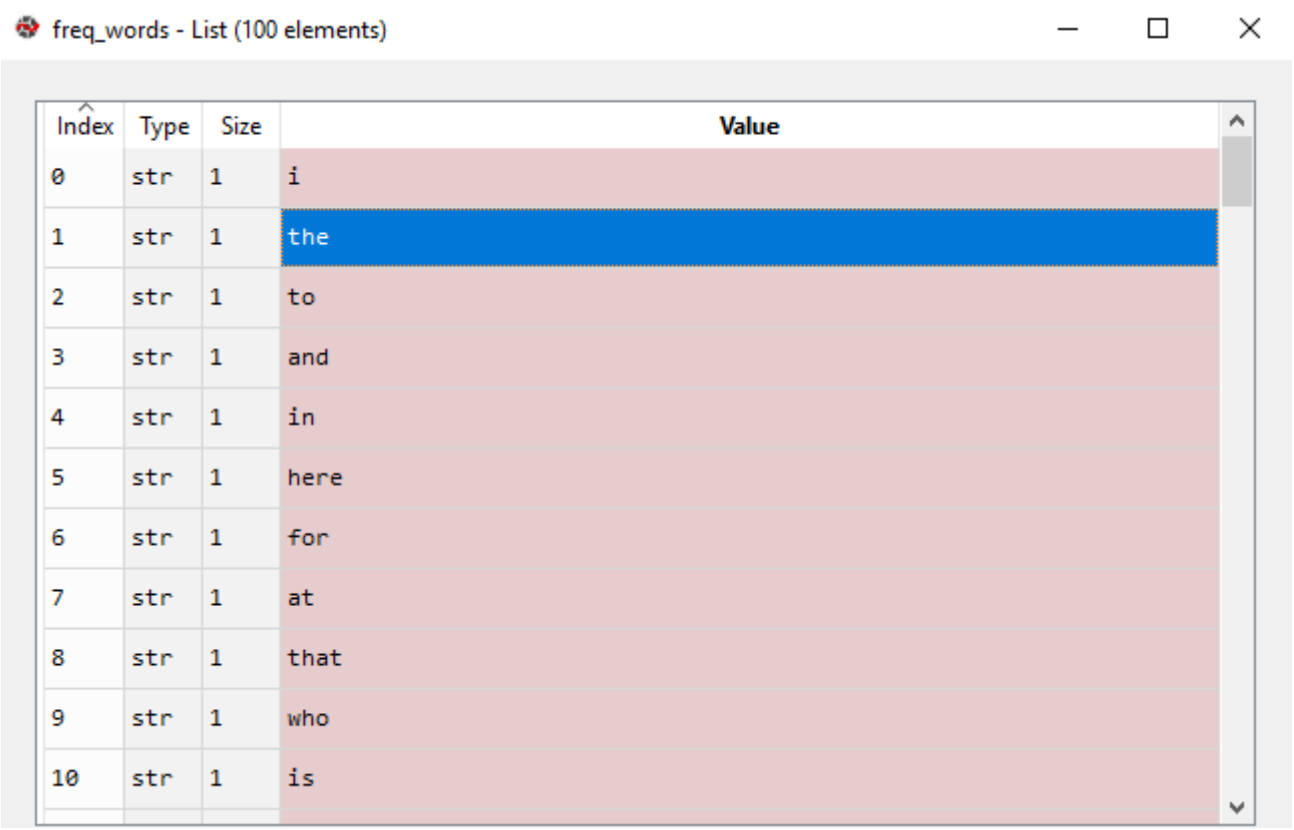
```
# Creating the Bag of Words model
word2count = {}
for data in dataset:
    words = nltk.word_tokenize(data)
    for word in words:
        if word not in word2count.keys():
            word2count[word] = 1
        else:
            word2count[word] += 1
```



In our model, we have a total of 118 words. However when processing large texts, the number of words could reach millions. We do not need to use all those words. Hence, we select a particular number of most frequently used words. To implement this we use:

```
import heapq
freq_words = heapq.nlargest(100, word2count, key=word2count.get)
```

where 100 denotes the number of words we want. If our text is large, we feed in a larger number.



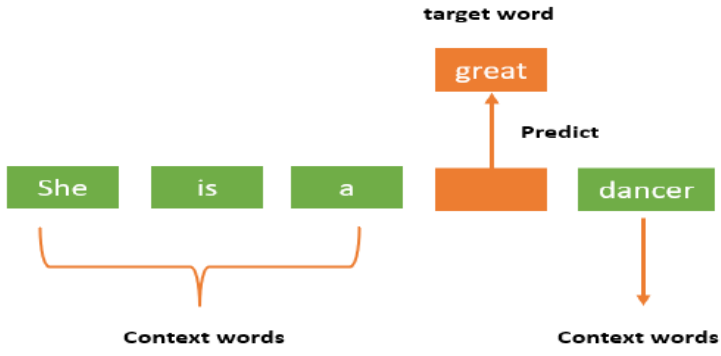
Step #3 : Building the Bag of Words model In this step we construct a vector, which would tell us whether a word in each sentence is a frequent word or not. If a word in a sentence is a frequent word, we set it as 1, else we set it as 0. This can be implemented with the help of following code:

```
X = []
for data in dataset:
    vector = []
    for word in freq_words:
        if word in nltk.word_tokenize(data):
            vector.append(1)
        else:
            vector.append(0)
    X.append(vector)
X = np.asarray(X)
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	1	0	0	0	1	0	0	0	1
2	0	0	0	0	0	0	0	0	1	0	0	0	0
3	0	0	0	1	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	1	0	1	0	0	0	0
5	1	1	1	1	1	1	0	0	0	1	1	1	1
6	1	1	0	1	1	1	0	0	0	1	0	1	0
7	0	0	1	1	0	0	0	0	0	0	0	0	0
8	1	1	1	1	0	1	1	1	0	0	0	0	0
9	1	1	1	1	0	0	0	1	1	0	1	0	0
10	0	1	0	0	1	1	1	0	0	1	1	0	1
11	1	1	1	0	1	0	0	0	0	0	0	0	0
12	1	1	0	0	0	0	0	1	0	0	0	0	0

Continuous Bag of Words (CBOW)

Continuous Bag of Words (CBOW) is a popular natural language processing technique used to generate word embeddings. Word embedding's are important for many NLP tasks because they capture semantic and syntactic relationships between words in a language. CBOW is a neural network-based algorithm that predicts a target word given its surrounding context words. It is a type of "unsupervised" learning, meaning that it can learn from unlabeled data, and it is often used to pre-train word embeddings that can be used for various NLP tasks such as sentiment analysis, text classification, and machine translation.

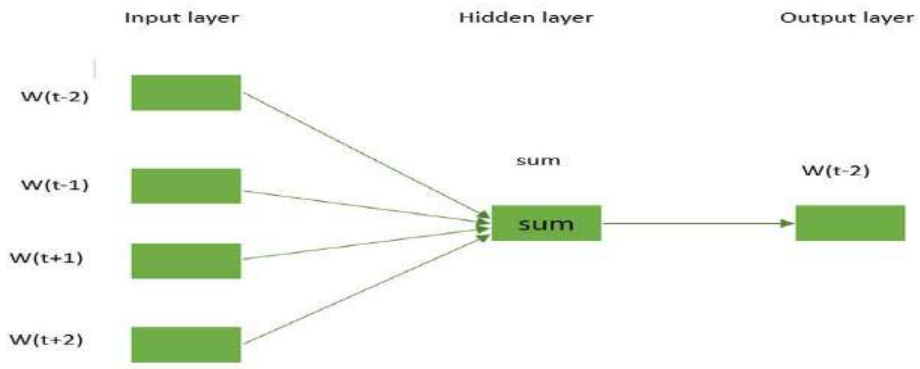


Is there any difference between Bag-of-Words (BoW) model and the Continuous Bag-of-Words (CBOW)?

- The Bag-of-Words model and the Continuous Bag-of-Words model are both techniques used in natural language processing to represent text in a computer-readable format, but they differ in how they capture context.
- The BoW model represents text as a collection of words and their frequency in a given document or corpus. It does not consider the order or context in which the words appear, and therefore, it may not capture the full meaning of the text. The BoW model is simple and easy to implement, but it has limitations in capturing the meaning of language.
- In contrast, the CBOW model is a neural network-based approach that captures the context of words. It learns to predict the target word based on the words that appear before and after it in a given context window. By considering the surrounding words, the CBOW model can better capture the meaning of a word in a given context.

Architecture of the CBOW model

The CBOW model uses the context words around the target word in order to predict it. Consider the above example **"She is a great dancer."** The CBOW model converts this phrase into pairs of context words and target words. The word pairings would appear like this **([she, a], is), ([is, great], a) ([a, dancer], great)** having window size=2.



The model considers the context words and tries to predict the target term. The four 1∗W input vectors will be passed to the input layer if have four words as context words are used to predict one target word. The hidden layer will receive the input vectors and then multiply them by a W∗N matrix. The 1∗N output from the hidden layer finally enters the sum layer, where the vectors are element-wise summed before a final activation is carried out and the output is obtained from the output layer.

Code Implementation of CBOW

Let's implement a word embedding to show the similarity of words using the CBOW model. In this article I have defined my own corpus of words, you use any dataset. First, we will import all the necessary libraries and load the dataset. Next, we will tokenize each word and convert it into a vector of integers.

```
# Re-import necessary modules
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Embedding, Lambda
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.utils import to_categorical
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Define the corpus
corpus = [
    'The cat sat on the mat',
    'The dog ran in the park',
    'The bird sang in the tree'
]

# Convert the corpus to a sequence of integers
tokenizer = Tokenizer()
tokenizer.fit_on_texts(corpus)
sequences = tokenizer.texts_to_sequences(corpus)
print("After converting our words in the corpus into vector of integers:")
print(sequences)
```

Output:

```
After converting our words in the corpus into vector of integers:
[[1, 3, 4, 5, 1, 6], [1, 7, 8, 2, 1, 9], [1, 10, 11, 2, 1, 12]]
```

Now, we will build the CBOW model having window size = 2.

```
# Define the parameters
vocab_size = len(tokenizer.word_index) + 1
embedding_size = 10
window_size = 2

# Generate the context-target pairs
contexts = []
targets = []
for sequence in sequences:
    for i in range(window_size, len(sequence) - window_size):
        context = sequence[i - window_size:i] + sequence[i + 1:i + window_size + 1]
        target = sequence[i]
        contexts.append(context)
        targets.append(target)

# Convert the contexts and targets to numpy arrays
X = np.array(contexts)
y = to_categorical(targets, num_classes=vocab_size)

# Define the CBOW model
model = Sequential()
model.add(Embedding(input_dim=vocab_size, output_dim=embedding_size, input_length=2 *
window_size))
model.add(Lambda(lambda x: tf.reduce_mean(x, axis=1)))
model.add(Dense(units=vocab_size, activation='softmax'))

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(X, y, epochs=100, verbose=0)
```

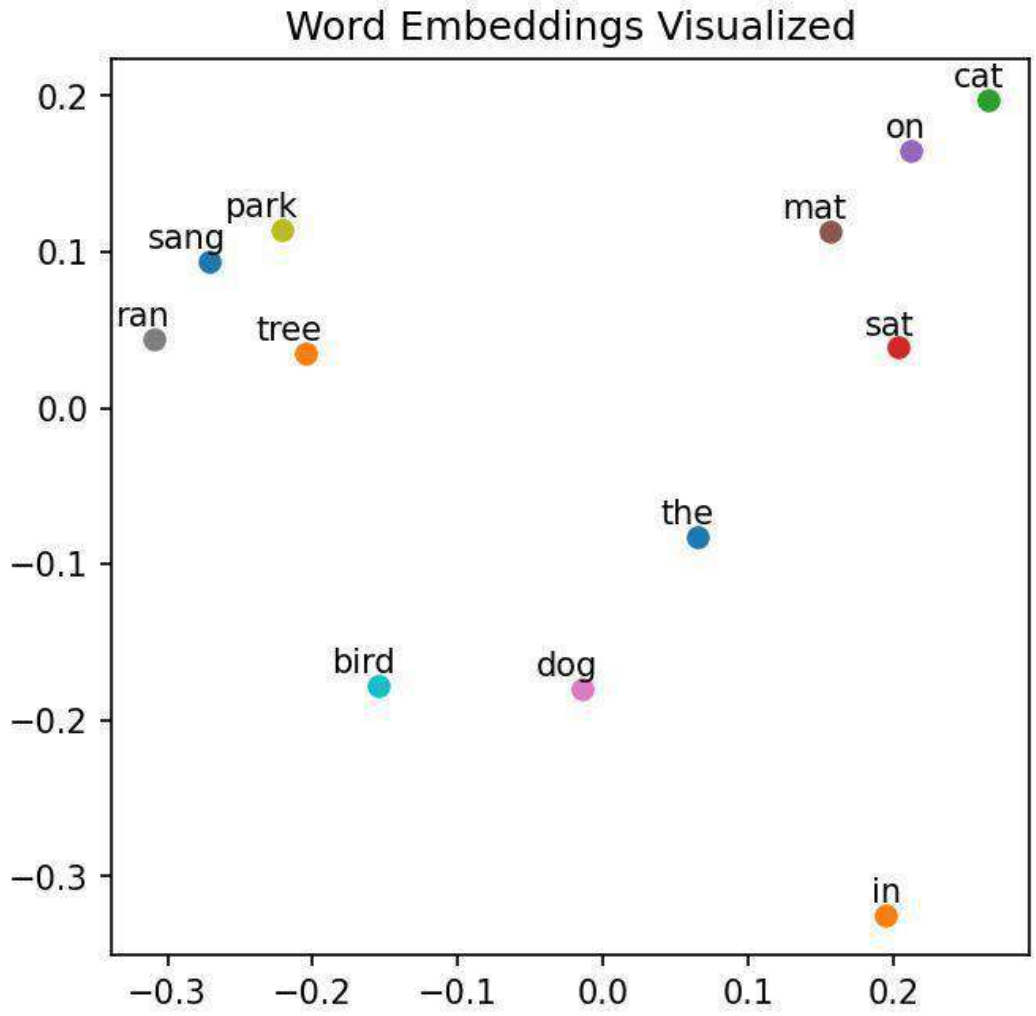
Next, we will use the model to visualize the embeddings.

