Contents

Chapter 7
Advanced Pragmatic Techniques and Specialized Topics in NLP

7.1 Why should you learn this chapter?

Pragmatic analysis in NLP delves in to understanding language in context. It goes beyond the literal meanings of words to interpret the intended message. Pragmatic analysis tries to pick up the correct speaker intent, and the social or situational context from the input speech or text while taking into account factors like speaker intent, tone, and the listener's perspective. Pragmatic analysis is essential for almost every serious NLP task including more common ones like sentiment analysis, chatbot development, and conversational AI. In short, pragmatic analysis is responsible for accurate interpretation of spoken or written text while considering the contexts of both leister and speaker.

Pragmatic analysis is responsible to make NLP applications to grasp the implied meanings, sarcasm, and indirect requests that are not directly conveyed by only the plain meanings of the words themselves but are understood through context., NLP applications integrated with pragmatic analysis are closely able to mimic human understanding. It leads to more natural and context-aware interactions between humans and machines (computers). Pragmatic analysis plays a pivotal role in NLP tasks dealing with dialogue systems and machine translation. Without learning pragmatic analysis, your aim of gaining any serious NLP skills is fa from complete.

So far, we have learnt in detail about lexical, syntactic analysis, and semantic analysis. Its natural for you to ask at this stage, how all of them relate with pragmatic analysis?

Lexical, syntactic, and semantic analyses are foundational steps that feed into pragmatic analysis in NLP. Lexical analysis is concerned with breaking down the text into words or tokens. After that, syntactic analysis step categorizes these tokens into grammatical structures to enable grammatically correct sentence formation. Semantic analysis interprets only the literal meaning of these sentence structures while concentrating on the relationships between words. Pragmatic analysis is the final step which gets input from the previous three layers and integrates context, speaker intent, and situational factors to correctly interpret the accurate, intended meaning of the text. Together, these four steps form a hierarchy as showed in the following figure (Figure 7.1). In this chapter itself, we will touch Disclosure Integration also along with the nuances of Pragmatic Analysis.

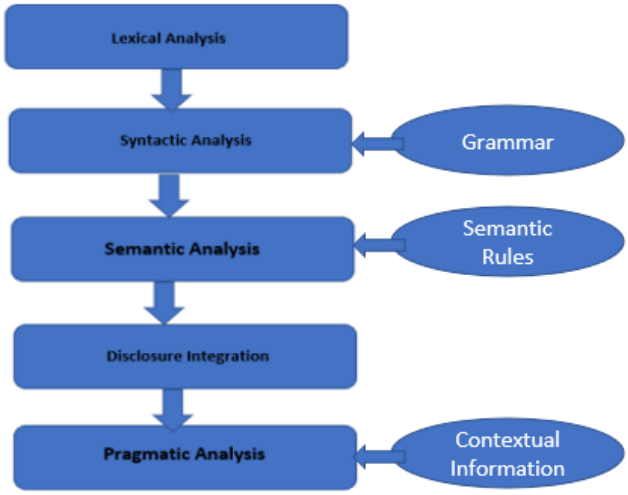

Figure 7.1 General process of most NLP applications[1]

7.2 Overview of Pragmatic Analysis Techniques

In the text below, we ate going to take up **key concepts in Pragmatic Analysis**. We will start with Distributional Semantics and Word Embeddings, which are helpful to recognize how words interact with each another in dynamic contexts. We will study Latent Semantic Analysis (LSA) along with its real-world applications practical applications to help us in the extraction of meaningful word relationships from large datasets. Word embeddings, such as Word2Vec, GloVe, CBOW, and Skip-gram, are fundamental in representing words in continuous vector spaces. BERT and GPT, further enhance word embeddings by integrating the surrounding context. It allows NLP algorithms to capture delicate shifts in word meanings. Topic Modelling and the techniques like Latent Dirichlet Allocation (LDA) are helpful in identifying underlying themes in text, which is pivotal to pragmatic analysis, especially while dealing with large corpora.

Finally, the techniques like Ensemble Methods and Word Similarity help us to further improve the effectiveness of pragmatic analysis. They combine different models and measure contextual word relationships. Towards the end of this section, we will introduce Coreference Resolution and its integration into NLP Pipelines to further establish the importance of recognising references.

7.3 Discourse Integration in NLP

In Discourse integration, we keep a track of an ongoing conversation to ensure that all the sentences fit together properly such that, as a collection, that make sense. In Discourse integration, the focus is not on understanding just one sentence at a time. Before trying to make any sense, we integrate all the sentences together and look how each sentence connects with others. After that we try to ensure that, together as a set, they make sense.

Let’s try to solidify this concept with an example of a virtual assistant (Like “Alexa” from Amazon). You instruct your virtual assistant, “Turn on the TV” and then say, “Also, make it louder.” The second sentence doesn’t explicitly use the word "TV," bot the assistant still analyses what you are talking about. This is Discourse Integration; we are trying to make the sense of this short conversation in

totality. Without this, the assistant might get confused as we didn't mention "TV" in the second sentence.

In real life, we use discourse integration all the time. If your friend says, "I'm hungry" and then, "Let's get pizza," you know both sentences are related. In a similar fashion, Discourse integration techniques help machines to follow and respond to conversations, just like humans. We often use pronouns ("he," "she," "it") in our regular conversations. Discourse integration techniques used in NLP application help to resolve, how these pronouns are linked to the correct noun previously mentioned in the discourse. It also helps to interpret implied meanings and how context dynamically changes during a conversation. In another example, in automated customer service applications (ChatBots), discourse integration helps the application to track the user's queries even over long conversations and respond in a relevant manner.

7.3.1 Techniques for Discourse Integration

The key techniques under Discourse Integration include Coreference resolution, anaphora resolution, coherence modelling, rhetorical structure theory (RST), discourse parsing, entity tracking, lexical chains, topic modelling, cohesion analysis, and temporal relation identification.

Coreference resolution enables NLP applications to connect the dots between words and their referents. For example, it ensures that "he," "she," or "it" connect to the right person or thing. For instance, in a news article, if it says, "Mary entered the class. She sat down," coreference resolution's job is to link "She" back to Mary. Anaphora resolution, on the other hand, zooms in on backward references. It resolves phrases like "this" or "that" to their earlier mentions. For example, in the sentence, "I bought a new car. This is my favourite machine," anaphora resolution links "this" to "car."

Coherence modelling is like an invisible thread that helps keep a conversation or story connected and flowing smoothly. Imagine you're reading a book, and one chapter talks about a character going on an adventure. Then, in the next chapter, it talks about the challenges the character faces. Coherence modelling helps link those chapters together so you can follow the journey without getting lost. For example, in a chatbot, if you ask, "What's the weather today?" and then, "Should I bring an umbrella?" coherence modelling helps the system understand that both questions are related to the weather.

Let's take the next technique, straight with an example. In an essay, if the opening paragraph describes the dangers of climate change and further explains possible solutions. Rhetorical Structure Theory (RST) reveals that the second part is describing for the first. RST delves to explain us how different parts of a conversation or text are related. RST role comes into play in explaining the language constructs like cause and effect, contrast, or explanation. Let's take another case of a conversation between two friends. The first person might ask a question, and the other gives an answer. Discourse parsing is the technique that comes in to play in separating these into "question" and "answer" units. Discourse parsing is equivalent to generating a blueprint of a conversation or text. It breaks the conversation or input text down into smaller, meaningful units to see how they work together. There are a couple of more such techniques related to Discourse Integration, which are useful if you are aware of them. They are explained below in brief. We will try to link each technique with a real-world example for better understanding.

Entity tracking deals with keeping a track of nouns, like people or things, throughout a story. For example, in a suspense story, it helps if you properly remember all the suspects and their actions that are mentioned in previous chapters. So, if the first chapter talks about a suspect named John and another talks about Alex's friend, Mary. In this example, the job of entity tracking is to connect these details throughout the story so that machines have a clear understanding of story.

Let's take a different use case. You are reading a story about a family dog. Lexical chains techniques are used to connect all the related words like "dog," "puppy," and "pet" that may have mentions different parts of the book. Keeping this link will help you to follow along without getting confused about what's being talked about. Lexical chains link connected words throughout the speech or input text to make sure everything sticks to the same topic. We will use another example top discuss topic modelling technique. Suppose, you're reading a few news articles online. Topic modelling can group all the stories about climate change together. This can help you to easily find and read all the articles about the same topic.

In another instance, if an article starts with talking about a problem and then changeovers to a solution, cohesion analysis is the technique that helps machine to make sure the changeover is clear and logical. Finally, Temporal relation identification helps to arrange events (in speech or text) in the order they occurred. In a history book, it helps to place events like wars and revolutions in chronological order. For example, it would show that World War I came before World War II.

We discussed quite a few techniques in this section. We may not detail each one of them in detail in the following ones for space constraints.

7.3.2 Demonstrating Techniques for Discourse Integration

Below we pick-up all the techniques and use python codes to demonstrate these processes.