```
# Extract the embeddings
embedding_layer = model.layers[0]
embeddings = embedding_layer.get_weights()[0]

# Perform PCA to reduce the dimensionality of the embeddings
pca = PCA(n_components=2)
reduced_embeddings = pca.fit_transform(embeddings)

# Visualize the embeddings
plt.figure(figsize=(5, 5))
for word, idx in tokenizer.word_index.items():
```

```
    x, y = reduced_embeddings[idx]
    plt.scatter(x, y)
    plt.annotate(word, xy=(x, y), xytext=(5, 2),
                 textcoords='offset points', ha='right', va='bottom')
plt.title("Word Embeddings Visualized")
plt.show()
```



This visualization allows us to observe the similarity of the words based on their embeddings. Words that are similar in meaning or context are expected to be close to each other in the plot.

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

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

- Term Frequency - Inverse Document Frequency (TF-IDF) is a widely used statistical method in natural language processing and information retrieval.
- It measures how important a term is within a document relative to a collection of documents (i.e., relative to a corpus).
- Words within a text document are transformed into important numbers by a text vectorization process.
- There are many different text vectorization scoring schemes, with TF-IDF being one of the most common.

As its name implies, TF-IDF vectorizes/scores a word by multiplying the word's Term Frequency (TF) with the Inverse Document Frequency (IDF).

Term Frequency: TF of a term or word is the number of times the term appears in a document compared to the total number of words in the document.

$$TF = \frac{\text{number of times the term appears in the document}}{\text{total number of terms in the document}}$$

Inverse Document Frequency: IDF of a term reflects the proportion of documents in the corpus that contain the term. Words unique to a small percentage of documents (e.g., technical jargon terms) receive higher importance values than words common across all documents (e.g., a, the, and).

$$IDF = log(\frac{\text{number of the documents in the corpus}}{\text{number of documents in the corpus contain the term}})$$

The TF-IDF of a term is calculated by multiplying TF and IDF scores.

$$TF\text{-}IDF = TF * IDF$$

Imagine the term t appears 20 times in a document that contains a total of 100 words. Term Frequency (TF) of t can be calculated as follow:

$$TF = \frac{20}{100} = 0.2$$

Assume a collection of related documents contains 10,000 documents. If 100 documents out of 10,000 documents contain the term t, Inverse Document Frequency (IDF) of t can be calculated as follows

$$IDF = log\frac{10000}{100} = 2$$

Using these two quantities, we can calculate TF-IDF score of the term t for the document.

$$TF\text{-}IDF = 0.2 * 2 = 0.4$$

Imagine the term t appears 20 times in a document that contains a total of 100 words. Term Frequency (TF) of t can be calculated as follow:

$$TF = \frac{20}{100} = 0.2$$

Assume a collection of related documents contains 10,000 documents. If 100 documents out of 10,000 documents contain the term t, Inverse Document Frequency (IDF) of t can be calculated as follows

$$IDF = log\frac{10000}{100} = 2$$

Using these two quantities, we can calculate TF-IDF score of the term t for the document.

$$TF\text{-}IDF = 0.2 * 2 = 0.4$$

Python Implementation

Some popular python libraries have a function to calculate TF-IDF. The popular machine learning library `Sklearn` has `TfidfVectorizer()` function (docs).

We will write a TF-IDF function from scratch using the standard formula given above, but we will not apply any preprocessing operations such as stop words removal, stemming, punctuation removal, or lowercasing. It should be noted that the result may be different when using a native function built into a library.

```
import pandas as pd
import numpy as np
```

First, let's construct a small corpus.

```
corpus = ['data science is one of the most important fields of science',
          'this is one of the best data science courses',
          'data scientists analyze data' ]
```

Next, we'll create a word set for the corpus:

```
words_set = set()

for doc in  corpus:
    words = doc.split(' ')
    words_set = words_set.union(set(words))

print('Number of words in the corpus:',len(words_set))
print('The words in the corpus: \n', words_set)
```

```
Number of words in the corpus: 14
The words in the corpus:
 {'important', 'scientists', 'best', 'courses', 'this', 'analyze', 'of', 'most', 'the',
```

```
ourses', 'this', 'analyze', 'of', 'most', 'the', 'is', 'science', 'fields', 'one', 'data'}
```

Computing Term Frequency

Now we can create a dataframe by the number of documents in the corpus and the word set, and use that information to compute the **term frequency (TF)**:

```
n_docs = len(corpus)          # Number of documents in the corpus
n_words_set = len(words_set) # Number of unique words in the

df_tf = pd.DataFrame(np.zeros((n_docs, n_words_set)), columns=words_set)

# Compute Term Frequency (TF)
for i in range(n_docs):
    words = corpus[i].split(' ') # Words in the document
    for w in words:
        df_tf[w][i] = df_tf[w][i] + (1 / len(words))

df_tf
```

	important	scientists	best	courses	this	analyze	of
0	0.090909	0.00	0.000000	0.000000	0.000000	0.00	0.181818
1	0.000000	0.00	0.111111	0.111111	0.111111	0.00	0.111111
2	0.000000	0.25	0.000000	0.000000	0.000000	0.25	0.00000

The dataframe above shows we have a column for each word and a row for each document. This shows the frequency of each word in each document.

most	the	is	science	fields	one	data
0.090909	0.090909	0.090909	0.181818	0.090909	0.090909	0.090
0.000000	0.111111	0.111111	0.111111	0.000000	0.111111	0.11111
0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.500

Computing Inverse Document Frequency

Now, we'll compute the **inverse document frequency (IDF)**:

```
print("IDF of: ")

idf = {}

for w in words_set:
    k = 0    # number of documents in the corpus that contain this word

    for i in range(n_docs):
        if w in corpus[i].split():
            k += 1

    idf[w] =  np.log10(n_docs / k)

    print(f'{w:>15}: {idf[w]:>10}' )
```

Putting it Together: Computing TF-IDF

Since we have TF and IDF now, we can compute **TF-IDF**:

```
df_tf_idf = df_tf.copy()

for w in words_set:
    for i in range(n_docs):
        df_tf_idf[w][i] = df_tf[w][i] * idf[w]

df_tf_idf
```

	important	scientists	best	courses	this	analyze	of
0	0.043375	0.00000	0.000000	0.000000	0.000000	0.00000	0.0
1	0.000000	0.00000	0.053013	0.053013	0.053013	0.00000	0.0
2	0.000000	0.11928	0.000000	0.000000	0.000000	0.11928	0.0

Notice that "data" has an IDF of 0 because it appears in every document. As a result, is not considered to be an important term in this corpus. This will change slightly in the following sklearn implementation, where "data" will be non-zero.

UNIT - IV

Semantic Parsing

1. Introduction

- Two approaches have emerged in the NLP for language understanding.
- In the first approach, a specific, rich meaning representation is created for a limited domain for use by application that are restricted to that domain, such as travel reservations, football game simulations, or querying a geographic database.
- In the second approach, a related set of intermediate-specific meaning representation is created, going from low-level analysis to a middle analysis, and the bigger understanding task is divided into multiple, smaller pieces that are more manageable, such as word sense disambiguation followed by predicate-argument structure recognition.
- Here two types of meaning representations: a domain-dependent, deeper representation and a set of relatively shallow but general-purpose, low-level, and intermediate representation.
- The task of producing the output of the first type is often called deep semantic parsing, and the task of producing the output of the second type is often called shallow semantic parsing.
- The first approach is so specific that porting to every new domain can require anywhere from a few modifications to almost reworking the solution from scratch.
- In other words, the reusability of the representation across domains is very limited.
- The problem with second approach is that it is extremely difficult to construct a general-purpose ontology and create symbols that are shallow enough to be learnable but detailed enough to be useful for all possible applications.
- Ontology means
  1. The branch of metaphysics dealing with the nature of being.
  2. a set of concepts and categories in a subject area or domain that shows their properties and the relations between them.

  "what's new about our ontology is that it is created automatically from large datasets"
- Therefore, an application specific translation layer between the more general representation and the more specific representation becomes necessary.

2. Semantic Interpretation

Semantic parsing can be considered as part of Semantic interpretation, which involves various components that together define a representation of text that can be fed into a computer to allow further computations manipulations and search, which are prerequisite for any language understanding system or application. Here we discuss the structure of semantic theory.

A Semantic theory should be able to:

- Explain sentence having ambiguous meaning: The bill is large is ambiguous in the sense that is could represent money or the beak of a bird.

- Resolve the ambiguities of words in context. The bill is large but need not be paid, the theory should be able to disambiguate the monetary meaning of bill.
- Identify meaningless but syntactically well-formed sentence: Colorless green ideas sleep furiously.
- Identify syntactically or transformationally unrelated paraphrasers of concept having the same semantic content.

- Here we look at some requirements for achieving a semantic representation.

2.1 Structural Ambiguity

- Structure means syntactic structure of sentences.
- The syntactic structure means transforming a sentence into its underlying syntactic representation and in theory of semantic interpretation refer to underlying syntactic representation.

2.2 Word Sense

- In any given language, the same word type is used in different contexts and with different morphological variants to represent different entities or concepts in the world.
- For example, we use the word **nail** to represent a part of the human anatomy and also to represent the generally metallic object used to secure other objects.

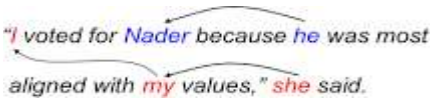
intended by the author or speaker. Let's take the following four examples. The presence of words such as *hammer* and *hardware store* in sentences 1 and 2, and of *clipped* and *manicure* in sentences 3 and 4, enable humans to easily disambiguate the sense in which *nail* is used:

1. He *nailed* the loose arm of the chair with a hammer.
2. He bought a box of *nails* from the hardware store.
3. He went to the beauty salon to get his *nails* clipped.
4. He went to get a manicure. His *nails* had grown very long.

Resolving the sense of words in a discourse, therefore, constitutes one of the steps in the process of semantic interpretation. We discuss it in greater depth in Section 4.4.

2.3 Entity and Event Resolution

- Any discourse consists of a set of entities participating in a series of explicit or implicit events over a period of time.
- So, the next important component of semantic interpretation is the identification of various entities that are sparkled across the discourse using the same or different phrases.
- The predominant tasks have become popular over the years: **named entity recognition** and **coreference resolution**.
- Coreference resolution is the task of finding all expressions that refer to the same entity in a text.



2.4 Predicate Argument Structure

- Once we have the word-sense, entities and events identified, another level of semantics structure comes into play: identifying the participants of the entities in these events.
- Resolving the argument structure of predicate in the sentence is where we identify which entities play what part in which event.
- A word which functions as the verb is called a **predicate** and words which function as the nouns arecalled **arguments**. Here are some other predicates and arguments:

Selena	slept		
argument	predicate		
Tom	is tall		
argument	predicate		
Percy	placed	the penguin on the podium	
argument	predicate	argument	argument

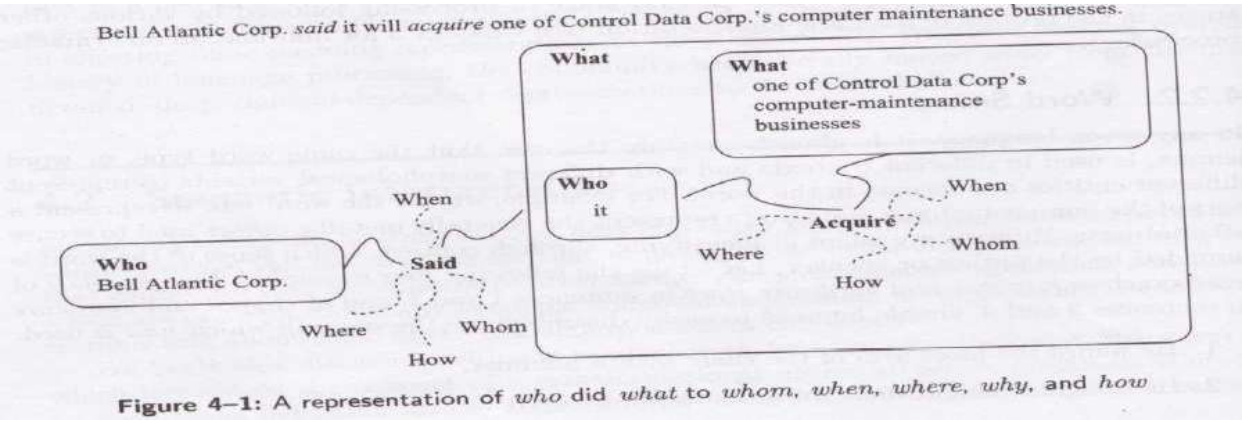


Figure 4–1: A representation of *who* did *what* to *whom*, *when*, *where*, *why*, and *how*

2.5 Meaning Representation

- The final process of the semantic interpretation is to build a semantic representation or meaning representation that can then be manipulated by algorithms to various application ends.
- This process is sometimes called the deep representation.

The following two examples

(1) If our player 2 has the ball, then position our player 5 in the midfield.
`((bowner (player our 2)) (do (player our 5) (pos (midfield)))`

(2) Which river is the longest?
`answer(`x_1`, longest(`x_1 `river(`x_1`)))`

3. System Paradigms

- It is important to get a perspective on the various primary dimensions on which the problem of semantic interpretation has been tackled.
- The approaches generally fall into the following three categories: 1.System architecture

2. Scope 3. Coverage**.**

1. System Architectures

a. **Knowledge based**: These systems use a predefined set of rules or a knowledge base to obtain a solution to a new problem.

b. **Unsupervised**: Thesesystemstendto requireminimal humanintervention to be functional by using existing resources that can be bootstrapped for a particular application or problem domain.

c. **Supervised**: these systems involve the manual annotation of some phenomena that appear in a sufficient quantity of data so that machine learning algorithms can be applied.

d. **Semi-Supervised**: manual annotation is usually very expensive and does not yield enough data to completely capture a phenomenon. In such instances, researches can automatically expand the data set on which their models are trained either by employing machine-generated output directly or by bootstrapping off an existing model by having humans correct its output.

2. Scope:

- **Domain Dependent**: These systems are specific to certain domains, such as air travel reservations or simulated football coaching.
- **Domain Independent**: These systems are general enough that the techniques can be applicable to multiple domains without little or no change.

3. Coverage

a. **Shallow**: These systems tend to produce an intermediate representation that can then be converted to one that a machine can base its action on.

b. **Deep**: These systems usually create a terminal representation that is directly consumed by a machine or application.

4. Word Sense

- **Word Sense** Disambiguation is an important method of **NLP** by which the meaning of a word is determined, which is used in a particular context.
- In a compositional approach to semantics, where the meaning of the whole is composed on the meaning of parts, the smallest parts under consideration in textual discourse are typically the words themselves: either tokens as they appear in the text or their lemmatized forms.
- Words sense has been examined and studied for a very long time.
- Attempts to solve this problem range from rule based and knowledge based to completely unsupervised, supervised, and semi-supervised learning methods.
- Very early systems were predominantly **rule based or knowledge based** and used dictionary definitions of senses of words.
- **Unsupervised** word sense induction or disambiguation techniques try to induce the senses or word as it appears in various corpora.
- These systems perform either a hard or soft clustering of words and tend to allow the tuning of these clusters to suit a particular application.
- Most recent **supervised** approaches to word sense disambiguation, usually application- independent-level of granularity (including small details). Although the output of supervised approaches can still be amendable to generating a ranking,

or distribution, of membership sense.

- Word sense ambiguities can be of three principal types: i.**homonymy** ii.**polysemy** iii.**categorial ambiguity**.
- **Homonymy** defined as the words having same spelling or same form but having different and unrelated meaning. For example, the word "Bat" is a homonymy word because bat can be an implement to hit a ball or bat is a nocturnal flying mammal also
- **Polysemy** is a Greek word, which means "many signs". polysemy has the same spelling but different and related meaning.
- Both polysemy and homonymy words have the same syntax or spelling. The main difference between them is that in polysemy, the meanings of the words are related but in homonymy, the meanings of the words are not related.
- For example: Bank Homonymy: financial bank and river bank
  Polysemy: financial bank, bank of clouds and book bank: indicate collection of things.
- **Categorial ambiguity**: the word book can mean a book which contain the chapters or police register which is used to enter the charges against someone.
- In the above note book, text book belongs to the grammatical category of noun and book is verb.
- Distinguishing between these two categories effectively helps disambiguate these two senses.
- Therefore, categorical ambiguity can be resolved with syntactic information (part of speech) alone, but polyseme and homonymy need more than syntax.
- Traditionally, in English, word senses have been annotated for each part of speech separately, whereas in Chinese, the sense annotation has been done per lemma.

Resources:

- As with any language understanding task, the availability of resources is key factor in the disambiguation of the word senses in corpora.
- Early work on word sense disambiguation used machine readable dictionaries or thesaurus as knowledge sources.
- Two prominent sources were the Longman dictionary of contemporary English (LDOCE) and Roget's Thesaurus.
- The biggest sense annotation corpus OntoNotes released through Linguistic Data Consortium (LDC).
- The Chinese annotation corpus is HowNet.

Systems:

Researchers have explored various system architectures to address the sense disambiguation problem.

We can classify these systems into four main categories: i. rules based or knowledge ii. Supervised iii.unsupervised iv. Semisupervised

Rule Based:

- The first-generation of word sense disambiguation systems was primarily based on dictionary sense definitions.
- Much of this information is historical and cannot readily be translated and made available for building systems today. But some of techniques and algorithms are still available.

- The simplest and oldest dictionary-based sense disambiguation algorithm was introduced by **Lesk.**

The core of the algorithm is that the dictionary senses whose terms most closely overlap with the terms in the context.

Algorithm 4–1 Pseudocode of the simplified Lesk algorithm
The function COMPUTEOVERLAP returns the number of words common to the two sets
Procedure: SIMPLIFIED_LESK(word, sentence) **returns** best sense of *word*

```
1:  best-sense ← most frequent sense of word
2:  max-overlap ← 0
3:  context ← set of words in sentence
4:  for all sense ∈ senses of word do
5:      signature ← set of words in gloss and examples of sense
6:      overlap ← COMPUTEOVERLAP(signature, context)
7:      if overlap gt max-overlap then
8:          max-overlap ← overlap
9:          best-sense ← sense
10:     end if
11: end for
12: return best-sense
```

The Simplified Lesk algorithm

- Let's disambiguate "**bank**" in this sentence:

  The **bank** can guarantee deposits will eventually cover future tuition costs because it invests in adjustable-rate mortgage securities.

- given the following two WordNet senses:

bank[1]	Gloss:	a financial institution that accepts deposits and channels the money into lending activities
	Examples:	"he cashed a check at the bank", "that bank holds the mortgage on my home"
bank[2]	Gloss:	sloping land (especially the slope beside a body of water)
	Examples:	"they pulled the canoe up on the bank", "he sat on the bank of the river and watched the currents"

Choose sense with most word overlap between gloss and context
(not counting function words)

The **bank** can guarantee deposits will eventually cover future tuition costs because it invests in adjustable-rate mortgage securities.

bank[1]	Gloss:	a financial institution that accepts deposits and channels the money into lending activities
	Examples:	"he cashed a check at the bank", "that bank holds the mortgage on my home"
bank[2]	Gloss:	sloping land (especially the slope beside a body of water)
	Examples:	"they pulled the canoe up on the bank", "he sat on the bank of the river and watched the currents"

Another dictionary-based algorithm was suggested Yarowsky.

This study used Roget's Thesaurus categories and classified unseen words into one of these 1042 categories based on a statistical analysis of 100 word concordances for each member of each category.

The method consists of three steps, as shown in Fig below.

- The first step is a collection of contexts.
- The second step computes weights for each of the salient words.
- P(w|Rcat) is the probability of a word w occurring in the context of a Roget's Thesaurus category Rcat.
- P(w|Rcat) |Pr(w) , the probability of a word (w) appearing in the context of a Roget category divided by its overall probability in the corpus.
- Finally, in third step, the unseen words in the test set are classified into the category that has the maximum weight.

1. Collect contexts for each of the *Roget's Thesaurus* categories.
2. Determine weights for each of the salient words in the context.

$$\frac{P(w_i|RCat)}{P(w_i)}$$

3. Use the weights for predicting the appropriate category of the word in the test corpus.

$$\arg\max_{RCat} \sum_{W} \log \frac{P(w_i|RCat)P(RCat)}{P(w_i)}$$

Figure 4–2: Algorithm for disambiguating words into *Roget's Thesaurus* categories

WALKER'S ALGORITHM

- A Thesaurus Based approach.
- ***Step 1:*** *For each sense of the target word find the thesaurus category to which that sense belongs.*
- ***Step 2:*** *Calculate the score for each sense by using the context words. A context words will add 1 to the score of the sense if the thesaurus category of the word matches that of the sense.*
  - *E.g. The money in this **bank** fetches an interest of 8% per annum*
  - Target word: ***bank***
  - Clue words from the context: ***money, interest, annum, fetch***

	Sense1: Finance	Sense2: Location
Money	+1	0
Interest	+1	0
Fetch	0	0
Annum	+1	0
Total	3	0

Context words add 1 to the sense when the topic of the word matches that of the sense

Supervised:

- The simpler form of word sense disambiguating systems the supervised approach, which tends to transfer all the complexity to the machine learning machinery while still requiring hand annotation tends to be superior to unsupervised and performs best

when tested on annotated data.

- These systems typically consist of a machine learning classifier trained on various features extracted for words that have been manually disambiguated in a given corpus and the application of the resulting models to disambiguating words in the unseen test sets.
- A good feature of these systems is that the user can incorporate rules and knowledge in the form of features.

Classifier:

Probably the most common and high performing classifiers are support vector machine (SVMs) and maximum entropy classifiers.

Features: Here we discuss a more commonly found subset of features that have been useful in supervised learning of word sense.

Lexical context: The feature comprises the words and lemma of words occurring in the entire paragraph or a smaller window of usually five words.

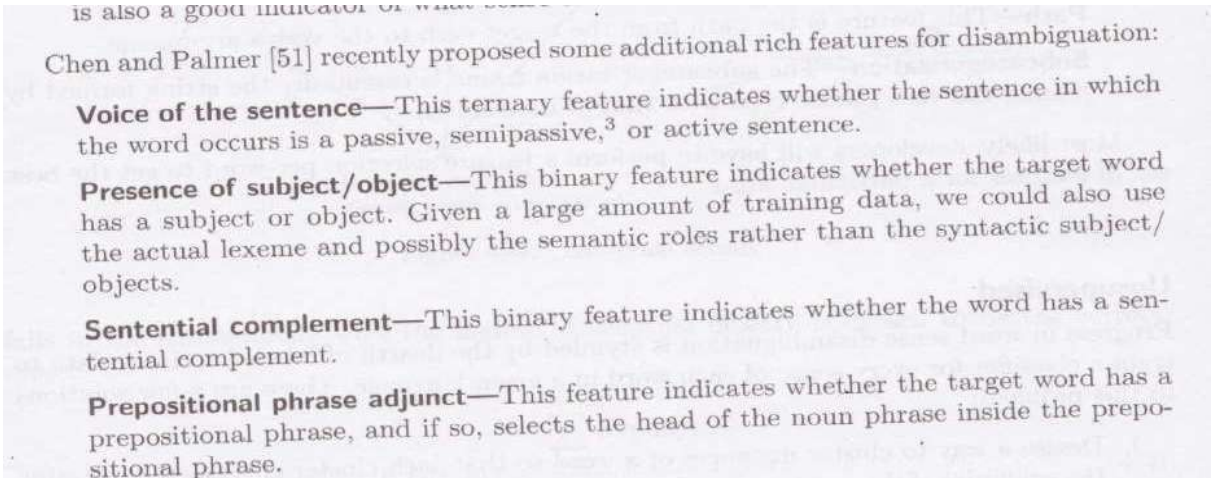
Parts of speech: the feature comprises the surrounding the word that is being sense tagged.

Bag of words context: this feature comprises using an unordered set of words in the context window.

Local Collocations: Local **collocations** are an ordered sequence of phrases near the target word that provide semantic context for disambiguation. Usually, a very small window of about three tokens on each side of the target word, most often in contiguous pairs or triplets, are added as a list of features.

Syntactic relations: if the parse of the sentence containing the target word is available, then we can use syntactic features.

Topic features: The board topic, or domain, of the article that word belongs to is also a good indicator of what sense of the word might be most frequent.



is also a good indicator of what sense

Chen and Palmer [51] recently proposed some additional rich features for disambiguation:

Voice of the sentence—This ternary feature indicates whether the sentence in which the word occurs is a passive, semipassive,[3] or active sentence.

Presence of subject/object—This binary feature indicates whether the target word has a subject or object. Given a large amount of training data, we could also use the actual lexeme and possibly the semantic roles rather than the syntactic subject/objects.

Sentential complement—This binary feature indicates whether the word has a sentential complement.

Prepositional phrase adjunct—This feature indicates whether the target word has a prepositional phrase, and if so, selects the head of the noun phrase inside the prepositional phrase.

Named entity—This feature is the named entity of the proper nouns and certain types of common nouns.

WordNet—WordNet synsets of the hypernyms of head nouns of the noun phrase arguments of verbs and prepositions.

[...] recently, following research in semantic role labeling, Dligach and Palmer [52]

"Word sense disambiguation" (WSD) is a natural language processing (NLP) task that involves determining the correct sense or meaning of a word within a given context. Many words in natural language have multiple meanings or senses, and WSD aims to choose the most appropriate sense for a word in a specific sentence or context.

Supervised learning with Support Vector Machines (SVM) is one approach to solving the WSD problem. Here's how it works:

1. **Data Collection**: To train an SVM for WSD, you need a labeled dataset where each word is tagged with its correct sense in various contexts. This dataset is typically created by human annotators who assign senses to words in sentences.
2. **Feature Extraction**: For each word in the dataset, you need to extract relevant features from its context. These features could include the words surrounding the target word, part-of-speech tags, syntactic information, and more. These features serve as the input to the SVM.
3. **Training**: Once you have the labeled dataset and extracted features, you can train an SVM classifier. The goal is to teach the SVM to learn patterns in the features that are indicative of specific word senses.
4. **Testing/Predicting**: After training, you can use the SVM to predict the sense of an ambiguous word in a new, unseen sentence. The SVM considers the context features and assigns the word the most likely sense based on what it learned during training.
5. **Evaluation**: To assess the performance of your WSD system, you can use various evaluation metrics, such as accuracy, precision, recall, and F1-score. These metrics help you measure how well your SVM-based WSD system is performing in disambiguating word senses.

SVMs are popular for WSD because they are effective at handling high-dimensional feature spaces and can learn complex decision boundaries. However, the success of the SVM-based WSD system heavily depends on the quality of the labeled dataset and the choice of features used for training.

Algorithm 4–2 Rules for selecting syntactic relations as features