7.3.2.1 Coreference resolution (code demo)

- Most of these techniques use popular NLP libraries in Python, such as spaCy, NLTK, Transformers, Gensim, and others.
- Here's a brief idea of how each technique can be demonstrated with code.

```
# Install the package needed for coreference resolution.
# Uncomment it if coreferee is not installed on your system.
!python -m pip install coreferee

# Installs the English coreference resolution model for the coreferee
package.
!python -m coreferee install en

# Download the large English model ('en_core_web_lg') for spaCy.
# This model includes word vectors and is more accurate for NER and POS
tagging.
# It's larger and more powerful than the smaller models lke
'en_core_web_sm'), but it may take up large memory.
# We have not loaded it in the main code body but without this step, the
code was throwing error.
# So it's a required step.
!python -m spacy download en_core_web_lg
```

```
# Code to ignore warnings.
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)

# Import the required package.
import spacy
# Load the transformer-based English model.
# As of now let's use it. We will take it up in detail in the later
chapters.
nlp = spacy.load('en_core_web_trf')

"""
Add 'coreferee' to the pipeline.
'coreferee' is a spaCy extension that enables the identification and
linking of coreferences in a text.
Adding it to the pipeline so that coreference resolution can be performed
after other NLP tasks like tokenization and parsing.
"""
nlp.add_pipe('coreferee')
# Create the sample text for demo.
sample_doc = nlp(""" Although she was very busy at office work, Mary felt
she had had enough of it.
                    She and her spouse decided they needed to go on a
holiday.
                    They travelled by train to France because they had
enough friends in the country.""")
print("")
print("OUTPUT \n")
sample_doc._.coref_chains.print()
```

```
OUTPUT

0: she(2), Mary(10), she(12), She(20), her(22)
1: work(8), it(17)
2: [She(20); spouse(23)], they(25), They(34), they(41)
3: France(39), country(47)
```

The output above does not look that easy to understand. The first line of the production (indexed 0) tells us that the pronouns, she(2), she(12), She(20), and her(22) refer to the same name, 'Mary(10).' Similarly, the second line (indexed 1) makes us understand that it(17) stands for work(8). At index 2 position, they(25), They(34), they(41) refer to [She(20); spouse(23)]. Finally, at index 3, country(47) stands for France(39). You will notice that in this output all the pronouns in the sample_doc are resolved to their proper nouns (coreference resolution process).

7.3.2.2 Rhetorical structure theory (RST) (code demo)

```
!pip install stanza

import stanza

# Download and set up the Stanford NLP mode.
stanza.download('en')
```

```
{"model_id":"b1022cc610b04d679f12c5211b381398","version_major":2,"version_
minor":0}

2024-09-08 16:28:37 INFO: Downloaded file to C:\Users\Shailendra
Kadre\stanza_resources\resources.json
2024-09-08 16:28:37 INFO: Downloading default packages for language: en
(English) ...
2024-09-08 16:28:38 INFO: File exists: C:\Users\Shailendra
Kadre\stanza_resources\en\default.zip
2024-09-08 16:28:40 INFO: Finished downloading models and saved to
C:\Users\Shailendra Kadre\stanza_resources

# Initialize a Stanza Pipeline for processing English text.
nlp = stanza.Pipeline(lang='en', processors='tokenize,mwt,pos,depparse,
lemma')

2024-09-08 16:30:35 INFO: Checking for updates to resources.json in case
models have been updated.  Note: this behavior can be turned off with
download_method=None or download_method=DownloadMethod.REUSE_RESOURCES

{"model_id":"aaa69eecc06847fc9c18c0cbd0e1c88f","version_major":2,"version_
minor":0}

2024-09-08 16:30:36 INFO: Downloaded file to C:\Users\Shailendra
Kadre\stanza_resources\resources.json
2024-09-08 16:30:36 INFO: Loading these models for language: en (English):
=================================
| Processor | Package           |
---------------------------------
| tokenize  | combined          |
| mwt       | combined          |
| pos       | combined_charlm   |
| lemma     | combined_nocharlm |
| depparse  | combined_charlm   |
=================================

2024-09-08 16:30:36 INFO: Using device: cpu
2024-09-08 16:30:36 INFO: Loading: tokenize
2024-09-08 16:30:36 INFO: Loading: mwt
2024-09-08 16:30:36 INFO: Loading: pos
2024-09-08 16:30:36 INFO: Loading: lemma
2024-09-08 16:30:36 INFO: Loading: depparse
2024-09-08 16:30:37 INFO: Done loading processors!

# Process a text
doc = nlp("Your example sentence goes here.")
for sentence in doc.sentences:
    print("Tokens:", [word.text for word in sentence.words])
    print("POS Tags:", [word.pos for word in sentence.words])
    print("Dependencies:", [(word.text, word.deprel) for word in
sentence.words])

Tokens: ['Your', 'example', 'sentence', 'goes', 'here', '.']
POS Tags: ['PRON', 'NOUN', 'NOUN', 'VERB', 'ADV', 'PUNCT']
Dependencies: [('Your', 'nmod:poss'), ('example', 'compound'),
```

```
('sentence', 'nsubj'), ('goes', 'root'), ('here', 'advmod'), ('.',
'punct')]

# Create the sample text for demo.
sample_doc = nlp(""" Although she was very busy at office work, Mary felt
she had had enough of it.
                     She and her spouse decided they needed to go on a
holiday.
                     They travelled by train to France because they had
enough friends in the country.""")

# Process the text
doc = nlp(sample_doc)

# Print out tokens, POS tags, and dependency relations.
for sentence in doc.sentences:
    print("Sentence:", " ".join([word.text for word in sentence.words]))
    print("Tokens:", [word.text for word in sentence.words])
    print("POS Tags:", [word.pos for word in sentence.words])
    print("Dependencies:", [(word.text, word.deprel) for word in
sentence.words])
    print("-----")

Sentence: Although she was very busy at office work , Mary felt she had
had enough of it .
Tokens: ['Although', 'she', 'was', 'very', 'busy', 'at', 'office', 'work',
',', 'Mary', 'felt', 'she', 'had', 'had', 'enough', 'of', 'it', '.']
POS Tags: ['SCONJ', 'PRON', 'AUX', 'ADV', 'ADJ', 'ADP', 'NOUN', 'NOUN',
'PUNCT', 'PROPN', 'VERB', 'PRON', 'AUX', 'VERB', 'ADJ', 'ADP', 'PRON',
'PUNCT']
Dependencies: [('Although', 'mark'), ('she', 'nsubj'), ('was', 'cop'),
('very', 'advmod'), ('busy', 'advcl'), ('at', 'case'), ('office',
'compound'), ('work', 'obl'), (',', 'punct'), ('Mary', 'nsubj'), ('felt',
'root'), ('she', 'nsubj'), ('had', 'aux'), ('had', 'ccomp'), ('enough',
'obj'), ('of', 'case'), ('it', 'obl'), ('.', 'punct')]
-----
Sentence: She and her spouse decided they needed to go on a holiday .
Tokens: ['She', 'and', 'her', 'spouse', 'decided', 'they', 'needed', 'to',
'go', 'on', 'a', 'holiday', '.']
POS Tags: ['PRON', 'CCONJ', 'PRON', 'NOUN', 'VERB', 'PRON', 'VERB',
'PART', 'VERB', 'ADP', 'DET', 'NOUN', 'PUNCT']
Dependencies: [('She', 'nsubj'), ('and', 'cc'), ('her', 'nmod:poss'),
('spouse', 'conj'), ('decided', 'root'), ('they', 'nsubj'), ('needed',
'ccomp'), ('to', 'mark'), ('go', 'xcomp'), ('on', 'case'), ('a', 'det'),
('holiday', 'obl'), ('.', 'punct')]
-----
Sentence: They travelled by train to France because they had enough
friends in the country .
Tokens: ['They', 'travelled', 'by', 'train', 'to', 'France', 'because',
'they', 'had', 'enough', 'friends', 'in', 'the', 'country', '.']
POS Tags: ['PRON', 'VERB', 'ADP', 'NOUN', 'ADP', 'PROPN', 'SCONJ', 'PRON',
'VERB', 'ADJ', 'NOUN', 'ADP', 'DET', 'NOUN', 'PUNCT']
Dependencies: [('They', 'nsubj'), ('travelled', 'root'), ('by', 'case'),
('train', 'obl'), ('to', 'case'), ('France', 'obl'), ('because', 'mark'),
```

```
('they', 'nsubj'), ('had', 'advcl'), ('enough', 'amod'), ('friends',
'obj'), ('in', 'case'), ('the', 'det'), ('country', 'nmod'), ('.',
'punct')]
-----
```

We will take the middle sentence for elaborations on the output.

Sentence"She and her spouse decided they needed to go on a holiday."

Tokens and POS Tags: Tokens: ['She', 'and', 'her', 'spouse', 'decided', 'they', 'needed', 'to', 'go', 'on', 'a', 'holiday', '.'] POS Tags: ['PRON', 'CCONJ', 'PRON', 'NOUN', 'VERB', 'PRON', 'VERB', 'PART', 'VERB', 'ADP', 'DET', 'NOUN', 'PUNCT'] PRON (Pronoun): 'She', 'her', 'they' CCONJ (Coordinating Conjunction): 'and' NOUN (Noun): 'spouse', 'holiday' VERB (Verb): 'decided', 'needed', 'go' PART (Particle): 'to' ADP (Adposition): 'on' DET (Determiner): 'a' PUNCT (Punctuation): '.'