```
if w is a noun then
    select parent head word (h)
    select part of speech of h
    select voice of h
    select position of h (left, right)
else if w is a verb then
    select nearest word l to the left of w such that w is the parent head word of l
    select nearest word r to the right of w such that w is the parent head word of r
    select part of speech of l
    select part of speech of r
    select part of speech of w
    select voice of w
else if w is a adjective then
    select parent head word (h)
    select part of speech of h
end if
```

The identification of the head word is important in syntax because it helps determine the grammatical structure of a phrase or sentence. For feature selection in NLP tasks like parsing or word sense disambiguation, knowing the head word and its relationships with other words in a

sentence can be valuable information. Syntactic relations often involve the relationship between a head word and its dependents or modifiers, and these relations can be used as features in various natural language processing applications.

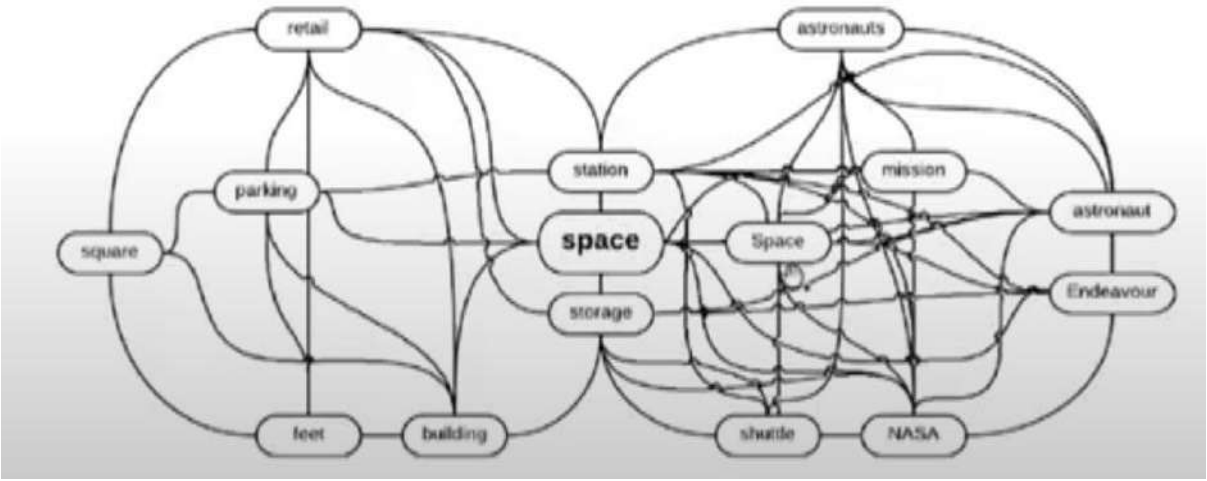
Unsupervised:

Unsupervised learning in Natural Language Processing (NLP) is a category of machine learning where the model is trained on unlabeled data without explicit supervision or predefined categories. It aims to discover patterns, structures, or representations within the data. One concept related to unsupervised learning in NLP is "Conceptual Density."

HyperLex

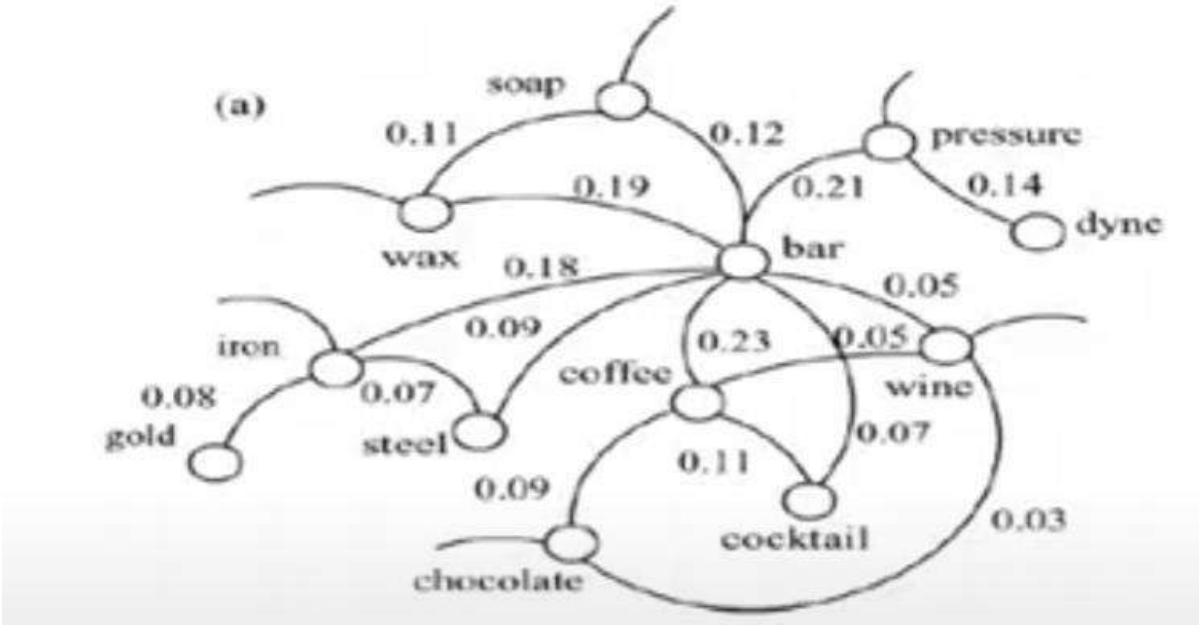
Key Idea: Word Sense Induction

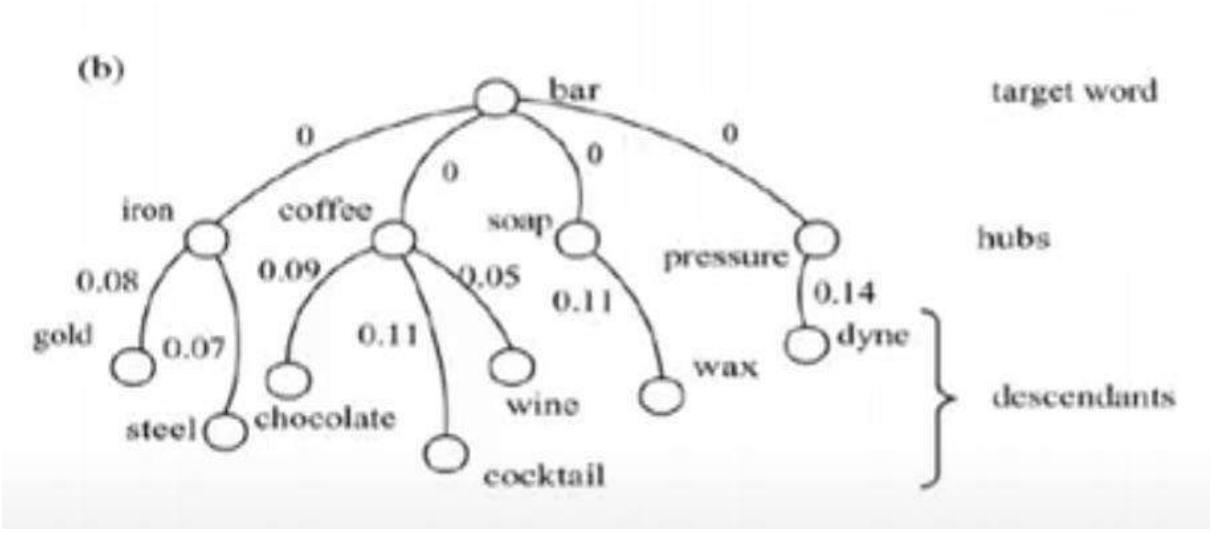
- Instead of using "dictionary defined senses", extract the "senses from the corpus" itself
- These "corpus senses" or "uses" correspond to clusters of similar contexts for a word.



Detecting Root Hubs

- Different uses of a target word form highly interconnected bundles (or high density components)
- In each high density component one of the nodes (hub) has a higher degree than the others.
- **Step 1:** Construct co-occurrence graph, G.
- **Step 2:** Arrange nodes in G in decreasing order of degree.
- **Step 3:** Select the node from G which has the highest degree. This node will be the hub of the first high density component.
- **Step 4:** Delete this hub and all its neighbors from G.
- **Step 5:** Repeat Step 3 and 4 to detect the hubs of other high density components





- Attach each node to the root hub closest to it.
- The distance between two nodes is measured as the smallest sum of weights of the edges on the paths linking them.

Computing distance between two nodes w_i and w_j

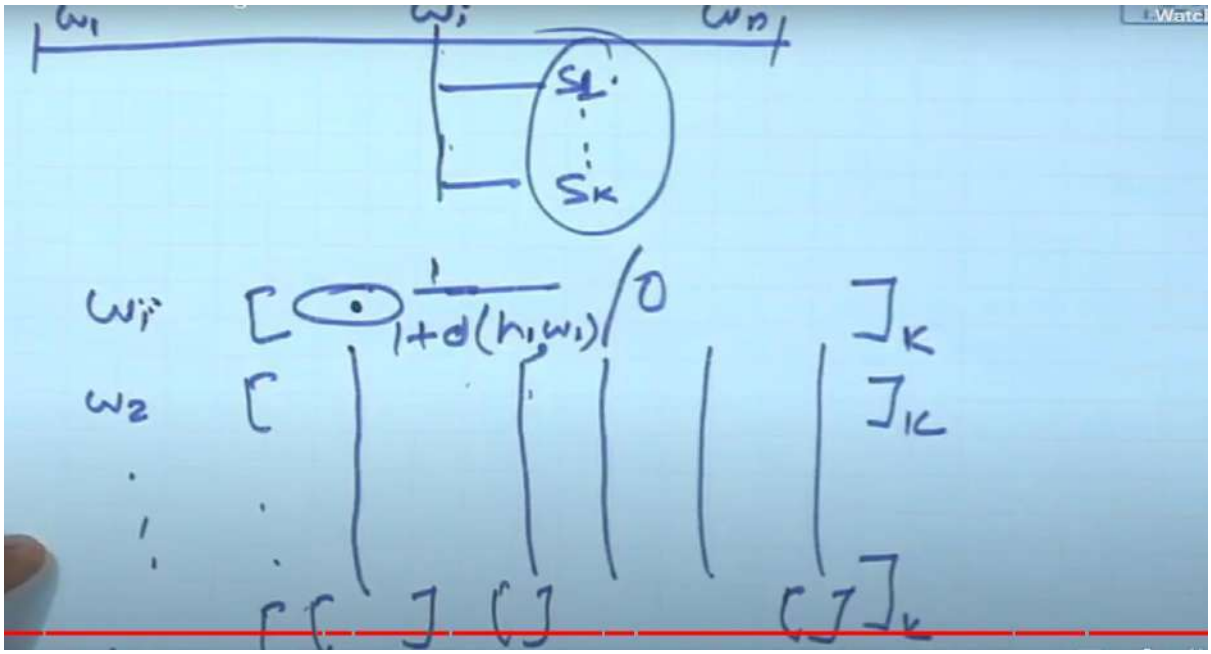
$$w_{ij} = 1 - max\{P(w_i|w_j), P(w_j|w_i)\}$$

where $P(w_i|w_j) = \frac{freq_{ij}}{freq_j}$

- Let $W = (w_1, w_2, \ldots, w_i, \ldots, w_n)$ be a context in which w_i is an instance of our target word.
- Let w_i has k hubs in its minimum spanning tree
- A score vector s is associated with each $w_j \in W(j \neq i)$, such that s_k represents the contribution of the kth hub as:

$$s_k = \frac{1}{1 + d(h_k, w_j)} \text{ if } h_k \text{ is an ancestor of } w_j$$
$$s_i = 0 \text{ otherwise.}$$

- All score vectors associated with all $w_j \in W(j \neq i)$ are summed up
- The hub which receives the maximum score is chosen as the mo
appropriate sense



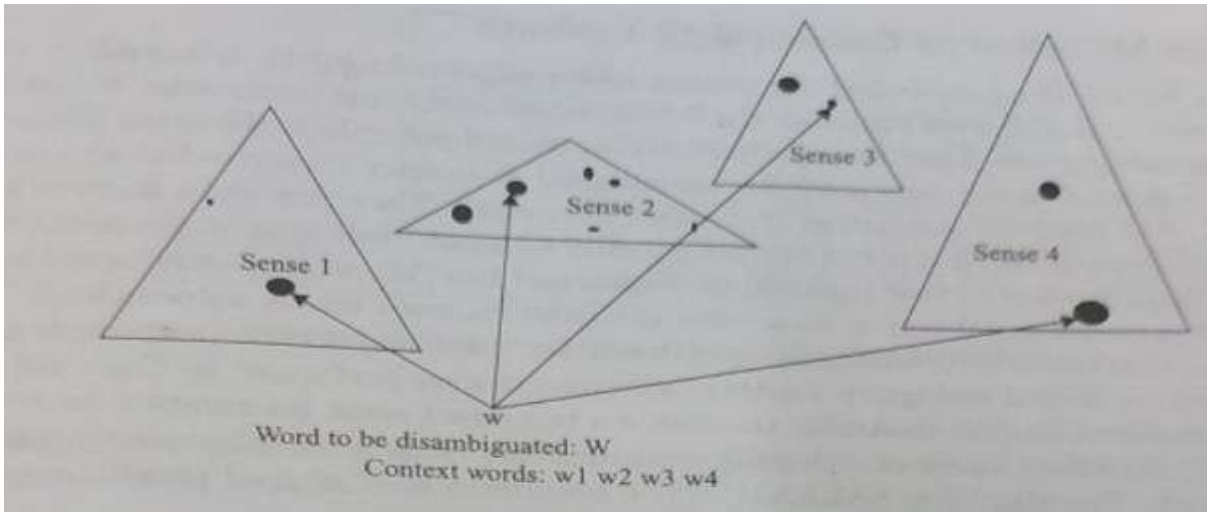


Figure: Conceptual Density

Semi Supervised:

Semi-supervised learning is a machine learning paradigm that combines both labeled and unlabeled data to improve model performance. In the context of word sense disambiguation

(WSD) in Natural Language Processing (NLP), semi-supervised learning techniques can be quite beneficial because labeled data for WSD is often limited and expensive to obtain. Here's an overview of a semi-supervised learning algorithm for WSD:

Self-Training for WSD:

Self-training is a popular semi-supervised learning approach that can be adapted for WSD. In self-training for WSD, you start with a small set of labeled examples and a larger set of unlabeled examples. The process involves iterative steps:

1. **Initialization**: Begin with a small labeled dataset where each example consists of a sentence containing an ambiguous word and its corresponding sense label.
2. **Feature Extraction**: Extract relevant features from the labeled examples, which typically include information about the target word, its context words, part-of-speech tags, syntactic relations, and more.
3. **Model Training**: Train a WSD model using the labeled data. This can be a supervised machine learning model like Support Vector Machines (SVM), Naive Bayes, or a neural network-based model.
4. **Prediction**: Use the trained model to predict word senses for the unlabeled data. Apply the model to the sentences containing the ambiguous word from the unlabeled dataset to assign senses to those instances.
5. **Confidence Threshold**: Introduce a confidence threshold or some criteria to filter the predictions. For instance, you can choose to keep only the predictions where the model is highly confident.
6. **Adding Labeled Data**: Add the confidently predicted examples to the labeled dataset, marking them as newly labeled instances.
7. **Iteration**: Repeat steps 2-6 for a fixed number of iterations or until convergence.
8. **Final Model**: Train a final model using the combined labeled data (original labeled dataset plus the newly labeled instances) to create a more robust WSD model.

Advantages of Self-Training for WSD:

- It leverages a larger pool of unlabeled data, which can be especially beneficial when labeled data is scarce.
- It allows the model to learn from its own predictions and iteratively improve.
- Self-training is a flexible approach and can be used with various machine learning models.

Challenges:

- Labeling errors: The initial labeled dataset should be of high quality because errors can accumulate during self-training iterations.

Semi-supervised learning with self-training can be effective for WSD, but it's essential to carefully design the process, monitor model performance, and apply filtering criteria to ensure the quality of the added labeled instances.