Dependencies: ('She', 'nsubj'): 'She' is the nominal subject of the main verb 'decided'. ('and', 'cc'): 'and' is a coordinating conjunction linking 'She' and 'her spouse'. ('her', 'nmod:poss'): 'her' is a possessive modifier for 'spouse'. ('spouse', 'conj'): 'spouse' is a conjunct connected to 'She' by 'and'. ('decided', 'root'): 'decided' is the main verb (root) of the sentence. ('they', 'nsubj'): 'they' is the subject of the embedded verb 'needed'. ('needed', 'ccomp'): 'needed' is a clausal complement of 'decided'. ('to', 'mark'): 'to' is a marker for the infinitive verb 'go'. ('go', 'xcomp'): 'go' is an open clausal complement of 'needed'. ('on', 'case'): 'on' is a preposition marking the case of 'holiday'. ('a', 'det'): 'a' is a determiner for 'holiday'. ('holiday', 'obl'): 'holiday' is the oblique object of the preposition 'on'. ('.', 'punct'): '.' is punctuation marking the end of the sentence.

Having this information in your folds, you can now apply RST principles and manually complete the the process of RST. More advanced tools are available to complete the RST process for you. Stanza provides valuable syntactic information. However, an end-to-end RST analysis would need additional steps or tools to explicitly categorize and recognise the rhetorical relationships between different text parts. To this date, a Python library for RST is unavailable to our knowledge.

7.3.2.3 Discourse parsing (code demo)

A direct discourse parsing library is currently unavailable in Python. Professionals who write code often use advanced NLP libraries like spaCy or AllenNLP, along with deep learning techniques. We will stick to spaCy for a simplified demo. We will base our code on the following logic.

1. Load SpaCy Model
   - Initialize SpaCy's English language model
1. Define Sample Text
   - Set a string variable with the sample text
2. Process Text
   - Use SpaCy to analyze the sample text
   - Split the text into sentences
3. Define Function to Extract Discourse Information
   - For each sentence in the processed text:
     a. Print the sentence

b. Extract and print named entities (if any)

c. Extract and print syntactic dependencies for each token

d. Infer basic discourse relations based on keywords:

- If the sentence contains the word "because":
  - Record that this sentence provides a reason for the previous sentence
- If the sentence contains the word "although":
  - Record that this sentence contrasts with the previous sentence

4. Call Function to Extract and Display Discourse Information
   - Print inferred discourse relations

```
!python -m spacy download en_core_web_sm
```

```
import spacy

# Load SpaCy's English model.
nlp = spacy.load('en_core_web_sm')

# Sample text.
text = """
Although she was very busy with office work, Mary felt she had had enough
of it.
She and her spouse decided they needed to go on a holiday.
They travelled by train to France because they had enough friends in the
country.
"""

# Process the text with SpaCy.
doc = nlp(text)

# Function to extract and display discourse-like information
def extract_discourse_info(doc):
    sentences = list(doc.sents)
    relations = []

    for i, sent in enumerate(sentences):
        print(f"Sentence {i+1}: {sent.text}")

        # Extract named entities.
        entities = [(ent.text, ent.label_) for ent in sent.ents]
        if entities:
            print("  Named Entities:", entities)

        # Extract syntactic dependencies
        dependencies = [(token.text, token.dep_, token.head.text) for
token in sent]
        print("  Dependencies:", dependencies)

        # Basic inference of discourse relations.
        if i > 0:
            previous_sent = sentences[i-1]
```

```
            if "because" in sent.text.lower():
                relations.append(f"Sentence {i+1} provides a reason for
Sentence {i}")
            elif "although" in sent.text.lower():
                relations.append(f"Sentence {i+1} contrasts with Sentence
{i}")

    return relations

# Extract and print discourse information.
relations = extract_discourse_info(doc)
print("\nInferred Discourse Relations:")
for relation in relations:
    print(relation)
```

```
Sentence 1:

  Dependencies: [('\n', 'dep', '\n')]
Sentence 2: Although she was very busy with office work, Mary felt she had
had enough of it.

  Named Entities: [('Mary', 'PERSON')]
  Dependencies: [('Although', 'mark', 'was'), ('she', 'nsubj', 'was'),
('was', 'advcl', 'felt'), ('very', 'advmod', 'busy'), ('busy', 'acomp',
'was'), ('with', 'prep', 'busy'), ('office', 'compound', 'work'), ('work',
'pobj', 'with'), (',', 'punct', 'felt'), ('Mary', 'nsubj', 'felt'),
('felt', 'ROOT', 'felt'), ('she', 'nsubj', 'had'), ('had', 'aux', 'had'),
('had', 'ccomp', 'felt'), ('enough', 'dobj', 'had'), ('of', 'prep',
'enough'), ('it', 'pobj', 'of'), ('.', 'punct', 'felt'), ('\n', 'dep',
'.')]
Sentence 3: She and her spouse decided they needed to go on a holiday.

  Named Entities: [('a holiday', 'DATE')]
  Dependencies: [('She', 'nsubj', 'decided'), ('and', 'cc', 'She'),
('her', 'poss', 'spouse'), ('spouse', 'conj', 'She'), ('decided', 'ROOT',
'decided'), ('they', 'nsubj', 'needed'), ('needed', 'ccomp', 'decided'),
('to', 'aux', 'go'), ('go', 'xcomp', 'needed'), ('on', 'prep', 'go'),
('a', 'det', 'holiday'), ('holiday', 'pobj', 'on'), ('.', 'punct',
'decided'), ('\n', 'dep', '.')]
Sentence 4: They travelled by train to France because they had enough
friends in the country.

  Named Entities: [('France', 'GPE')]
  Dependencies: [('They', 'nsubj', 'travelled'), ('travelled', 'ROOT',
'travelled'), ('by', 'prep', 'travelled'), ('train', 'pobj', 'by'), ('to',
'prep', 'travelled'), ('France', 'pobj', 'to'), ('because', 'mark',
'had'), ('they', 'nsubj', 'had'), ('had', 'advcl', 'travelled'),
('enough', 'amod', 'friends'), ('friends', 'dobj', 'had'), ('in', 'prep',
'friends'), ('the', 'det', 'country'), ('country', 'pobj', 'in'), ('.',
'punct', 'travelled'), ('\n', 'dep', '.')]

Inferred Discourse Relations:
```

```
Sentence 2 contrasts with Sentence 1
Sentence 4 provides a reason for Sentence 3
```

Discourse parsing deals with how different parts of a text are related to one another. It can be through logical and communicative connections.The main part of the output is the "discourse information." In the following input text,

Below is the input text, labelled by sentence numbers: (Sentence 1) Although she was very busy with office work, (Sentence 2) Mary felt she had had enough of it. (Sentence 3) She and her spouse decided they needed to go on a holiday. (Sentence 4) They travelled by train to France because they had enough friends in the country.

The following is the summary of Discourse Relations

- Contrasting Relation: Sentence 2 seems to contrast with Sentence 1. The word "Although" in Sentence 1 talks about a contrast with the decision pronounced in Sentence 2.
- Reason Relation: Sentence 4 seems to give a reason for the decision made in Sentence 3. The word "because" specifies that Sentence 4 explains why they travelled to France.

7.3.2.4 Entity tracking (code demo)

Entity tracking deals with keeping track of the entities (like people, places, or objects) mentioned throughout a text. It's like keeping track of characters throughout a novel. Here's a simple code demo using spaCy for Named Entity Recognition (NER) and tracking those entities.

```
import spacy

# Load the English NLP model
nlp = spacy.load('en_core_web_sm')

# Sample text.
text = """
Although she was very busy with office work, Mary felt she had had enough
of it.
She and her spouse decided they needed to go on a holiday.
They travelled by train to France because they had enough friends in the
country.
"""

# Process the text with spaCy
doc = nlp(text)

# Dictionary to track entities
entity_tracking = {}

# Iterate through the sentences in the doc
for sent in doc.sents:
    print(f"Sentence: {sent}")
    # Iterate through named entities in the sentence
    for ent in sent.ents:
```

```
        print(f"Entity: {ent.text}, Label: {ent.label_}")
        # Track entities and update if seen again
        if ent.text in entity_tracking:
            entity_tracking[ent.text] += 1
        else:
            entity_tracking[ent.text] = 1

# Output tracked entities
print("\nEntity Tracking:")
for entity, count in entity_tracking.items():
    print(f"{entity}: mentioned {count} times")
```

```
Sentence:

Sentence: Although she was very busy with office work, Mary felt she had
had enough of it.

Entity: Mary, Label: PERSON
Sentence: She and her spouse decided they needed to go on a holiday.

Entity: a holiday, Label: DATE
Sentence: They travelled by train to France because they had enough
friends in the country.

Entity: France, Label: GPE

Entity Tracking:
Mary: mentioned 1 times
a holiday: mentioned 1 times
France: mentioned 1 times
```

More professional code for entity tracking is written using packages like spaCy, AllenNLP, or Hugging Face Transformers. Entity code functions first detect, classify, and link entities across texts. These systems make use of NER and Coreference Resolution to track entities even when they are mentioned by pronouns or synonyms. Professional code is frequently a part of larger NLP pipelines that integrate with databases or knowledge graphs to accomplish entity relationships and maintain the required accuracy over long input documents or conversations.

7.3.2.5 Lexical chains (code demo)

A lexical chain is a sequence of related words that are connected either through direct synonyms or semantically related terms. These related words share a common meaning or topic. We discussed lexical chains with an example in the theory section. In this book, we will provide a simple code demo of lexical chains using WordNet from the nltk library.

For professional NLP applications, lexical chain code involves advanced algorithms that may be based on WordNet, distributional semantics, or word embeddings like Word2Vec or BERT to pick up semantic relationships between words. Such systems make use of synonyms, hypernyms, and context-based similarities to arrive at accurate lexical chains. This code is often united with text segmentation, word sense

disambiguation, and coherence modelling for tasks involving summarization or topic detection.

```
# !pip install nltk
# !python -m nltk.downloader wordnet

import nltk
from nltk.corpus import wordnet as wn

# Sample text
text = "Anil is a blood student in my class. He runs very fast."

# Tokenize the text
words = nltk.word_tokenize(text.lower())

# Function to find synonyms from WordNet
def get_synonyms(word):
    synonyms = set()
    for syn in wn.synsets(word):
        for lemma in syn.lemmas():
            synonyms.add(lemma.name())
    return synonyms

# Build lexical chains
lexical_chains = []

for word in words:
    found_chain = False
    word_synonyms = get_synonyms(word)

    # Check if word fits into any existing chain
    for chain in lexical_chains:
        if chain.intersection(word_synonyms):
            chain.update(word_synonyms)
            found_chain = True
            break

    # If not, start a new chain
    if not found_chain and word_synonyms:
        lexical_chains.append(set(word_synonyms))

# Output lexical chains
for i, chain in enumerate(lexical_chains):
    print(f"Chain {i + 1}: {chain}")
```

```
Chain 1: {'indigo', 'indigotin', 'Indigofera_suffruticosa', 'anil',
'Indigofera_anil'}
Chain 2: {'personify', 'represent', 'be', 'equal', 'follow', 'live',
'cost', 'constitute', 'embody', 'comprise', 'make_up', 'exist'}
Chain 3: {'type_A', 'adenine', 'ampere', 'a', 'angstrom_unit',
'antiophthalmic_factor', 'vitamin_A', 'angstrom', 'axerophthol', 'amp',
'group_A', 'deoxyadenosine_monophosphate', 'A'}
Chain 4: {'debauched', 'lineage', 'line', 'riotous', 'blood',
'blood_line', 'fast', 'degenerate', 'quick', 'bloodline', 'dissolute',
```

```
'flying', 'libertine', 'tight', 'fasting', 'firm', 'parentage', 'rake',
'ancestry', 'degraded', 'loyal', 'profligate', 'rip', 'rakehell', 'stock',
'dissipated', 'immobile', 'stemma', 'pedigree', 'descent', 'truehearted',
'line_of_descent', 'roue', 'origin'}
Chain 5: {'scholarly_person', 'bookman', 'pupil', 'educatee', 'student',
'scholar'}
Chain 6: {'Indiana', 'atomic_number_49', 'In', 'IN', 'inward',
'Hoosier_State', 'inch', 'inwards', 'indium', 'in'}
Chain 7: {'execute', 'division', 'incline', 'family', 'outpouring',
'bleed', 'runnel', 'consort', 'running_play', 'bunk', 'pass', 'persist',
'social_class', 'be_given', 'campaign', 'run_for', 'escape', 'hunt_down',
'discharge', 'running_game', 'ply', 'lam', 'guide', 'trial',
'hightail_it', 'classify', 'category', 'go', 'lead', 'head_for_the_hills',
'rivulet', 'sort', 'rill', 'lean', 'carry', 'unravel', 'turn_tail',
'course_of_study', 'melt_down', 'move', 'streak', 'separate', 'foot_race',
'political_campaign', 'ravel', 'fly_the_coop', 'stratum', 'scat', 'race',
'running', 'course_of_instruction', 'melt', 'run_away', 'scarper',
'streamlet', 'function', 'work', 'endure', 'draw', 'hunt', 'test',
'black_market', 'tally', 'operate', 'sort_out', 'track_down', 'play',
'run', 'break_away', 'tend', 'ladder', 'form', 'grade', 'feed',
'die_hard', 'footrace', 'class', 'flow', 'assort', 'extend',
'take_to_the_woods', 'socio-economic_class', 'course', 'year', 'prevail',
'range'}
Chain 8: {'atomic_number_2', 'helium', 'He', 'he'}
Chain 9: {'very', 'real', 'really', 'selfsame', 'rattling', 'identical'}
```

The output characterizes lexical chains as groups of semantically related words from the input text. Sometimes you see a couple of unrelated terms (not given in the input text) in these lexical chains. This is an example of how lexical chaining can bring in terms that aren't explicitly present but are linked conceptually in the word database. The algorithm sometimes needs fine-tuning to limit the chain generation to the terms that more closely match your input text.

7.3.2.6 Topic modelling (code demo)

Topic modelling techniques use word patterns and groupings to identify the key themes or topics in a large collection of texts. In this chapter, we will present a simple demo of topic modelling using Tf-Idf technique. We talked about this technique in detail in the earlier chapters. Professional topic modelling code is often written using advanced techniques like Latent Dirichlet Allocation (LDA) or Non-Negative Matrix Factorization (NMF). It leverages Python packages like Gensim or scikit-learn. We will take up LDA in detail in the upcoming chaptes.

We will follow these five simple steps to write our code.

- Prepare Documents: Start with a list of text documents you want to analyse.
- Initialize Vectorizer: Create a TfidfVectorizer object. A TfIdf score will convert the input text documents into numerical data. This process ignores stop words.
- Transform Text: Use the vectorizer to convert the input text documents into a Tf-Idf matrix.
- Get Feature Names: Extract the list of words that were considered in the TF-IDF analysis.

- Display Scores: For each document, print the words and their Tf-Idf scores.

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Sample text data (documents)

documents =   ["Although she was very busy with office work, Mary felt she
had had enough of it.",
               "She and her spouse decided they needed to go on a
holiday.",
               "They travelled by train to France because they had enough
friends in the country."]


# Initialize the TF-IDF Vectorizer
tfidf_vectorizer = TfidfVectorizer(stop_words='english')

# Fit and transform the documents into a TF-IDF matrix
tfidf_matrix = tfidf_vectorizer.fit_transform(documents)

# Get the feature names (terms) from the TF-IDF model
feature_names = tfidf_vectorizer.get_feature_names_out()

# Display the TF-IDF scores for each document
for doc_idx, doc in enumerate(documents):
    print(f"\nDocument {doc_idx + 1}: {doc}")
    # Get the TF-IDF scores for each word in the document
    for word_idx in tfidf_matrix[doc_idx].nonzero()[1]:
        print(f"{feature_names[word_idx]}: {tfidf_matrix[doc_idx,
word_idx]:.4f}")
```

```
Document 1: Although she was very busy with office work, Mary felt she had
had enough of it.
felt: 0.4472
mary: 0.4472
work: 0.4472
office: 0.4472
busy: 0.4472

Document 2: She and her spouse decided they needed to go on a holiday.
holiday: 0.5000
needed: 0.5000
decided: 0.5000
spouse: 0.5000

Document 3: They travelled by train to France because they had enough
friends in the country.
country: 0.4472
friends: 0.4472
france: 0.4472
train: 0.4472
travelled: 0.4472
```

The output talk about the TF-IDF scores for the words in Document 1, 2, and 3. The higher the score, the more relevant the word is to the content of the document. In the case of document 2, each of these words has a high score of 0.5000. It means all the words are equally important in this document. If a word has a high TF-IDF score, it means that the word is exclusive to that document and it could be a strong pointer of its topic or content. Note that this was a simplified analysis. better results can be obtained using advanced techniques like LDA.

7.3.2.7 Cohesion analysis (code demo)

Cohesion analysis has two objectives. First, it looks at how well different parts of a text fit together. And second, how they connect to make the text flow smoothly and make sense. In our below demo, we are bringing in a new concept of cosine similarity, which measures the similarity between two vectors. An interpretation of the similarity scores is given below. We will cover this concept in detail in the later pages of this chapter.

We have used a basic TF-Idf technique in our code demo. More advanced vectorization techniques used by professionals for cohesion insight include word embeddings like Word2Vec and GloVe to capture more nuanced insights into text similarity and cohesion. We will take up these techniques later in this chapter.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# Revised sentences
sentences = [
    "Mary is feeling overwhelmed with her busy office job but finds some
relief in her evening walks.",
    "Mary feels overwhelmed with her hectic work schedule, yet she finds
relaxation in her daily evening walks.",
    "Despite a busy workday, Mary enjoys unwinding with a walk in the
evening."
]

# Initialize the TF-IDF Vectorizer
vectorizer = TfidfVectorizer(stop_words='english')

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

# Compute the cosine similarity matrix
cosine_sim_matrix = cosine_similarity(tfidf_matrix)

# Display the cosine similarity matrix
print("Cosine Similarity Matrix:")
for i in range(len(sentences)):
    for j in range(len(sentences)):
        print(f"Similarity between Sentence {i + 1} and Sentence {j + 1}:
{cosine_sim_matrix[i, j]:.2f}")
```

```
Cosine Similarity Matrix:
Similarity between Sentence 1 and Sentence 1: 1.00
Similarity between Sentence 1 and Sentence 2: 0.32
```

```
Similarity between Sentence 1 and Sentence 3: 0.19
Similarity between Sentence 2 and Sentence 1: 0.32
Similarity between Sentence 2 and Sentence 2: 1.00
Similarity between Sentence 2 and Sentence 3: 0.10
Similarity between Sentence 3 and Sentence 1: 0.19
Similarity between Sentence 3 and Sentence 2: 0.10
Similarity between Sentence 3 and Sentence 3: 1.00
```

Interpretation of the output:

- Interpreting the scores: Higher scores specify more cohesive sentences with shared vocabulary and similar content.
- Low Inter-Sentence Similarity: The highest is 0.32. It shows limited shared content or vocabulary.
- Minimal Overlap: Low similarity scores of 0.10 and 0.19 indicate minimal thematic or lexical overlap between the sentences.
- A similarity score of 1.00 indicates perfect similarity with itself. Similarity scores of 0 indicate no shared vocabulary and similarity of content between sentences.