Motivation and concept of Yorowsky algorithm

- Annotations are expensive!
- "Bootstrapping" or co-training
  - Start with (small) seed, learn decision list
  - Use decision list to label rest of corpus
  - Retain 'confident' labels, treat as annotated data to learn new decision list
  - Repeat ...
- Heuristics (derived from observation):
  - One sense per discourse
  - One sense per collocation

One Sense per Discourse

- A word tends to preserve its meaning across all its occurrences in a given discourse

One Sense per Collocation

- A word tends to preserve its meaning when used in the same collocation
  - Strong for adjacent collocations
  - Weaker as the distance between the words increases

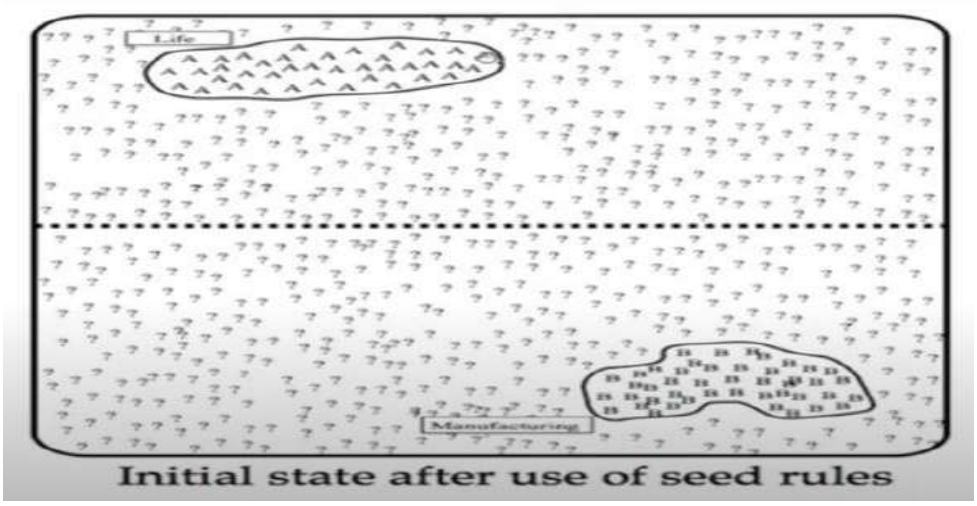
Example

- Disambiguating plant (industrial sense) vs. plant (living thing sense)
- Think of seed features for each sense
  - Industrial sense: co-occurring with 'manufacturing'
  - Living thing sense: co-occurring with 'life'
- Use 'one sense per collocation' to build initial decision list classifier
- Treat results (having high probability) as annotated data, train new decision list classifier, iterate

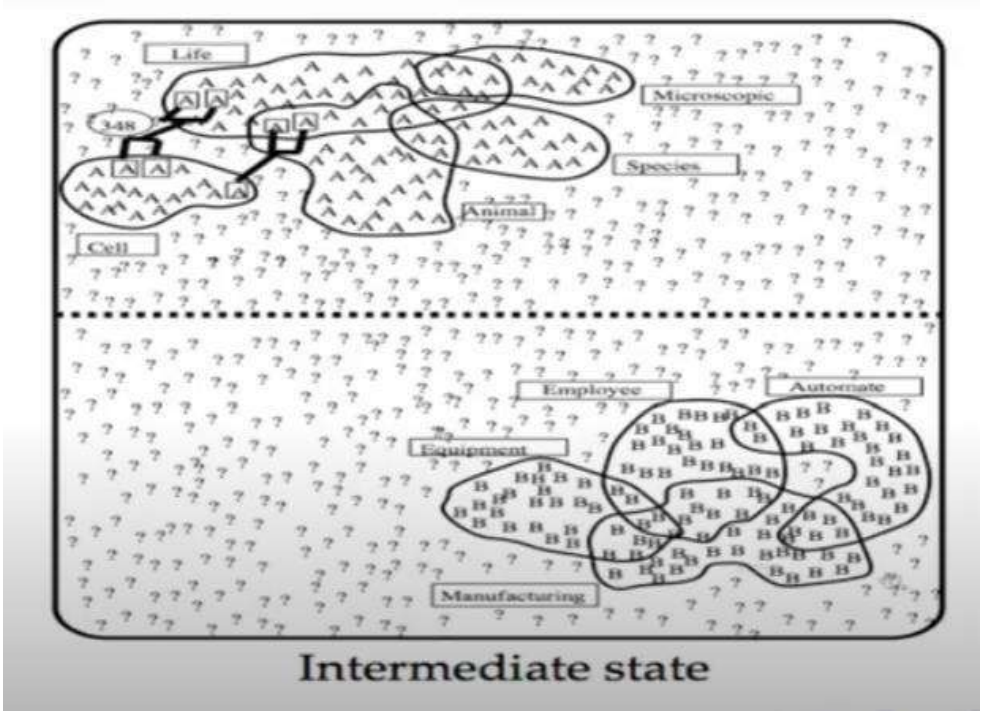
used to strain microscopic plant life from the
zonal distribution of plant life .
close-up studies of plant life and natural
too rapid growth of aquatic plant life in water
the proliferation of plant and animal life
establishment phase of the plant virus life cycle
that divide life into plant and animal kingdom
many dangers to plant and animal life
mammals . Animal and plant life are delicately

automated manufacturing plant in Fremont
vast manufacturing plant and distribution
chemical manufacturing plant , producing viscose
keep a manufacturing plant profitable without
computer manufacturing plant and adjacent
discovered at a St. Louis plant manufacturing
copper manufacturing plant found that they
copper wire manufacturing plant , for example

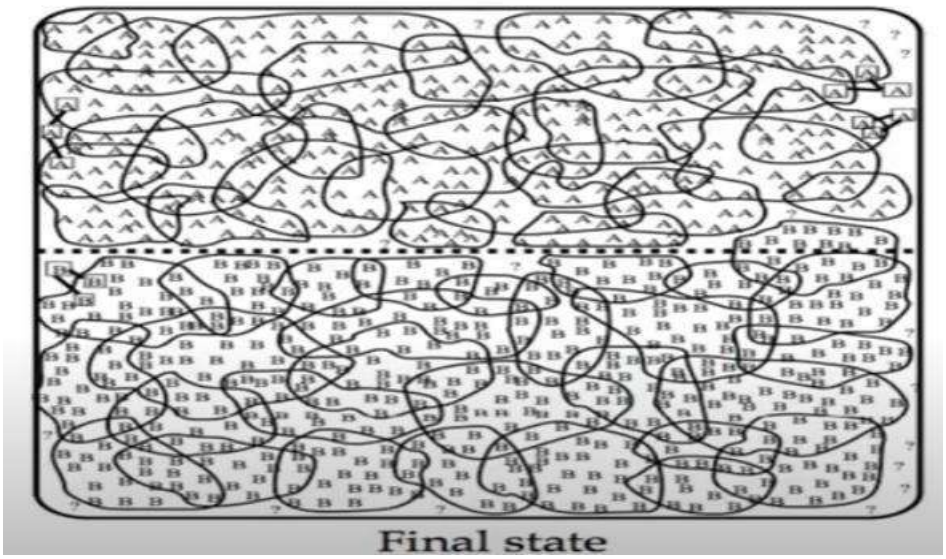
vinyl chloride monomer plant , which is
molecules found in plant and animal tissue
Nissan car and truck plant in Japan is
and Golgi apparatus of plant and animal cells
union responses to plant closures .
cell types found in the plant kingdom are
company said the plant is still operating
Although thousands of plant and animal species
animal rather than plant tissues can be



Initial state after use of seed rules



Intermediate state



Termination

- Stop when
  - Error on training data is less than a threshold
  - No more training data is covered
- Use final decision list for WSD

Advantages

- Accuracy is about as good as a supervised algorithm
- Bootstrapping: far less manual effort

Step 1. In a sufficiently large corpus, identify all the instances of a particular polysemous word that needs to be disambiguated, storing its context alongside.
Step 2. Identify a small set of instances that are strongly representative of one of the senses of the word. This can either be done in a completely unsupervised fashion by identifying collocations that give a strong indication of the sense usage for the word under consideration or by manually tagging a small portion of the data. In this example, we assume a polysemous word with only two senses, but this algorithm can be extended to n senses.
Step 3.

Step 3a. Train a supervised classifier on this set of examples.

Step 3b. Using these classifiers, classify the remaining instances of the word in the corpus and select those that are classified above a certain level of confidence.

Step 3c. Filter out the possible misclassifications using one sense per discourse constraint, and identify possible new collocations to be added to the list of seed collocations.

Step 3d. Repeat step 3 iteratively, thereby slowly shrinking the residual.

Step 4. Stop. At some point, a small, stable residual will remain.
Step 5. The trained classifier can now be used to classify new data, and that in turn can be used to annotate the original corpus with sense tags and probabilities.

Figure: Yorowsky algorithm

Word Embedding Techniques for semantic analysis

Word2Vec

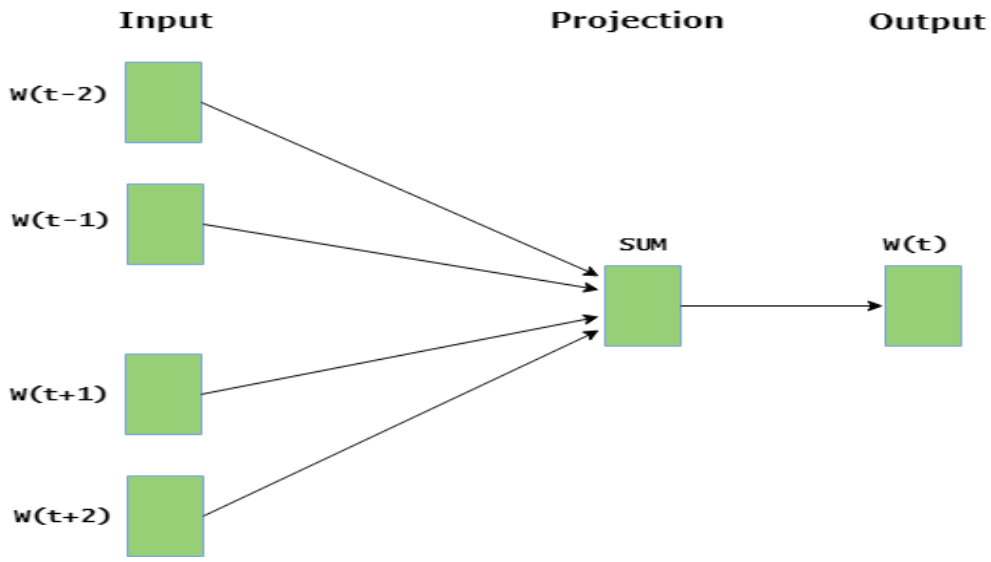
What is Word Embedding?

Word Embedding is a language modeling technique for mapping words to vectors of real numbers. It represents words or phrases in vector space with several dimensions. Word embeddings can be generated using various methods like neural networks, co-occurrence matrices, probabilistic models, etc. **Word2Vec** consists of models for generating word embedding. These models are shallow two-layer neural networks having one input layer, one hidden layer, and one output layer.

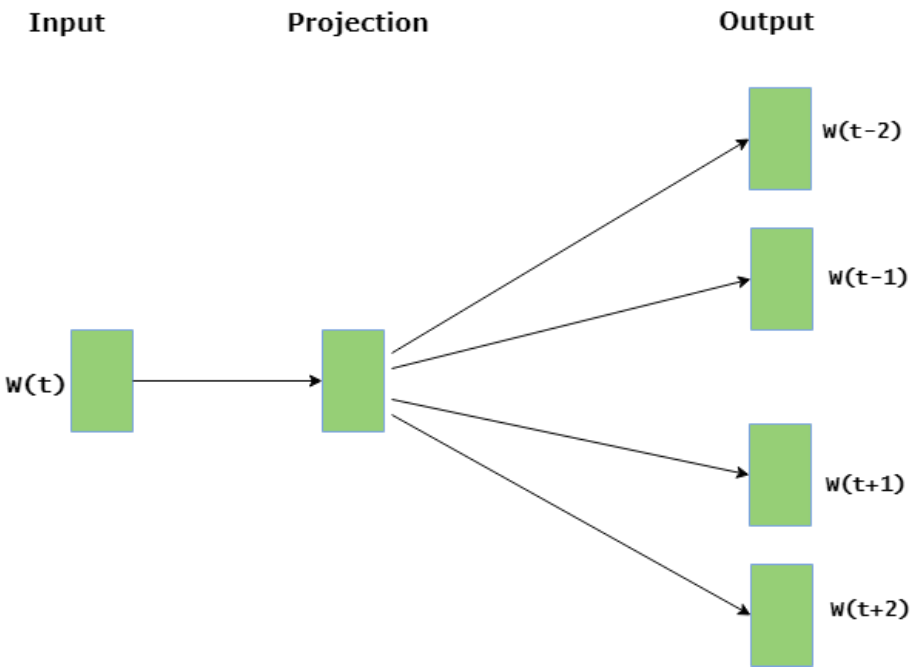
What is Word2Vec?

Word2Vec is a widely used method in natural language processing (NLP) that allows words to be represented as vectors in a continuous vector space. Word2Vec is an effort to map words to high-dimensional vectors to capture the semantic relationships between words, developed by researchers at Google. Words with similar meanings should have similar vector representations, according to the main principle of Word2Vec. Word2Vec utilizes two architectures:

- **CBOW (Continuous Bag of Words):** The CBOW model predicts the current word given context words within a specific window. The input layer contains the context words and the output layer contains the current word. The hidden layer contains the dimensions we want to represent the current word present at the output layer.



Skip Gram : Skip gram predicts the surrounding context words within specific window given current word. The input layer contains the current word and the output layer contains the context words. The hidden layer contains the number of dimensions in which we want to represent current word present at the input layer.



The basic idea of word embedding is words that occur in similar context tend to be closer to each other in vector space. For generating word vectors in Python, modules needed are **nltk** and **gensim**. Run these commands in terminal to install **nltk** and

gensim:

```
pip install nltk
pip install gensim
```

- **NLTK:** For handling human language data, NLTK, or Natural Language Toolkit, is a potent Python library. It offers user-friendly interfaces to more than 50 lexical resources and corpora, including WordNet. A collection of text processing libraries for tasks like categorization, tokenization, stemming, tagging, parsing, and semantic reasoning are also included with NLTK.
- **GENSIM:** Gensim is an open-source Python library that uses topic modelling and document similarity modelling to manage and analyse massive amounts of unstructured text data. It is especially well-known for applying topic and vector space modelling algorithms, such as Word2Vec and Latent Dirichlet Allocation (LDA), which are widely used.

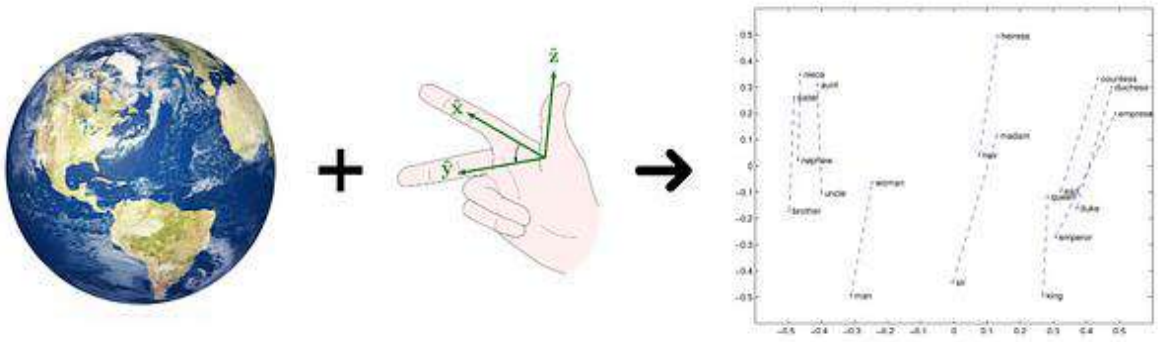
Why we need Word2Vec?

In natural language processing (NLP), Word2Vec is a popular and significant method for representing words as vectors in a continuous vector space. Word2Vec has become popular and is utilized in many different NLP applications for several reasons:

- **Semantic Representations:** Word2Vec records the connections between words semantically. Words are represented in the vector space so that similar words are near to one another. This enables the model to interpret words according to their context within a particular corpus.
- **Distributional Semantics:** The foundation of Word2Vec is the distributional hypothesis, which holds that words with similar meanings are more likely to occur in similar contexts. Word2Vec generates vector representations that reflect semantic similarities by learning from the distributional patterns of words in a large corpus.
- **Vector Arithmetic:** Word2Vec generates vector representations that have intriguing algebraic characteristics. Vector arithmetic, for instance, can be used to record word relationships. One well-known example is that the vector representation of "queen" could resemble the vector representation of "king" less "man" plus "woman."

- **Efficiency:** Word2Vec's high computational efficiency makes training on big datasets possible. Learning high-dimensional vector representations for a large vocabulary requires this efficiency.
- **Transfer Learning:** A variety of natural language processing tasks can be initiated with pre-trained Word2Vec models. Time and resources can be saved by fine-tuning the embeddings discovered on a sizable dataset for particular uses.
- **Applications:** Word2Vec embeddings have shown promise in a number of natural language processing (NLP) applications, such as machine translation, text classification, sentiment analysis, and information retrieval. These applications are successful in part because of their capacity to capture semantic relationships.
- **Scalability:** Word2Vec can handle big corpora with ease and is scalable. Scalability like this is essential for training on large text datasets.
- **Open Source Implementations:** Word2Vec has open-source versions, including one that is included in the Gensim library. Its widespread adoption and use in both research and industry can be attributed in part to its accessibility

Global Vector for word representation (GloVe),



GloVe embedding improves Word-2-Vec's by computing the co-occurrence of corpus words once and deriving vector representations from it. GloVe also takes a descriptive approach towards deriving its cost function by first examining the expected outcome of co-occurrence ratios.

GloVe implementation is often based on CRF (Conditional Random Field) which is not only a great application of Hidden Markov Model, it leads to a discussion on Forward-Backward and Viterbi algorithms. I also found it to serve as a prelude to *Bayesian* vs *Generative* models debate.

The focus of this article is to describe the GloVe model.

GloVe Intuition

In both Word-2-Vec and GloVe, we design word embeddings such that they reflect probability of seeing pairs of words together. Word-2-Vec is based on presence of neighboring words in a moving window fashion. In case of Skipgram, we observe what words appear in a neighborhood of any given corpus word. The cost function minimizes negative likelihood of observing expected neighbor words given a center word, Equation (1).

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{-m\leq j\leq m}\log\left(p(w_{t+j}|w_t;\theta)\right), \quad for\ j \neq 0 \qquad (1)$$

Note that in Word-2-Vec we tend to re-visit neighboring words multiple times as a new center word vector is being trained.

Aware of this duplication of effort, GloVe training starts by forming a co-occurrence matrix X where the ij-th entry is the number of times words on i-th row and j-th column appeared together, Figure (1).

A fair question here is, over what window the co-occurrence matrix is formed?

Co-Occurrence Count Approaches

a. Window Count

Where a window length is decided in advance, e.g. 5 or 10. This approach tends to capture both syntactic, e.g. Part of Speech, and semantic information.

b. Document Count

Where co-occurrences are across documents, per document in the corpus. This approach tends to extracts general topics information and leads to Latent Semantic Analysis.

Note that in both approaches, the co-occurrence matrix is calculated once across the entire corpus and later processed to train the word vectors. This proves to enhance the training time significantly in comparison to Word-2-Vec.

Figure (1) depicts one such co-occurrence matrix for a window size of 1 when corpus consists of only the following three sentences, excluding punctuations.

- I like deep learning.
- I like NLP.
- I enjoy flying.

counts	I	like	enjoy	deep	learning	NLP	flying
I	0	2	1	0	0	0	0
like	2	0	0	1	0	1	0
enjoy	1	0	0	0	0	0	1
deep	0	1	0	0	1	0	0
learning	0	0	0	1	0	0	0
NLP	0	1	0	0	0	0	0
flying	0	0	1	0	0	0	0

Co-Occurrence Ratios

After counting the co-occurrences, GloVe argues that for any group of three corpus words, *w1, w2, w3*; one of the following three scenarios applies:

1. ***w3* is relevant to both *w1* and *w2:***

$$\frac{P(w_3|w_1)}{P(w_3|w_2)} \approx 1 \qquad (2)$$

For example, let *w1= "gymnastics", w2= "javelin"* and *w3= "olympics"*:

$$P(olympics|gymnastics) = \frac{\#(olympics, gymnastics)}{\sum_{w \in corpus} \#(w, gymnastics)}$$

$$P(olympics|javelin) = \frac{\#(olympics, javelin)}{\sum_{w \in corpus} \#(w, javelin)}$$

where *#(v, z)* denotes the respective entry in the co-occurrence matrix representing words *v* and *z*. It's not hard to see Equation (2) holds:

$$\frac{P(olympics|gymnastics)}{P(olympics|javelin)} \approx 1$$

$w3$ is only relevant to one of $w1$ or $w2$, not both:

$$\frac{P(w_3|w_1)}{P(w_3|w_2)} \gg 1 \qquad (3)$$

$$\frac{P(w_3|w_1)}{P(w_3|w_2)} \ll 1 \qquad (4)$$

Following the previous example, let *w3= "vault"* or *w3= "spear"*. Take a moment to see how the ratio this time will be significantly smaller or larger than 1.

$$\frac{P(vault|gymnastics)}{P(vault|javelin)} \gg 1$$

$$\frac{P(spear|gymnastics)}{P(spear|javelin)} \ll 1$$

3. *w3* is irrelevant to both *w1* and *w2:*

$$\frac{P(w_3|w_1)}{P(w_3|w_2)} \approx 1 \qquad (5)$$

Continuing on the same example, let *w3= "croissant"*. Once again the ratio tends to be close to 1, as in case 1.

$$\frac{P(croissant|gymnastics)}{P(croissant|javelin)} \approx 1$$

From Equations (2) to (5) we conclude that the co-occurrence ratio can help distinguish case 2 from case 1 or 3. This becomes the core idea in formulating GloVe's cost function.

GloVe Cost Function

Let *F* denote the procedure through which we learn GloVe word embeddings. From Equations (2) to (5) we expect *F* to be of the following general form:

$$F(w_i, w_j, w_k) = \frac{P(w_k|w_i)}{P(w_k|w_j)} \qquad (6)$$

Inputs to the left hand side of Equation (6) are vectors while the right hand side is scalar. One common way to meet this requirement is for *F* to include the inner product of its inputs in its formulation. Following similar deduction line, laid out in great details in [1], GloVe cost function takes the following form:

$$J(\theta) = \frac{1}{2}\sum_{i,j=1}^{W} f(x_{ij})\left(w_i^T w_j - \log x_{ij}\right)^2 \quad (7)$$

$$f(x) = \begin{cases} (x/x_{max})^\alpha & \text{if } x < x_{max} \\ 1 & \text{otherwise} \end{cases} \quad (8)$$

x_{ij} being the (i, j)-th entry in co-occurrence matrix,
w_i and w_j being the word vector embeddings,
$\alpha = 3/4,\ x_{max} = 100.$

α and *x* are heuristically chosen. Here I add a note on the choice of *f,* Equation (8), as I found it insightful in conducting other research works.

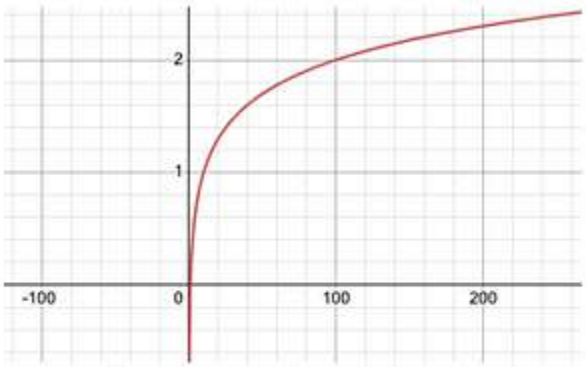
Log(x) function is well-behaved for *x≥1,* but it diverges when *0≤x<1*. Since *x=0* is a probable value in the co-occurrence matrix *X*, we need to account for this divergence. *Log(1+x)* would fix the issue by uniformly shifting all values but, we can do better!

Note that when $x \to 0$, we are dealing with rarely co-occurring words. What if the cost function was designed such that it under-weighted low-occurring words as compared to high-occurring combinations? This is the role of *f* from Equation (8).

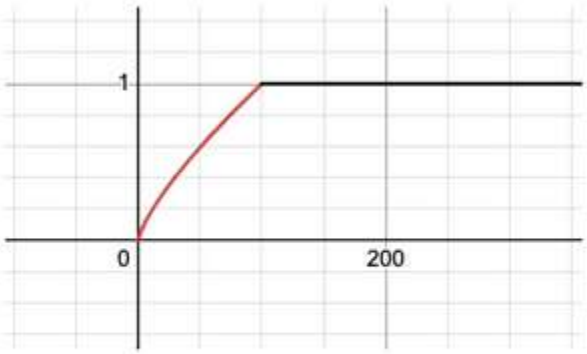
To decide *f*, let's specify its qualities first:

1. *f(0) = 0*
2. *f(x)* is monotonically increasing.
3. *f(x)* is asymptotically bounded.

The first condition is somewhat self explanatory, although based on the context can be adjusted to any intercept/constant value. The second condition states that *f* is never decreasing. Because otherwise a less frequently co-occurring pair is weighted higher than a pair with higher co-occurrence. The third condition ensures highly co-occurring pairs are not over-weighted disproportionately. This is especially true as information decays in overly high co-occurring pair of words. Such pairs tend to contain a stop-word, e.g. (*"the"*, *nouns*), which carries little information.



Log(x)



f(x) from Equation (8)

During GloVe's training, we iteratively try to minimize Equation (7). That is to arrive at word vectors such that the distance between their pairwise inner product and log of their co-occurrence is minimized.

You may have already picked up on the symmetry of word vectors on the left and right of inner product. Since the co-occurrence matrix is symmetric, each word gets two vector embeddings, once when its read as column value and once as row value. GloVe assigns the sum of these two vectors as the final word embedding:

$$w_{final} = w_l + w_r \qquad (9)$$

w_l, w_r are the learned vectors from $(w_l^T w_r - \log X_{lr})$ term in Equation (8).

Bidirectional encoder representations from transformers (BERT)

What is BERT?

BERT (Bidirectional Encoder Representations from Transformers) leverages a transformer-based neural network to understand and generate human-like language. BERT employs an encoder-only architecture. In the original **Transformer architecture,** there are both encoder and decoder modules. The decision to use an encoder-only architecture in BERT suggests a primary emphasis on understanding input sequences rather than generating output sequences.

Bidirectional Approach of BERT

Traditional language models process text sequentially, either from left to right or right to left. This method limits the model's awareness to the immediate context preceding the target word. BERT uses a bi-directional approach considering both the left and right context of words in a sentence, instead of analyzing the text sequentially, BERT looks at all the words in a sentence simultaneously.

Example: "The bank is situated on the _____ of the river."

In a unidirectional model, the understanding of the blank would heavily depend on the preceding words, and the model might struggle to discern whether "bank" refers to a financial institution or the side of the river.

BERT, being bidirectional, simultaneously considers both the left ("The bank is situated on the") and right context ("of the river"), enabling a more nuanced understanding. It comprehends that the missing word is likely related to the geographical location of the bank, demonstrating the contextual richness that the bidirectional approach brings.

Pre-training and Fine-tuning BERT Model

The BERT model undergoes a two-step process:

1. Pre-training on Large amounts of unlabeled text to learn contextual embeddings.
2. Fine-tuning on labeled data for specific NLP tasks.

Pre-Training on Large Data

- BERT is pre-trained on large amount of unlabeled text data. The model learns contextual embeddings, which are the representations of words that take into account their surrounding context in a sentence.
- BERT engages in various unsupervised pre-training tasks. For instance, it might learn to predict missing words in a sentence **(Masked Language Model or MLM task)**, understand the relationship between two sentences, or predict the next sentence in a pair.

Fine-Tuning on Labeled Data

- After the pre-training phase, the BERT model, armed with its contextual embeddings, is then fine-tuned for specific natural language processing (NLP) tasks. This step tailors the model to more

targeted applications by adapting its general language understanding to the nuances of the particular task.

- BERT is fine-tuned using labeled data specific to the downstream tasks of interest. These tasks could include sentiment analysis, question-answering, named entity recognition, or any other NLP application. The model's parameters are adjusted to optimize its performance for the particular requirements of the task at hand.

BERT's unified architecture allows it to adapt to various downstream tasks with minimal modifications, making it a versatile and highly effective tool in natural language understanding and processing.

How BERT work?

BERT is designed to generate a language model so, only the encoder mechanism is used. Sequence of tokens are fed to the Transformer encoder. These tokens are first embedded into vectors and then processed in the neural network. The output is a sequence of vectors, each corresponding to an input token, providing contextualized representations.

When training language models, defining a prediction goal is a challenge. Many models predict the next word in a sequence, which is a directional approach and may limit context learning.

BERT addresses this challenge with two innovative training strategies:

1. Masked Language Model (MLM)
2. Next Sentence Prediction (NSP)

1. Masked Language Model (MLM)

In BERT's pre-training process, a portion of words in each input sequence is masked and the model is trained to predict the original values of these masked words based on the context provided by the surrounding words.

In simple terms,

1. **Masking words:** Before BERT learns from sentences, it hides some words (about 15%) and replaces them with a special symbol, like [MASK].
2. **Guessing Hidden Words:** BERT's job is to figure out what these hidden words are by looking at the words around them. It's like a game of guessing where some words are missing, and BERT tries to fill in the blanks.
3. **How BERT learns:**
    - BERT adds a special layer on top of its learning system to make these guesses. It then checks how close its guesses are to the actual hidden words.
    - It does this by converting its guesses into probabilities, saying, "I think this word is X, and I'm this much sure about it."
4. **Special Attention to Hidden Words**

- BERT's main focus during training is on getting these hidden words right. It cares less about predicting the words that are not hidden.
- This is because the real challenge is figuring out the missing parts, and this strategy helps BERT become really good at understanding the meaning and context of words.

In technical terms,

1. BERT adds a classification layer on top of the output from the encoder. This layer is crucial for predicting the masked words.
2. The output vectors from the classification layer are multiplied by the embedding matrix, transforming them into the vocabulary dimension. This step helps align the predicted representations with the vocabulary space.
3. The probability of each word in the vocabulary is calculated using the SoftMax activation function. This step generates a probability distribution over the entire vocabulary for each masked position.
4. The loss function used during training considers only the prediction of the masked values. The model is penalized for the deviation between its predictions and the actual values of the masked words.
5. The model converges slower than directional models. This is because, during training, BERT is only concerned with predicting the masked values, ignoring the prediction of the non-masked words. The increased context awareness achieved through this strategy compensates for the slower convergence.

2. Next Sentence Prediction (NSP)

BERT predicts if the second sentence is connected to the first. This is done by transforming the output of the [CLS] token into a 2×1 shaped vector using a classification layer, and then calculating the probability of whether the second sentence follows the first using SoftMax.

1. In the training process, BERT learns to understand the relationship between pairs of sentences, predicting if the second sentence follows the first in the original document.
2. 50% of the input pairs have the second sentence as the subsequent sentence in the original document, and the other 50% have a randomly chosen sentence.
3. To help the model distinguish between connected and disconnected sentence pairs. The input is processed before entering the model:
   - A [CLS] token is inserted at the beginning of the first sentence, and a [SEP] token is added at the end of each sentence.
   - A sentence embedding indicating Sentence A or Sentence B is added to each token.
   - A positional embedding indicates the position of each token in the sequence.

4. BERT predicts if the second sentence is connected to the first. This is done by transforming the output of the [CLS] token into a 2×1 shaped vector using a classification layer, and then calculating the probability of whether the second sentence follows the first using SoftMax.

During the training of BERT model, the Masked LM and Next Sentence Prediction are trained together. The model aims to minimize the combined loss function of the Masked LM and Next Sentence Prediction, leading to a robust language model with enhanced capabilities in understanding context within sentences and relationships between sentences.

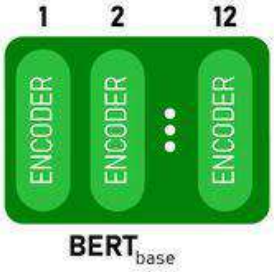
Why to train Masked LM and Next Sentence Prediction together?

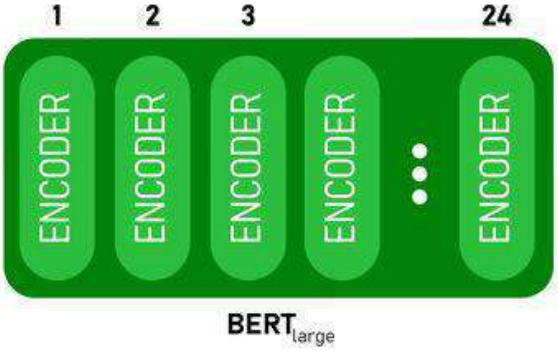
Masked LM helps BERT to understand the context within a sentence and Next Sentence Prediction helps BERT grasp the connection or relationship between pairs of sentences. Hence, training both the strategies together ensures that BERT learns a broad and comprehensive understanding of language, capturing both details within sentences and the flow between sentences.

BERT Architecture

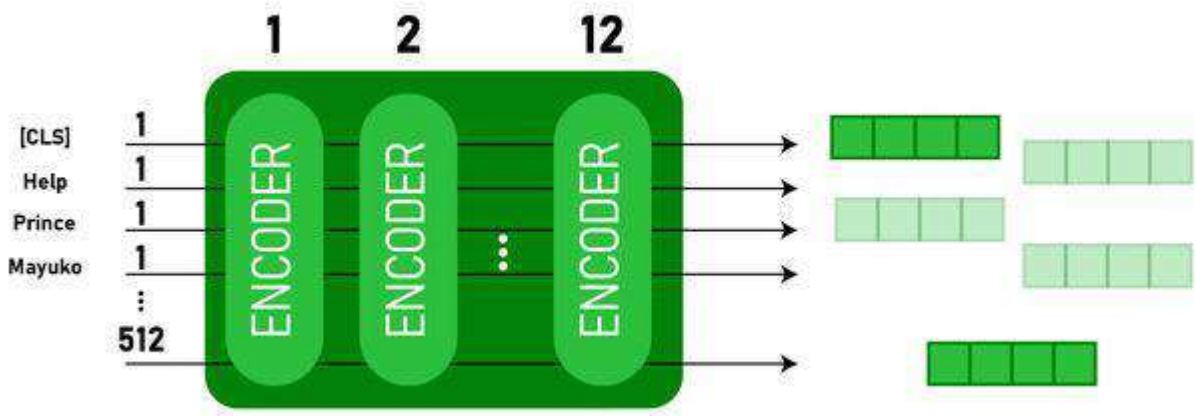
The architecture of BERT is a multilayer bidirectional transformer encoder which is quite similar to the transformer model. A transformer architecture is an encoder-decoder network that uses self-attention on the encoder side and attention on the decoder side.

1. BERTBASE has 1*2 layers in the Encoder stack* while BERTLARGE has *24 layers in the Encoder stack*. These are more than the Transformer architecture described in the original paper (*6 encoder layers*).
2. BERT architectures (BASE and LARGE) also have larger feedforward networks (768 and 1024 hidden units respectively), and *more attention heads (12 and 16 respectively)* than the Transformer architecture suggested in the original paper. It contains *512 hidden units and 8 attention heads*.
3. BERTBASE contains 110M parameters while BERTLARGE has 340M parameters.





This model takes the **CLS** token as input first, then it is followed by a sequence of words as input. Here CLS is a classification token. It then passes the input to the above layers. Each layer applies self-attention and passes the result through a feedforward network after then it hands off to the next encoder. The model outputs a vector of hidden size (*768* for BERT BASE). If we want to output a classifier from this model we can take the output corresponding to the CLS token.



Now, this trained vector can be used to perform a number of tasks such as classification, translation, etc. For Example, the paper achieves great results just by using a single layer Neural Network on the BERT model in the classification task.

How to use BERT model in NLP?

BERT can be used for various natural language processing (NLP) tasks such as:

1. Classification Task

- BERT can be used for classification task like sentiment analysis, the goal is to classify the text into different categories (positive/ negative/ neutral), BERT can be employed by adding a classification layer on the top of the Transformer output for the [CLS] token.

- The [CLS] token represents the aggregated information from the entire input sequence. This pooled representation can then be used as input for a classification layer to make predictions for the specific task.

2. Question Answering

- In question answering tasks, where the model is required to locate and mark the answer within a given text sequence, BERT can be trained for this purpose.
- BERT is trained for question answering by learning two additional vectors that mark the beginning and end of the answer. During training, the model is provided with questions and corresponding passages, and it learns to predict the start and end positions of the answer within the passage.

3. Named Entity Recognition (NER)

- BERT can be utilized for NER, where the goal is to identify and classify entities (e.g., Person, Organization, Date) in a text sequence.
- A BERT-based NER model is trained by taking the output vector of each token form the Transformer and feeding it into a classification layer. The layer predicts the named entity label for each token, indicating the type of entity it represents.

How to Tokenize and Encode Text using BERT?

To tokenize and encode text using BERT, we will be using the 'transformer' library in Python.

Command to install transformers:

```
!pip install transformers
```

- We will load the pretrained BERT tokenize with a cased vocabulary using ***BertTokenizer.from_pretrained("bert-base-cased").***
- ***tokenizer.encode(text)*** tokenizes the input text and converts it into a sequence of token IDs.
- ***print("Token IDs:", encoding)*** prints the token IDs obtained after encoding.
- ***tokenizer.convert_ids_to_tokens(encoding)*** converts the token IDs back to their corresponding tokens.
- print("Tokens:", tokens) prints the tokens obtained after converting the token IDs

```
from transformers import BertTokenizer

# Load pre-trained BERT tokenizer
tokenizer = BertTokenizer.from_pretrained("bert-base-cased")

# Input text
```

```
text = 'ChatGPT is a language model developed by OpenAI, based on the GPT (Generative Pre-trained Transformer) architecture. '

# Tokenize and encode the text
encoding = tokenizer.encode(text)

# Print the token IDs
print("Token IDs:", encoding)

# Convert token IDs back to tokens
tokens = tokenizer.convert_ids_to_tokens(encoding)

# Print the corresponding tokens
print("Tokens:", tokens)
```

Output:

Token IDs: [101, 24705, 1204, 17095, 1942, 1110, 170, 1846, 2235, 1872, 1118, 3353, 1592, 2240, 117, 1359, 1113, 1103, 15175, 1942, 113, 9066, 15306, 11689, 118, 3972, 13809, 23763, 114, 4220, 119, 102]
Tokens: ['[CLS]', 'Cha', '##t', '##GP', '##T', 'is', 'a', 'language', 'model', 'developed', 'by', 'Open', '##A', '##I', ',', 'based', 'on', 'the', 'GP', '##T', '(', 'Gene', '##rative', 'Pre', '-', 'trained', 'Trans', '##former', ')', 'architecture', '.', '[SEP]']

The **tokenizer.encode** method adds the special **[CLS] - classification** and **[SEP] - separator** tokens at the beginning and end of the encoded sequence. In the token IDs section, **token id: 101** refers to the start of the sentence and **token id: 102** represents the end of the sentence.

Application of BERT

BERT is used for:

1. **Text Representation:** BERT is used to generate word embeddings or representation for words in a sentence.

2. **Named Entity Recognition (NER)**: BERT can be fine-tuned for named entity recognition tasks, where the goal is to identify entities such as names of people, organizations, locations, etc., in a given text.
3. **Text Classification:** BERT is widely used for text classification tasks, including sentiment analysis, spam detection, and topic categorization. It has demonstrated excellent performance in understanding and classifying the context of textual data.
4. **Question-Answering Systems:** BERT has been applied to question-answering systems, where the model is trained to understand the context of a question and provide relevant answers. This is particularly useful for tasks like reading comprehension.
5. **Machine Translation:** BERT's contextual embeddings can be leveraged for improving machine translation systems. The model captures the nuances of language that are crucial for accurate translation.
6. **Text Summarization:** BERT can be used for abstractive text summarization, where the model generates concise and meaningful summaries of longer texts by understanding the context and semantics.
7. **Conversational AI:** BERT is employed in building conversational AI systems, such as chatbots, virtual assistants, and dialogue systems. Its ability to grasp context makes it effective for understanding and generating natural language responses.
8. **Semantic Similarity:** BERT embeddings can be used to measure semantic similarity between sentences or documents. This is valuable in tasks like duplicate detection, paraphrase identification, and information retrieval.

Predicate - Argument Structure :-

Shallow semantic parsing is popularly known as semantic role labelling. This semantic parsing is done with predicate argument structure. In the this predicate is verb arguments are subject and objects. Sometimes predicate may be noun, adjectives and prepositions in the sentence.

Resources :- for shallow semantic parsing two resources are extremely useful. In 1990, two important corpora were developed. They are frameNet and PropBank. These resources made a transition from rule based approaches to a more data oriented approaches. These approaches focus on transforming linguistic insights to features rather than rules so that the machine learning framework use those features to learn a model that helps automatically tag the semantic information encoded in such resources.

frameNet is based on frame semantics where a given predicate invokes a semantic frame, thus instantiating some or all of possible semantic roles belonging to that frame.

PropBank is based on Dowty's prototype theory. In this each predicate has a set of core arguments that are predicate dependent and all predicates share a set of non-core or adjuncive arguments. It builds on the syntactic PennTree Bank Corpus.

FrameNet:- FrameNet contains feame-specific semantic annotation of a number of predicates in English. It contains tagged sentences extracted from the British National Corpus (BNC). The process of FrameNet annotation consists of identifying specific semantic frames and creating a set of frame specific roles called frame elements. The figure below shows FrameNet example.

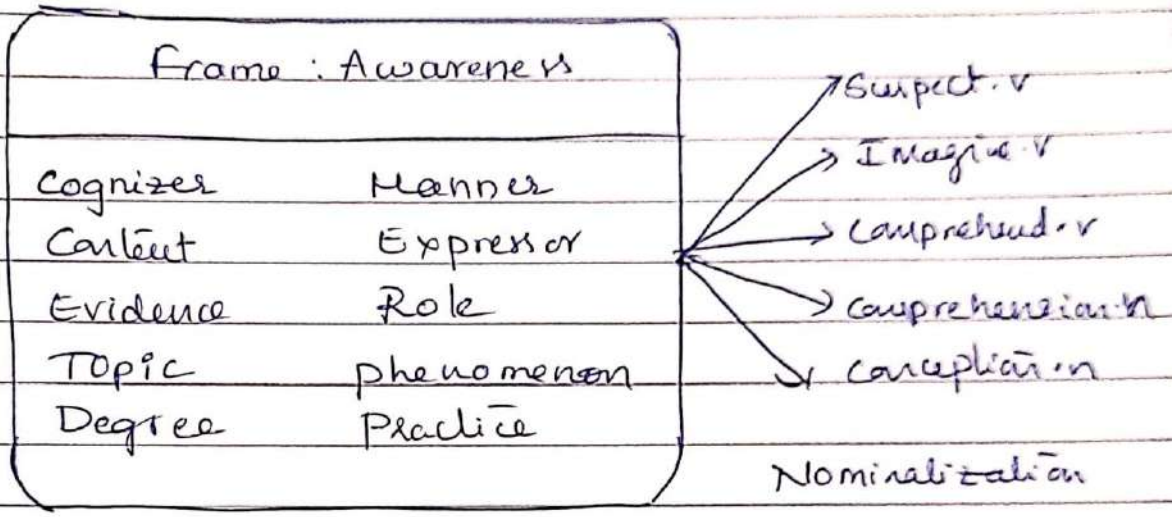


fig: FrameNet example

The following example illustrates the general idea. Here, the frame AWARENESS is instantiated by the verb predicate believe and the noun predicate comprehension. The above fig. shows the AWARENESS frame along with the frame-elements and a sample set of predicates that involve it- including verbs and nominalization

1. [cognizer We] [predicate-verb believe] [content it is a fair and generous price]
2. No doubt existed as to [cognizer Our] [predicate-noun comprehension] [content of it]

FrameNet includes a wide variety of nominal predicates.

The combination of predicate - lemma and the frame that its instance invokes is called a lexical unit (LU). This is therefore pairing of a word with its meaning.

The latest release of FrameNet, R1.5, contains about 1,43,000 predicate instances covering about 8,000 frame elements from roughly 1,000 frames over the BNC.

PropBank:- It only includes annotations of arguments of verb predicates. All the verbs in WSJ part of the Penn TreeBank have been labelled with their semantic arguments. The arguments are tagged as either core arguments with labels of type ARGN where N takes values from 0 to 5 or adjunctive arguments with labels of type ARGM-X, where X can take values of TMP, LOC and so on.

Adjunctive arguments share the same meaning across all predicates, where as the meaning of core arguments has to be interpreted in connection with a predicate. ARG0 is the PROTO-AGENT (usually the subject of a transitive verb) ARG1 is the PROTO-PATIENT (usually the direct object of transitive verb.

figure below shows a list of core arguments for the predicates 'operate' and 'author'

Predicate	Argument	Description
operate.01	ARG0	Agent, operator
	ARG1	Thing operated
	ARG2	Explicit patient
	ARG3	Explicit (thing operated on) argument
	ARG4	Explicit instrument

Predicate	Argument	Description
author.01	ARG0	Author, agent
	ARG1	Text authored.

The above two tables represent Argument labels associated with operate.01 (sense: work) and for author.01 (sense: to write or construct) in the PropBank corpus.

Table below shows list of adjunctive arguments in PropBank – ARGMs

Tag	Description	Example
ARGM-LOC	Locative	the museum, in Westborough, Mass
ARGM-TMP	Temporal	now, by next summer
ARGM-MNR	Manner	heavily, clearly, at a rapid rate
ARGM-DIR	Direction	to market, to Bangkok
ARGM-CAU	Cause	In response to the ruling
ARGM-DIS	Discourse	for example, in part, Similarly
ARGM-EXT	Extent	at $38.375, 50 points
ARGM-PRP	Purpose	to pay for the plant
ARGM-NEG	Negation	not, n't
ARGM-MOD	Modal	can, might, should, will
ARGM-REC	Reciprocals	each other
ARGM-PRD	Secondary Predication	to become a teacher
ARGM	Bare ARGM	With a Power expert
ARGM-ADV	-Adverbials	

An example extracted from the PropBank corpus along with its syntax tree representation and argument labels as shown in figure below.

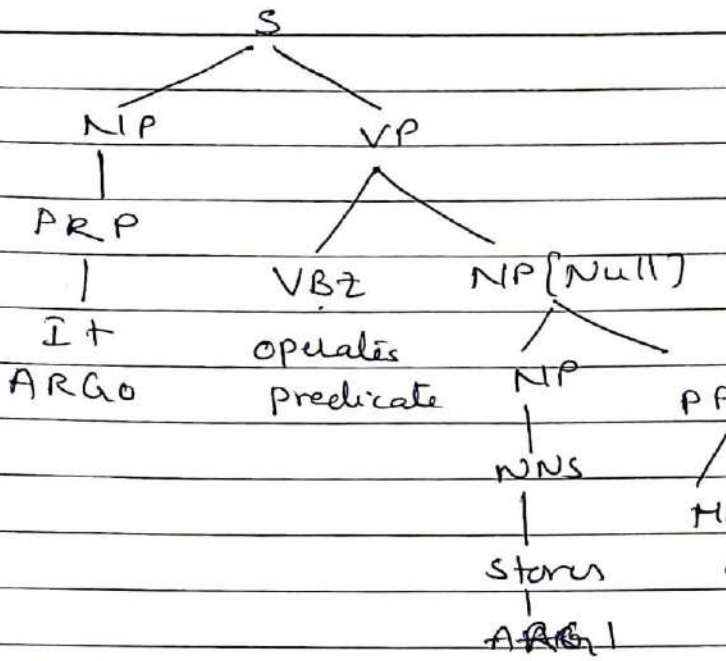


fig: Syntax tree for a sentence illustrating the propBank tags.

[ARG0 It] [predicate operates] [ARG1 stores]

[ARGM-LOC mostly in Iowa and Nebraska]

Distinction between frameNet and PropBank :-

In frameNet, there exists lexical units, which are words paired with their meanings or the frames that they invoke.

In propBank for each lemma, there is a list of different framesets that represent all the senses for which there is a different argument structure.

Systems :- Semantic Role labelling is a supervised classification problems that assumes the arguments of a predicate and the predicate itself can be mapped to a node in the syntax tree for that sentence. There are three tasks that could be used to evaluate

the system. They are

1. Argument Identification :- This is the task of identifying all and only the parse constituents that represent valid semantic arguments of a predicate.

2. Argument classification :- Given constituents known to represent arguments of a predicate, assign the appropriate argument labels to them.

3. Argument Identification and classification :- This task is a combination of the previous two tasks, where the constituents that represent arguments of a predicate are identified, and the appropriate argument label is assigned to them.

Once the sentence is parsed using a syntactic parser, each node in the parse tree can be classified as either one that represents the semantic argument (i.e., non-null node) or one that does not represent any semantic argument (i.e., a null node). The non-null nodes can be further classified into a set of argument labels.

For example in the above tree, the Noun phrase that encompasses stores mostly in Iowa and Nebraska is a null node because it does not correspond to a semantic argument. The node NP that encompasses stores is a non-null node because it does correspond to a semantic argument : ARG1.

The pseudo code for the generic semantic role labelling (SRL) algorithm is shown below.

The Semantic Role Labelling Algorithm:-

Procedure :- SRL (sentence) returns best semantic role labeling.

Input :- Syntactic parse of the sentence
1: generate a full syntactic parse of the sentence
2: identify all the predicates
3. for all predicate ∈ sentence do
4. extract a set of features for each node in the tree relative to the predicate.
5. classify each feature vector using the model created in training
6. select the class of highest scoring classifier.
7. return best semantic role labelling.
8. end for.

Before semantic representation ~~for~~ syntactic representation is required. PropBank was created as a layer of annotation on top of Penn Tree Bank-style phrase structure trees. FrameNet and PropBank used Phrase Structure Grammar (PSG), Combinatory Categorical Grammar (CCG). Dependency Trees, Base phrase Chunks

Learning Representation:-

Software :- following is a list of software packages available for semantic role labeling.
1. ASSERT :- Automatic Statistical semantic Role Tagger. A semantic role labeler trained on the English PropBank data

2. C-ASSERT : An extension of ASSERT for the Chinese language.
3. SwiRL :- A Wilher semantic role labeller trained on PropBank data
4. Shalmaneser :- A shallow Semantic Parser. A toolchain for shallow semantic parsing based on the FrameNet data.

Meaning Representation :-

Here we are discussing deeper level of semantic interpretation whose objective is to take natural language input and transform it into an unambiguous representation that a machine can act on. This is the form that would be more likely to be as incomprehensible to humans as it would be comprehensible to machines.

Although compilers and interpreters impose various specific syntactic and semantic restrictions on a program written in a high-level programming language, no such restrictions are imposed on the form that natural language can take; natural language relies on the recipient to disambiguate it using context and general world knowledge. However, there is still a lot of progress to be made, and so far the techniques that have been developed only work within specific domains and problems instead of being scalable to arbitrary domains. This is often termed deep semantic parsing, as opposed to shallow semantic parsing that comprises word sense disambiguation and semantic role labeling.

Resources :- A number of projects have created

representations and Resources that have promoted experimentation in this area.

ATIS:- The Air Travel Information System (ATIS) project is considered one of the first concerted efforts to build systems to transform natural language into a representation that could be used by an end application to make decisions. The task involved a machine to transform a user query in spontaneous speech, using a restricted vocabulary about flight information. It then formed a representation that was compiled into a SQL query, which was used to encode the intermediate semantic information.

Frame Representation