7.3.2.8 Temporal relation identification (code demo)

Temporal relation identification is the process of finding time-based relationships between events in an input text. For instance, take the sentence "I want to complete this chapter before lunch;" the word " before" indicates the time-based connection between two events. It talks about finishing work earlier than lunch.

Our code demo for temporal relation identification would first perform dependency parsing for extracting events and detect temporal signals, which are swords like "before," "after," "during," "until," and "while.". After this, the classification of events will be done through rule-based methods or machine-learning models. This approach aims to find the sequence and timing of actions in a given input text. We will utilize spaCy to extract events.

Our program below follows these five steps

- Import Libraries and Load Model
- Feature Extraction from Sentences
- Prepare Training Data
- Train Random Forest Classifier
- Predict Temporal Relations in New Sentences

```
# Required Libraries
import spacy
from sklearn.ensemble import RandomForestClassifier
import numpy as np

# Load the spaCy model for dependency parsing
nlp = spacy.load("en_core_web_sm")

# Sample sentences with temporal relations
sentences = [
    "She finished her work before going to lunch.",
```

```
    "He went to the gym after work.",
    "They waited until the show started.",
    "The meeting was delayed during the storm."
]

# Temporal signals
temporal_signals = ["before", "after", "during", "until", "while"]

# Feature extraction - identifying events and temporal signals
def extract_features(sent):
    doc = nlp(sent)
    events = []
    temporal_relation = ""

    for token in doc:
        if token.dep_ == "ROOT":  # Event (verb) extraction
            events.append(token.lemma_)
        if token.text in temporal_signals:  # Temporal signal detection
            temporal_relation = token.text
    return events, temporal_relation

# Prepare training data - sentences, events, and labels (1 for temporal
relation present, 0 otherwise)
X = []
y = []

for sentence in sentences:
    events, signal = extract_features(sentence)
    if signal:  # If temporal signal is present, we classify as 1
        X.append([len(events)])  # Simple feature: number of events
        y.append(1)
    else:
        X.append([0])
        y.append(0)

# Convert to numpy arrays
X = np.array(X)
y = np.array(y)

# Train a Random Forest Classifier
clf = RandomForestClassifier(n_estimators=10, random_state=42)
clf.fit(X, y)

# Predict on new sentences
new_sentences = ["John started his project before the deadline.",
                 "She arrived after the party had begun."]

for new_sent in new_sentences:
    events, signal = extract_features(new_sent)
    prediction = clf.predict([[len(events)]])
    print(f"Sentence: {new_sent}")
    print(f"Detected Temporal Signal: {signal}, Prediction: {'Temporal
relation' if prediction == 1 else 'No temporal relation'}\n")
```

```
Sentence: John started his project before the deadline.
Detected Temporal Signal: before, Prediction: Temporal relation

Sentence: She arrived after the party had begun.
Detected Temporal Signal: after, Prediction: Temporal relation
```

We will throw some line on the function def extract_features(sent): for better understanding.

- extract_features(sent): Processes each sentence using spaCy. The function returns the list of events (verbs) and the temporal relation signal (if present).
- doc = nlp(sent): Prepared the sentence and gets it into a structured format for analysis.
- for token in doc:: Loops through each word in the sentence.
- token.dep_ == "ROOT": Checks if the word is the main verb (the root action) and stores it.
- if token.text in temporal_signals:: If a word is found in the list of temporal signals, it is stored that as a temporal signal.

Converting words into ML model format: Without vectorizing the entire sentence, the logic uses a simple logic for converting words into numbers. It counts the number of verbs (events) as features in X, and puts a binary label (1 or 0) for y based on the presence of temporal signals. This keeps the dataset simple and in numerical format without using a vectorizer.

The output confirms that the code is effectively detecting temporal signals present in the input sentences. The model is also correctly predicting the existence of temporal relations based on these signals.

```
>>> Code Snippet 7.1
```

7.4 Distributional Semantics and Word Embeddings

Distributional Semantics deals with discovering word meanings based on the company they keep. Words appearing in similar contexts often have interrelated meanings. Let's take an example. "Doctor" and "nurse" are many times used together in hospital and healthcare documents; we can assume they are related in meaning. Let's take one more example, the words "king" and "queen" are often found together in contexts related to royalty, palaces, or authority. This way they are seen as similar. Distributional Semantics works based on the distributional hypothesis**,** which states, "You shall know a word by the company it keeps."

To recognize word meanings, each word is assigned a unique vector, which appear as a list of numbers. These word vectors are created using a large amount of text data. If two words meanings, the vectors for both are close to each other. Similar words have less Euclidean distance or cosine similarity (we will talk about it later) between the vectors. With these numerical forms of words, we can measure similarity. These word vectors are sometimes used interchangeably in meaning with "word embeddings."

Let’s see how word vectors (word embeddings) look like? The actual numerical values of word embeddings for words like "king," "man," "woman," and "queen" can change depending on the type of word embedding model used. There are many word embedding models in use, few examples are Word2Vec, GloVe, and FastText. To give you an idea, we will use the hypothetical word embeddings of king, man, and woman as follows.

1. king: 0.6,0.4,0.1,−0.2,0.3,0.5,−0.10
2. man: 0.3,0.5,0.2,−0.1,0.1,0.4,0.00
3. woman: 0.4,0.3,0.5,0.0,0.2,0.6,0.1

If you do a simple math (king - man) + woman, you will get calculated value of queen.

- $Queen_c$: 0.7,0.2,0.4, −0.1,0.4,0.7,0.0

The calculated vector $Queen_c$ is close to the following hypothetical vector of queen ($Queen_h$).

- $Queen_h$**:** 0.7,0.5,0.3,0.1,0.4,0.8,0.2

Here we have used the classic example of queen ≈ king - man + woman, which demonstrates how word embeddings can capture relationships between words through mathematical operations. Word embeddings are useful for various NLP tasks like machine translation and sentiment analysis.

Cosine Similarity: A couple of paragraphs before, we used the terms cosine similarity and Euclidean distance. Let’s examine what they represent. We will start our discussion with Euclidean distance.

Euclidean distance is the distance between two word vectors in space. Let's take a simple example of two word vectors with hypothetical numerical values as follows. To keep it simple, we will take only three numbers in each word vector.

1. king: [0.8, 0.5, 0.6]
2. queen: [0.7, 0.6, 0.5]
3. apple: [0.2, 0.1, 0.3]

The Euclidean distance "king" and "queen " will be calculated as:

Euclidean distance (king, queen) = SQUARE ROOT$((0.8-0.7)^2+(0.5-0.6)^2+(0.6-0.5)^2)$ = 0.173

Euclidean distance (king, apple) = SQUARE ROOT$((0.8-0.2)^2+(0.5-0.1)^2+(0.6-0.3)^2)$ = 0.781

As expected, Euclidean distance (king, queen) is lesser than Euclidean distance (king, apple). The reason is simple; king and queen are similar in meaning as both belong to royal family, while king and apple have different meanings.

Similar to Euclidean distance, cosine similarity also measures how close or similar word vectors are. Euclidean distance is based on calculating the actual distance between vectors, while cosine similarity measures the angle between them. Cosine similarity focuses on direction rather than magnitude. Let’s take an example and calculate the cosine similarity between two words. Below we will use the same word vectors of “king”, “queen”, and “apple” with same hypothetical numerical values.

To calculate the cosine similarity between the words “king” and “queen,” we will take these steps.

- Dot product of “king” and “queen” word vectors = $(0.8\times0.7)+(0.5\times0.6)+(0.6\times0.5)=1.16$
- Magnitude of “kind” word vector = SQUARE ROOT$((0.8)^2+(0.5)^2+(0.6)^2)$=1.118
- Magnitude of “queen” word vector= SQUARE ROOT$((0.7)^2+(0.6)^2+(0.5)^2)$= 1.049

- Cosine similarity = 1.16/ (1.118×1.049) = 0.989

Similarly, if we compute the cosine similarity between “king” and “apple,” it will come out as 0.933. Note that cosine similarity of “king” and “queen” (0.989) is more than the cosine similarity of “king” and “apple,” (0.933). It indicates closer relationship between the former pair compared to the later. Cosine similarity ranges from -1 to 1.

- 1 indicates perfect similarity (parallel vectors that point in the same direction).
- 0 means no similarity (vectors are orthogonal).
- -1 indicates perfect dissimilarity (vectors point in opposite directions).

Cosine similarity is often used in information retrieval, where It is used to assess how closely two documents or words relate in meaning. Euclidean distance is frequently used in the NLP tasks like document clustering.

7.4.1 Latent Semantic Analysis (LSA), a Popular Approaches to Develop Word Embeddings

Word embeddings enable machines to process natural languages, word by word. They convert words into numerical representations, which machines (often computers) understand. This way machines can interpret and process language more effectively. Over the years several models have come up to develop word embeddings like, Word2Vec, GloVe, and contextualized embeddings such as BERT and GPT. These approaches help to model the delicate nuances of language, which allow NLP applications to identify semantic similarity, context, and deeper language structures. In this section, we will explore these techniques, from traditional embeddings to the latest advancements, to understand how they contribute to language understanding in NLP applications. In this section we will discuss Latent Semantic Analysis (LSA), which is a foundational technique to develop word embeddings.

LSA is a technique to find the meaning of words based on their contexts. It examines large text corpora to identify patterns showing how words relate to each other. Here, *patterns* refer to the ways words are frequently found together in similar contexts. This frequency reveals that words often convey related ideas or themes when they appear in similar contexts.

We can use either the Term Document Matrix (TDM) or the TF-IDF matrix for LSA, but they serve different purposes. TF-IDF is preferred as it helps LSA as it focuses on important words in documents. TF-IDF puts more weightage to exceptional words and it help us realize the main ideas. To remind, we had discussed TDM and TF-IDF in detail earlier in this book. Following are steps of preconforming LDA using TF-IDF.

- Data Cleaning and Formatting for Analysis
- Construct the TF-IDF matrix from the cleaned documents
- Apply Singular Value Decomposition (SVD) to the TF-IDF matrix to reduce its dimensions and to identify latent topics within the text.
- Analyse the results to figure out term-document relationships and identify prominent themes or topics.
- Visualize the reduced matrix or the results of the LDA to better understand patterns and meanings in the data.

7.4.1.1 Code Demo of LSA

Now we are ready for a code demo of LSA using a simplified text corpus. We will use TF-IDF matrix in this demo, which we had discussed in the earlier in this book.

```python
# The following is a simplified code demo of LSA

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD

# Sample common text corpus
documents = [
    "The cat sat on the mat.",
    "Dogs are great companions.",
    "The sun is bright today.",
    "Cats and dogs are popular pets.",
    "The weather is nice for a walk."
]

# Step 1: Data Cleaning and Formatting for Analysis,
# Skipping this step as the data has already been cleaned.

# Step 2: Construct the TF-IDF matrix
vectorizer = TfidfVectorizer()
tfidf_matrix = vectorizer.fit_transform(documents)

# Step 3: Apply SVD to reduce dimensions
svd = TruncatedSVD(n_components=2)
lsa_matrix = svd.fit_transform(tfidf_matrix)

# Step 4: Analyze the results
# Create a DataFrame for viewing
lsa_df = pd.DataFrame(lsa_matrix, columns=['Concept 1', 'Concept 2'])
print("Reduced Dimensions (Latent Concepts):")
print(lsa_df)
```

```
Reduced Dimensions (Latent Concepts):
      Concept 1     Concept 2
0  6.263182e-01  1.932910e-16
1 -4.140572e-16  8.096747e-01
2  7.173435e-01 -7.926662e-15
3 -1.891549e-16  8.096747e-01
4  6.985829e-01  8.989044e-15
```

- *Below is the interpretation of the above output matrix.*
- In the output matrix, rows represent individual documents.
- Columns represent latent concepts or themes.
- Values indicate the strength of each document's relationship to those concepts.
- Negative values indicate a lack of relevance of that document to that concept (column).
- High values indicate more relevance of that document (row) to that concept (column).

- The first and third documents (column 1) show high values. It shows strong relevance to Concept 1.
- The second document has a significant value (column 2). It shows strong relevance to Concept 2.
- Other documents in both columns are less relevant.
- The first and third documents may share a theme.
- The second document relates to a different topic.
- Higher scores indicate thematic similarities among documents. It helps in clustering and topic identification.
- Look which documents have higher values for each concept and then checking their content.
- You can then decode what Concept 1 and Concept 2 represent based on their thematic similarities.

>>> Code Snippet 7.2

7.4.2 Popular Word Embeddings

Word embeddings use large amounts of text to learn the relationship between the words. Popular word embeddings like Word2Vec, GloVe, and FastText capture context, similarity, and semantic nuances to map words into numerical vectors. These numerical vectors essentially capture their meaning and relationships between words. Below we will divide these word embeddings in traditional and newer ones, like BERT and GPT, based on contextual word representations.

7.4.1.2 Traditional Word Embeddings

Traditional word embeddings map words to fixed-size vectors. They were the early techniques in NLP based on word co-occurrence patterns in large text datasets. Traditional techniques like **Word2Vec** and **GloVe** analyse how frequently and in what context words are appearing together in large training datasets to capture relationships between words and create word representations in the form of word vectors.

7.4.1.2.1 Word2Vec

Word2Vec (word to vector) converts words to machine readable numeric vectors, which represent a specific word (as a vector) in multidimensional space. These vectors help machines to capture word meaning, semantic similarity, and relationship with surrounding text. Word2Vec is essentially an pretrained model. It utilises a shallow neural network model to acquire the meaning of words from a large corpus of (training) texts. To make the processing prompt, faster, and transparent, Word2Vec neural networks use only one or two hidden layers. These neural networks are trained using large databases of texts. Word2Vec algorithm essentially use either continuous bag of words (CBOW) or skip-gram approaches to generate word embeddings.

7.4.1.2.1.1 Continuous Bag of Words Model (CBOW)

CBOW is an unsupervised method of find word embeddings. It predicts the target word by utilising the words surrounding it (context words). For this purpose, it utilises a shallow neural network program (CBOW model). This program learns to predict any target word by utilising the words that appear before and after it in a given context window. By concentrating on the surrounding context words, the CBOW model is able to capture the meaning of a word (in a given context) in the form of numerical vectors (embeddings).

Commented [SK1]:

For instance, in the sentence, `"he is a great scholar,"` if we want the word embeddings of the word “great,” We will first take a context window. Assume this context window has two words, then we will consider two words before the target word “great” and also consider two words after it. So, for our example sentence, the context words will be

- Two words before the target word “great” – (“is”, “a”)
- Two words after the target word “great” – (“scholar”)

Using this context window, the CBOW model will learn to calculate the word embeddings of the target word “great.” For the code demonstration of CBOW, below is a well-documented shallow neural network that follows exactly the same steps.

```
#Running a shallow neural network program to demonstrate CBOW.

# Ignore deprecation warnings for cleaner output.
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)

# Import libraries
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential  # Sequential model
from tensorflow.keras.layers import Dense, Embedding, Flatten  # Layers
for neural network
from tensorflow.keras.preprocessing.text import Tokenizer  # Tokenizer
for text preprocessing
from tensorflow.keras.preprocessing.sequence import pad_sequences  #
For padding sequences

# Sample corpus of sentences (training curpus)
sentences = [
    "he is a great scholar",
    "a great scholar writes great papers",
    "the scholar is very great",
    "great ideas come from great minds"
]

# Tokenize the sentences
tokenizer = Tokenizer()
tokenizer.fit_on_texts(sentences)  # Fit tokenizer on sentences
total_words = len(tokenizer.word_index) + 1  # Total unique words in
corpus

# Initialize lists for CBOW input-output pairs
input_data = []  # Context words
output_data = []  # Target word

# Define the context window size
window_size = 2

# Create input-output pairs for CBOW
for sentence in sentences:
    words = sentence.split()
```

```
    # Loop through the words
    for i in range(window_size, len(words) - window_size):
        context = []
        for j in range(i - window_size, i + window_size + 1):
            if j != i:  # Skip target word
                context.append(tokenizer.word_index[words[j]])  #
Append context word index
        input_data.append(context)  # Add context to input data
        output_data.append(tokenizer.word_index[words[i]])  # Add
target word to output data

# Pad input sequences to ensure uniform length
input_data = pad_sequences(input_data, padding='post')

# One-hot encode output data for categorical labels
output_data = np.array(output_data)
output_data = np.eye(total_words)[output_data]  # Convert to one-hot

# Build CBOW model
model = Sequential()
model.add(Embedding(input_dim=total_words, output_dim=10,
input_length=window_size * 2))  # Embedding layer
model.add(Flatten())  # Flatten to 1D

# Output layer with softmax for word prediction
model.add(Dense(total_words, activation='softmax'))


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

# Train the model on input-output pairs
model.fit(input_data, output_data, epochs=20, verbose=1)

# Retrieve embedding for the word "great"

# Index of "great" in vocabulary
great_index = tokenizer.word_index['great']
great_embedding = model.layers[0].get_weights()[0][great_index]  #
Extract embedding
print("Embedding for 'great':", great_embedding)
```

```
Epoch 1/20

1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 725ms/step - accuracy: 0.1667 - loss:
2.6119
Epoch 2/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.3333 - loss: 2.6053
Epoch 3/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.5000 - loss: 2.5986
.....
Epoch 20/20
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 1.0000 - loss: 2.4814
```

OUTPUT >>> Embedding for 'great': [0.019222 -0.01932465 -0.03971442 -0.02196687 -0.02987816 0.00716395 -0.01269478 0.02227857 0.01485946 -0.07757436]

>>> Code Snippet 7.3

```
    -0.01269478  0.02227857  0.01485946 -0.07757436]
```

To highlight it, the following is the desired ten element word vector or embedding for the word "great" by using CBOW model:

[0.019222 -0.01932465 -0.03971442 -0.02196687 -0.02987816 0.00716395 -0.01269478 0.02227857 0.01485946 -0.07757436]

7.4.2.1.1.2 Skip-gram Model

Skip-gram models are also unsupervised methods. Like CBOW, they are also based on shallow neural networks with only one or two hidden layers. But skip-gram is exactly opposite of CBOW. In skip-gram models, we predict context words (surrounding words) given a target word. If we choose window size=2 in the sample sentence, “he is a great scholar” then we get the following possible pairs.