```
SHOW:
  FLIGHTS:
  TIME:
    PART-OF-DAY:
ORIGIN
    CITY: Boston
DEST:
    CITY: San Francisco
DATE:
    DAY-OF-WEEK: Tuesday
```

Natural Language Representation

Please show me morning flights from Boston to San Francisco on Tuesday.

Fig: A Sample user query and its frame representation in the ATIS Program.

Communicator: The communicator program was the follow-on to ATIS. While ATIS was more focused on user-initiated dialog (i.e., the user asked questions to the machine, which provided answers), Communicator involved a mixed-initiative dialog, whereby the human and machine had a dialog with each other with the computer presenting users with real-time travel information and helping them negotiate a preferred itinerary. Over the period of the program, many thousands of dialogs were collected and are available through the Linguistic Data Consortium. Carnegie-Mellon university collected more data, a portion of which is available for research. Roughly a million words and 1600 dialogs have been annotated with dialog acts.

GeoQuery :- In the domain of U.S. geography there is a natural language interface (NLI) to a geographic database called Geobase, which has about 800 Prolog facts stored in a relational database with geographic information such as population, neighboring states, major rivers, and major cities. Some sample queries and their representations are as follows.

1. What is the capital of the state with the largest population
answer(C, (capital(S, C), largest(P, (state(S), population(S, P)))))

2. what are the major cities in Kansas?
answer(C, (major(C), city(C), loc(C, S), equal(S, stateid(kansas))))

This is the GeoQuery corpus, which has also been translated into Japanese, Spanish and Turkish.

RoboCup : C Lang

RoboCup is an international initiative by the artificial intelligence community that uses robotic soccer as its domain. There is a special formal language, Clang, which is used to encode the advice from the team coach, and the behaviours are expressed as if-then rules. Following is an example representation in this domain:

(1) If the ball is in our penality area, all our players except player 4 should stay in our half.

((bpos (penality-area our)) (do (player-except our 4) (pos (half our))))

Systems :- As per the above applications, the meaning representation can be a SQL query, a Prolog query, or a domain-specific query representation.

Rule Based :- Some of the semantic parsing systems that performed very well for both the ~~AT~~ ATIS and Communicator projects were rule-based systems in the sense that they used an interpreter whose semantic grammar was handcrafted to be robust to speech recognition errors. The underlying semantic explanation of a sentence is much more complex than the underlying semantic information so parsing the meaning units in the sentence into semantics proved to be a better approach.

Especially in dealing with spontaneous speech, the system has to account for ungrammatical constructions, stutters, filled pauses and so on. Word order is therefore becomes less important, which leads to meaning units scattered in the sentences/utterances and not necessarily in the order that would make sense to a syntactic parser.

Supervised:- Although rule-based techniques are relatively easy to craft in the beginning and serve a good purpose to formulate solutions to various tasks, they have several downsides.

(i) they need some effort upfront to create the rules.

(ii) the time and specificity required to write rules usually restricts the development to systems that operate in limited domains.

(iii) they are hard to maintain and scale up as the problems becomes more complex and more domain independent

(iv) they tend to be brittle.

The alternative is to use statistical models derived from hand-annotated data. However, unless some hand-annotated data is available, statistical models cannot be used to deal with unknown phenomena.

During the ATIS evaluations, some data was hand-tagged for semantic information. they had four components in their system:

(i) Semantic parse (ii) Semantic frame

(iii) discourse (iv) backend

this system used a supervised learning approach

Combined with quick training augmentation through a human in-the-loop corrective approach to generate slightly lower quality but more data for improved supervision.

Software - Some rule based systems are
1. WASP 2. KRISPER 3. CHILL

Discourse Processing (Unit-5 Part-2)

Definition of Discourse: **Discourse** is the coherent structure of language above the level of sentences or clauses. A **discourse** is a coherent structured group of sentences.
What makes a passage coherent? A practical answer: **It has meaningful connections between its utterances.**

Cohesion

Relations between words in two units (sentences, paragraphs) "glue" them together.
Example: Before winter **I** built a chimney, and **shingled** the sides of my **house**… **I** have thus a tight **shingled** and plastered **house**.

There are Three Main Classes of Features for Discourse Cohesion
• Lexical overlap/lexical chains
• Coreference chains
• Cue words/discourse markers

Discourse Processing:

One of the major problems in NLP is discourse processing − building theories and models of how utterances stick together to form **coherent discourse**. Actually, the language always consists of collocated, structured and coherent groups of sentences rather than isolated and unrelated sentences like movies. These coherent groups of sentences are referred to as discourse.

Concept of Coherence

Coherence and discourse structure are interconnected in many ways. Coherence, along with property of good text, is used to evaluate the output quality of natural language generation system. The question that arises here is what does it mean for a text to be coherent? Suppose we collected one sentence from every page of the newspaper, then will it be a discourse? Of-course, not. It is because these sentences do not exhibit coherence. The coherent discourse must possess the following properties −

Coherence relation between utterances

The discourse would be coherent if it has meaningful connections between its utterances. This property is called coherence relation. For example, some sort of explanation must be there to justify the connection between utterances.

Discourse structure

An important question regarding discourse is what kind of structure the discourse must have. The answer to this question depends upon the segmentation we applied on discourse. Discourse segmentations may be defined as determining the types of structures for large discourse. It is quite difficult to implement discourse segmentation, but it is very important for **information retrieval, text summarization and information extraction** kind of applications.

Algorithms for Discourse Segmentation

In this section, we will learn about the algorithms for discourse segmentation. The algorithms are described below −

Unsupervised Discourse Segmentation

The class of unsupervised discourse segmentation is often represented as linear segmentation. We can understand the task of linear segmentation with the help of an example. In the example, there is a task of segmenting the text into multi-paragraph units; the units represent the passage of the original text. These algorithms are dependent on cohesion that may be defined as the use of certain linguistic devices to tie the textual units together. On the other hand, lexicon cohesion is the cohesion that is indicated by the relationship between two or more words in two units like the use of synonyms.

Supervised Discourse Segmentation

The earlier method does not have any hand-labeled segment boundaries. On the other hand, supervised discourse segmentation needs to have boundary-labeled training data. It is very easy to acquire the same. In supervised discourse segmentation, discourse marker or cue words play an important role. Discourse marker or cue word is a word or phrase that functions to signal discourse structure. These discourse markers are domain-specific.

Text Coherence

Lexical repetition is a way to find the structure in a discourse, but it does not satisfy the requirement of being coherent discourse. To achieve the coherent discourse, we must focus on coherence relations in specific. As we know that coherence relation defines the possible connection between utterances in a discourse. Hebb has proposed such kind of relations as follows −

We are taking two terms $\mathbf{S_0}$ and $\mathbf{S_1}$ to represent the meaning of the two related sentences −

Result

It infers that the state asserted by term $\mathbf{S_0}$ could cause the state asserted by $\mathbf{S_1}$. For example, two statements show the relationship result: Ram was caught in the fire. His skin burned.

Explanation

It infers that the state asserted by $\mathbf{S_1}$ could cause the state asserted by $\mathbf{S_0}$. For example, two statements show the relationship − Ram fought with Shyam's friend. He was drunk.

Parallel

It infers p(a1,a2,…) from assertion of $\mathbf{S_0}$ and p(b1,b2,…) from assertion $\mathbf{S_1}$. Here ai and bi are similar for all i. For example, two statements are parallel − Ram wanted car. Shyam wanted money.

Elaboration

It infers the same proposition P from both the assertions − **S_0** and **S_1** For example, two statements show the relation elaboration: Ram was from Chandigarh. Shyam was from Kerala.

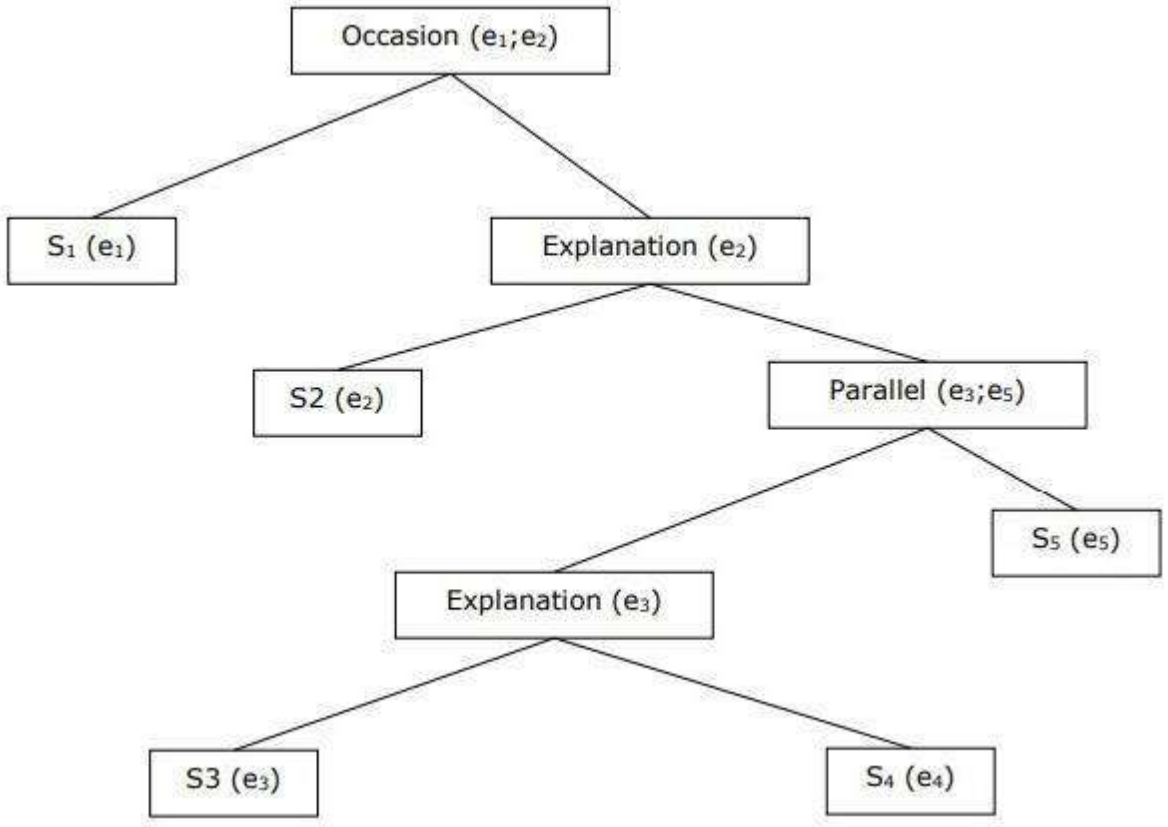
Occasion

It happens when a change of state can be inferred from the assertion of **S_0**, final state of which can be inferred from **S_1** and vice-versa. For example, the two statements show the relation occasion: Ram picked up the book. He gave it to Shyam.

Building Hierarchical Discourse Structure

The coherence of entire discourse can also be considered by hierarchical structure between coherence relations. For example, the following passage can be represented as hierarchical structure −

- **S_1** − Ram went to the bank to deposit money.
- **S_2** − He then took a train to Shyam's cloth shop.
- **S_3** − He wanted to buy some clothes.
- **S_4** − He do not have new clothes for party.
- **S_5** − He also wanted to talk to Shyam regarding his health



Reference Resolution

Interpretation of the sentences from any discourse is another important task and to achieve this we need to know who or what entity is being talked about. Here, interpretation reference is the key element. **Reference** may be defined as the linguistic expression to denote an entity or

individual. For example, in the passage, Ram, the manager of ABC bank, saw his friend Shyam at a shop. He went to meet him, the linguistic expressions like Ram, His, He are reference.

On the same note, **reference resolution** may be defined as the task of determining what entities are referred to by which linguistic expression.

Terminology Used in Reference Resolution

We use the following terminologies in reference resolution −

- **Referring expression** − The natural language expression that is used to perform reference is called a referring expression. For example, the passage used above is a referring expression.
- **Referent** − It is the entity that is referred. For example, in the last given example Ram is a referent.
- **Corefer** − When two expressions are used to refer to the same entity, they are called corefers. For example, ***Ram*** and ***he*** are corefers.
- **Antecedent** − The term has the license to use another term. For example, ***Ram*** is the antecedent of the reference ***he***.
- **Anaphora & Anaphoric** − It may be defined as the reference to an entity that has been previously introduced into the sentence. And, the referring expression is called anaphoric.
- **Discourse model** − The model that contains the representations of the entities that have been referred to in the discourse and the relationship they are engaged in.

Types of Referring Expressions

Let us now see the different types of referring expressions. The five types of referring expressions are described below −

Indefinite Noun Phrases

Such kind of reference represents the entities that are new to the hearer into the discourse context. For example − in the sentence Ram had gone around one day to bring him some food − some is an indefinite reference.

Definite Noun Phrases

Opposite to above, such kind of reference represents the entities that are not new or identifiable to the hearer into the discourse context. For example, in the sentence - I used to read The Times of India – The Times of India is a definite reference.

Pronouns

It is a form of definite reference. For example, Ram laughed as loud as he could. The word **he** represents pronoun referring expression.

Demonstratives

These demonstrate and behave differently than simple definite pronouns. For example, this and that are demonstrative pronouns.

Names

It is the simplest type of referring expression. It can be the name of a person, organization and location also. For example, in the above examples, Ram is the name-refereeing expression.

Reference Resolution Tasks

The two reference resolution tasks are described below.

Coreference Resolution

It is the task of finding referring expressions in a text that refer to the same entity. In simple words, it is the task of finding corefer expressions. A set of coreferring expressions are called coreference chain. For example - He, Chief Manager and His - these are referring expressions in the first passage given as example.

Constraint on Coreference Resolution

In English, the main problem for coreference resolution is the pronoun it. The reason behind this is that the pronoun it has many uses. For example, it can refer much like he and she. The pronoun it also refers to the things that do not refer to specific things. For example, It's raining. It is really good.

Pronominal Anaphora Resolution

Unlike the coreference resolution, pronominal anaphora resolution may be defined as the task of finding the antecedent for a single pronoun. For example, the pronoun is his and the task of pronominal anaphora resolution is to find the word Ram because Ram is the antecedent.