- (he, is) (is, he)
- (is, a) (a, is)
- (a, great) (great, a)
- (great, scholar) (scholar, great)

Each pair has a central word as input and a context word as output. For the first pair (he, is), the central word is "he," and the context word is "is." It’s we decide which is the target, the remaining one automatically becomes the surrounding. Similarly, in (great, scholar) if we decide “great” as central or target word, “scholar” automatically becomes the context or the output word. Its for a 2-gram model. Similarly, we can have n-gram models like 2-gram and 3-gram. In 3-gram, sequences of three consecutive words are formed.

If we choose a window size = 3, the valid 3-gram groups are:

- (he, is, a), (he, a, is)
- (is, a, great), (is, great, a)
- (a, great, scholar), (a, scholar, great)

Each group consists of a central word with context before and after it. For instance, in the group (a, great, scholar)**,** the central word is "great," and the context words are "a" and "scholar."

Below we will create and run a sample neural network based skip-gam model for demonstration. It will follow the following broad steps.

- Import Libraries
- Define Corpus
- Initialize Tokenizer
- Generate Vocabulary
- Convert to Sequences
- Create Skip-Gram Pairs
- Prepare Input/Output Data

- Build Model
- Compile Model
- Train Model
- Extract Embeddings

```
#Running a shallow neural network program to demonstrate skip-gram
model

# Import necessary libraries
# For numerical operations
import numpy as np
# For deep learning models
import tensorflow as tf
# For creating a linear model
from tensorflow.keras.models import Sequential
# Layers for model building
from tensorflow.keras.layers import Embedding, Dense, Flatten
# For tokenizing text
from tensorflow.keras.preprocessing.text import Tokenizer
# For generating Skip-gram pairs
from tensorflow.keras.preprocessing.sequence import skipgrams

# Sample corpus for demonstration
sentences = ["he is a great scholar", "a great scholar writes great
papers"]

# Initialize tokenizer and fit on text
tokenizer = Tokenizer()
# Fit tokenizer on sample sentences
tokenizer.fit_on_texts(sentences)
# Get vocabulary size
total_words = len(tokenizer.word_index) + 1

# Convert sentences to sequences of word indices
# Convert sentences to integer sequences
sequences = tokenizer.texts_to_sequences(sentences)

# Generate word pairs (center, context) for Skip-gram
# Initialize list to store Skip-gram pairs
skip_gram_pairs = []
for seq in sequences:
    # For each sentence sequence
    pairs, _ = skipgrams(seq, vocabulary_size=total_words,
window_size=2)
    # Generate Skip-gram pairs
    skip_gram_pairs.extend(pairs)  # Add pairs to list

# Separate input (center) and output (context) words
input_words, context_words = zip(*skip_gram_pairs)
# Convert input words to array
input_words = np.array(input_words)
# Convert context words to array
context_words = np.array(context_words)
```

```
# Define Skip-gram model
model = Sequential()
# Embedding layer with vector size 10
model.add(Embedding(total_words, 10, input_length=1))
# Flatten the output
model.add(Flatten())
# Output layer for vocabulary-size classification
model.add(Dense(total_words, activation='softmax'))

# Compile model with categorical cross-entropy
# Set optimizer and loss function
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy')

# Train model on (input, context) pairs
model.fit(input_words, context_words, epochs=20, verbose=1)

# Get embeddings for a specific word, e.g., "great"
# Find index of word "great"
great_index = tokenizer.word_index['great']
# Get embedding vector for "great"
great_embedding = model.layers[0].get_weights()[0][great_index]
# Print embedding
print("Embedding for 'great':", great_embedding)
```

- ```
  Epoch 1/20
  2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - loss: 2.0738
  Epoch 2/20
  2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 2.0708

  …….
  Epoch 20/20
  2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 2.0389
  OUTPUT >>> Embedding for 'great': [ 0.01801779 -0.0028935  -
  0.04373218  0.06102643 -0.04138939 -0.05270927
    0.1154007  -0.08093932 -0.07753371 -0.02089559]
  ```

```
>>> Code Snippet 7.4
```

To highlight it, the following is the desired ten element word vector or embedding for the word "great" by using skip-gram model:

```
[ 0.01801779 -0.0028935  -0.04373218  0.06102643 -0.04138939 -0.05270927
0.1154007  -0.08093932 -0.07753371 -0.02089559]
```

You may like to compare this output to that of CBOW model, calculated in the previous section.

7.4.2.1.2 CBOW and Skip-gram Code Demonstration Using Word2Vec

Below we run a Word2Vec based program to demonstrate CBOW and Skip-gram models. The below Word2Vec code implementation of these models is based on a pre-built, optimized algorithm for training word embeddings. While neural network based programs that we discussed in earlier
```

sections have manually created and trained customised shallow neural networks. Word2Vec is highly efficient and handles large corpora better when compared to custom built models.

Our program below will follow the following broad steps:

- Import Libraries and Prepare Data
- Tokenize Sentences and Prepare Corpus
- Initialize Word2Vec Model for CBOW or Skip-gram
- Train the Word2Vec Model
- Access Word Embeddings
- Evaluate and Use the Embeddings

```
#Running a Word2Vec based program to together demonstrate CBOW and
skip-gram models

# Import necessary libraries
import gensim
from gensim.models import Word2Vec
from nltk.tokenize import word_tokenize

# Sample corpus (list of sentences)
sentences = ["he is a great scholar", "a great scholar writes great
papers"]

# Tokenize the sentences into words
tokenized_sentences = [word_tokenize(sentence.lower()) for sentence
in sentences]

# CBOW Model (use skip_gram=False for CBOW)
cbow_model = Word2Vec(tokenized_sentences, vector_size=10, window=2,
min_count=1, sg=0)
# sg=0 denotes CBOW model

# Train CBOW model
cbow_model.save("cbow_model.bin")

# Predict word embedding for 'great' using CBOW model
cbow_embedding = cbow_model.wv['great']
print("CBOW Embedding for 'great':", cbow_embedding)

# Skip-gram Model (use skip_gram=True for Skip-gram)
skipgram_model = Word2Vec(tokenized_sentences, vector_size=10,
window=2, min_count=1, sg=1)
# sg=1 denotes Skip-gram model

# Train Skip-gram model
skipgram_model.save("skipgram_model.bin")

# Predict word embedding for 'great' using Skip-gram model
skipgram_embedding = skipgram_model.wv['great']
print("Skip-gram Embedding for 'great':", skipgram_embedding)
```

```
OUTPUT 1>>> CBOW Embedding for 'great': [-0.00536227  0.00236431
0.0510335   0.09009273 -0.0930295  -0.07116809 0.06458873  0.08972988 -
```

```
   0.05015428 -0.03763372]
   OUTPUT 2>>> Skip-gram Embedding for 'great': [-0.00536227  0.00236431
   0.0510335   0.09009273 -0.0930295  -0.07116809 0.06458873  0.08972988 -
   0.05015428 -0.03763372]
```

```
>>> Code Snippet 7.5
```

Above you can compare and notice the embeddings for 'great' based on CBOW and skip-gram models using Gensim library, which provides an efficient code implementation of Word2Vec for training and using word embeddings.

7.4.2.1.3 GloVe: Global Vectors for Word Representation

GloVe is an unsupervised machine learning algorithm that was first presented in 2014 in a conherence, organised at Doha, Qatar. GloVe is used to generate vector representations (word embeddings) of words. The glove contains pre-defined dense vectors for approximately over 6 billion words of English literature including some general-use characters like commas, braces, and semicolons. Users can make use a pre-trained GloVe embedding in different dimensions like, 50-d, 100-d, 200-d, or 300-d vectors as per computational resources availability and the task needs. Here d represents dimension. 50-d would indicate a vector of size 50 and so on.

The main difference between Word2Vec and GloVe is that Word2Vec learns word embeddings (or word vectors) by predicting context words using Skip-gram or CBOW models (utilising large corpora). Its focus is on local context. While GloVe utilises global word co-occurrence statistics to seize overall word relationships in its embeddings. GloVe model emphases on global as well as local context. Word2Vec is generally faster and more efficient for context-specific NLP tasks, because it dynamically captures word relationships during training. While GloVe works great for capturing overall word relationships across an entire text dataset, because it learns from global word co-occurrences (meaning how often words appear together across many sentences). This way GloVe captures a broader understanding of word meanings and their relationships.

Like Word2Vec, the word embeddings provided by GloVe can be used in multiple NLP tasks like Named Entity Recognition (NER), Machine Translation, Question Answering Systems, Document Similarity and Document Clustering.

Below is a code demo that utilises pre-trained GloVe embeddings to find similar words.

```
# Using pre-trained GloVe embeddings to find similar words.

# Necessary steps before you run the the code
```

Go to the GloVe project page.

Download the file glove.6B.zip. It contains word embeddings of multiple dimensions like 50d, 100d, etc.

Unzip the file to see glove.6B.50d.txt.

Place glove.6B.50d.txt in the same directory as your code file, or specify the correct path in the code if it's located elsewhere.

```
# Import necessary libraries
import numpy as np  # For numerical operations
from sklearn.metrics.pairwise import cosine_similarity  # For similarity
calculation
```

```python
# Load GloVe embeddings
# Path to GloVe file (ensure this file is in the directory or provide the
correct path)
embeddings_index = {}  # Dictionary to store word vectors

# Open and read the GloVe file
with open('glove.6B.50d.txt', encoding='utf-8') as f:
    for line in f:
        values = line.split()  # Split each line into components
        word = values[0]  # First item is the word
        vector = np.asarray(values[1:], dtype='float32')  # Remaining
items are vector values
        embeddings_index[word] = vector  # Add word and its vector to
dictionary

# Check if "scholar" exists in the GloVe vocabulary
if "scholar" in embeddings_index:
    scholar_vector = embeddings_index["scholar"]  # Get the vector for
"scholar"

# Calculate cosine similarity between "scholar" and all other words
similar_words = {}  # Dictionary to store words and their similarity
scores
for word, vector in embeddings_index.items():
    similarity = cosine_similarity([scholar_vector], [vector])[0][0]  #
Compute similarity
    similar_words[word] = similarity  # Add word and similarity score to
dictionary

# Sort words by similarity score in descending order
sorted_similar_words = sorted(similar_words.items(), key=lambda x: x[1],
reverse=True)

# Print top 10 words similar to "scholar"
print("Top words similar to 'scholar':")
# Skip the first word as it will be "scholar" itself
for word, similarity in sorted_similar_words[1:11]:
    print(f"{word}: {similarity}")
```

```
OUTPUT >>> Top words similar to 'scholar':
historian: 0.8646411895751953
philosopher: 0.8056567311286926
poet: 0.7976035475730896
author: 0.7962594032287598
professor: 0.7924675345420837
eminent: 0.7883749008178711
literature: 0.7770242094993591
theologian: 0.7625619173049927
sociologist: 0.7605342864990234
linguist: 0.7595699429512024
```

Note that similarity stands for how close in meaning two words are based on the cosine similarity between their vectors. Values close to 1 means words are closer in meanings.

```
>>> Code Snippet 7.6
```

7.4.2.1.4 Advances like BERT and GPT

There are significant advances in NLP in the recent years. BERT stands for Bidirectional Encoder Representations from Transformers. It was first introduced in 2018. The most popular model in GPT category is ChatGPT. It was first released in 2022. Both BERT and GPT come under the category of large language models (LLM) and are based on transformer models. LLMs are trained on massive amount of data and are able to perform a variety of tasks like text translation and text generation with little fine tuning. In this chapter, we will concentrate only on BERT as there is a separate chapter towards the end of this book which is solely devoted to GPT based models.

BERT is a kind of neural network based on transformer architecture. It's a pretrained model, which can be fine-tuned to specific NLP tasks just by adding new layer or a small network on top of BERT's architecture. BERT model is available in a variety of sizes to suit the availability of computational resources. The following are the BERT models used in practice.

- **BERT-Base:** It is a smaller BERT model. It consists of 12 layers and 110 million parameters. It strikes a balance between performance and speed for general NLP tasks.
- **BERT-Large:** A larger version of BERT. It consists of 24 layers and 340 million parameters. This BERT model offers higher accuracy mut it required more computational resources.
- **DistilBERT:** It is smaller and faster variant of BERT with around 66% of BERT's size. This BERT model maintains high accuracy while enabling faster processing time.
- **TinyBERT:** It is a compact BERT model. It is designed for fast performance and low memory usage. It has fewer network layers and hidden units.
- **ALBERT:** It is a lightweight version of BERT. It shares parameters across layers, there by reducing model size while maintaining performance.

The **BERT-Base model is a popular general purpose BERT model, used for a wide variety of NLP tasks.** Its wide uses are justified as it maintains a reasonable performance, speed, and consume less computational resources.

The Hugging Face framework simplifies working with pre-trained models like BERT, GPT, and others. Below we will give a code demo on how Hugging Face framework can be utilised for NLP tasks. For this purpose, we will utilise "bert-base-uncased" model. It is a version of BERT-Base, which does not differentiate between uppercase and lowercase letters. In this book, we will concentrate more on the usage of BERT and GPT rather than their internal architecture.

```
"""

Code demo for sentiment analysis using the sentence "Ramesh is my friend
and he is a great scholar" with Hugging Face and BERT.

"""
```

```
import torch
from transformers import BertTokenizer, BertForSequenceClassification,
pipeline

# Set random seed for reproducibility
torch.manual_seed(42)

# Load pre-trained BERT model and tokenizer
model = BertForSequenceClassification.from_pretrained('bert-base-uncased',
num_labels=2)
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# Create a sentiment-analysis pipeline
nlp = pipeline('sentiment-analysis', model=model, tokenizer=tokenizer)

# Test sentence
sentence = "Ramesh is my friend and he is a great scholar."

# Get sentiment prediction
result = nlp(sentence)

# Print result
print(result)
```

```
OUTPUT: [{'label': 'LABEL_1', 'score': 0.5709643363952637}]
```

```
The model is predicting LABEL_1, meaning a positive sentiment, using
57.09% confidence.
```

```
>>> Code Snippet 7.7
```

7.4.3 Case Studies and Applications

Below, we will take a business case now and demonstrate its code implementation.

The owner of a large customer service platform (large e-commerce company) wants to analyse incoming customer queries with an aim to improve response times and accuracy. He is a technology enthusiast and he wants to test BERT and Hugging Face in this customer service application. With the help of this application, he wants to categories the customer queries in to returns, product inquiries, and complaints. It will allow him to route specific customer queries directly to the specialised teams. He wants to use BERT to pick up the context and content of each query and provide targeted responses. He hopes, with such an implementation, his establishment will be able to streamlines customer service, cuts operational costs, and enables more efficient handling of high query volumes.

Our code demo does the following tasks.

- Loads pre-trained BERT model and tokenizer
- Tokenizes sample data queries
- Sets up minimal training to fine-tune BERT on sample data
- Uses fine-tuned model in Hugging Face pipeline for predictions
- Maps prediction results to defined categories

```
# Code demo for the business case.

# Install these libraries if not dopne already.
!pip install transformers torch

from transformers import BertTokenizer, BertForSequenceClassification,
Trainer, TrainingArguments, pipeline
import torch
from torch.utils.data import DataLoader, Dataset
import numpy as np

# Define sample data
data = [
    {"text": "I want to return my order", "label": 0},  # returns
    {"text": "Can I know the delivery status?", "label": 1},  # product
inquiries
    {"text": "My product is defective", "label": 2}  # complaints
]

# Map label indices to categories
label_map = {0: "returns", 1: "product inquiries", 2: "complaints"}

# Load pre-trained BERT tokenizer and model
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertForSequenceClassification.from_pretrained('bert-base-uncased',
num_labels=3)

# Tokenize and preprocess data
class SampleDataset(Dataset):
    def __init__(self, data, tokenizer):
        self.data = data
        self.tokenizer = tokenizer

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        item = self.data[idx]
        inputs = self.tokenizer(item["text"], truncation=True,
padding='max_length', max_length=32, return_tensors="pt")
        inputs = {k: v.squeeze() for k, v in inputs.items()}  # Squeeze to
remove extra dimension
        inputs['labels'] = torch.tensor(item["label"], dtype=torch.long)
        return inputs

# Create Dataset and DataLoader
dataset = SampleDataset(data, tokenizer)

# Set up minimal training for fine-tuning
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="no",
    per_device_train_batch_size=2,
```

```
    num_train_epochs=20,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=dataset,
)

# Train the model (this is minimal; actual fine-tuning requires more data
and epochs)
trainer.train()

# Use Hugging Face pipeline for prediction
nlp_pipeline = pipeline("text-classification", model=model,
tokenizer=tokenizer)

# Predict categories for new queries
new_queries = ["I received the wrong item", "Where is my package?", "Can I
return this?", "I received a defective product"]
for query in new_queries:
    result = nlp_pipeline(query)[0]
    predicted_label = int(result['label'].split('_')[-1])  # Get label
index from 'LABEL_0', 'LABEL_1', etc.
    category = label_map[predicted_label]  # Map label to category
    print(f"Query: '{query}'\nPredicted Category: {category}\n")
```

```
OUTPUT >>>

Query: 'I received the wrong item'
Predicted Category: complaints

Query: 'Where is my package?'
Predicted Category: product inquiries

Query: 'Can I return this?'
Predicted Category: product inquiries

Query: 'I received a defective product'
Predicted Category: complaints
```

>>> Code Snippet 7.8

In above code Hugging Face is used to:

- Load the pre-trained BERT model.
- Provide a tokenizer for text pre-processing.
- Set up text classification (Pipeline function).
- Encapsulate the prediction process in a single call.
- Map results to the defined categories.

7.4.3 Summary

In this chapter, we have explored the essence of pragmatic analysis. We mainly focused on discourse integration, distributional semantics, and word embeddings. We discussed these broad topics ion depth from conceptual, implementation, and practical applications point of view. These insights will help you to apply these practice and your future NLP endeavours.

The current NLP focus area are understanding multimodality, platform-specific communication styles, identity and community building, and politeness dynamics in digital interactions. Multimodality concentrates on how text, images, videos, and emojis work together to communicate meaning online. Popular web platforms like X and Instagram have developed their own style of communicating, used language, and shared ideas. Some leading researchers are focusing on how each platform influences users' communication styles. The current NLP research areas also focus on politeness and ethical; aspects of natural languages. All this research is keeping its main focus on online and social media communications between groups of people. The current trends indicate that the future NLP researchers will likely focus more on areas like AI in Communication, Cross-Cultural Interaction, Ethics of Interaction, Tech Advancements, and Digital Literacy

References

[1] Sharma, P., & Nagashree, N. (2022). Survey on Natural Language Processing and its Applications. *International Journal of Computational Learning & Intelligence*, *1*(2), 12-14.