Natural Language Processing for Machine Learning

Foundations of NLP: Core Concepts, Techniques, and Applications

by

Pranav Jha

August 26, 2025

Contents

1 Introduction to Natural Language Processing

Natural Language Processing (NLP) is one of the most exciting domains within machine learning and deep learning. It enables machines to process, understand, and generate human language. This chapter introduces the roadmap for preparing and working in NLP, outlining the fundamental steps required to move from traditional machine learning techniques to advanced deep learning methods such as transformers.

1.1 Foundations of Machine Learning for NLP

Before diving into NLP, it is essential to recall the fundamentals of machine learning. In supervised machine learning, problems are typically divided into:

- **Classification** – predicting discrete categories (e.g., spam vs. not spam).
- **Regression** – predicting continuous values.

In both cases, we deal with:

- **Independent features** (input features, e.g., F1, F2, F3, …).
- **Dependent feature** (output variable, e.g., spam or not spam).

Models are trained on datasets that contain these features, enabling predictions for new unseen inputs.

However, when features are in the form of text (such as email subjects or bodies), additional processing is required because models cannot directly understand human language. For example, in a spam classification task:

- Input features may include the **email subject** and **email body**.
- The output label indicates whether the email is **spam** or **ham** (not spam).

Since text cannot be directly fed into machine learning models, NLP techniques are employed to convert text into meaningful numerical representations (vectors).

1.2 Roadmap for NLP

The roadmap for NLP can be seen as a bottom-to-top approach, where each step increases in complexity and effectiveness. The following stages outline this progression.

1.2.1 Step 1: Text Pre-processing (Part I)

The first step involves cleaning and preparing raw text. Common preprocessing tasks include:

- **Tokenization** – breaking a paragraph into sentences and words.
- **Stopword removal** – removing commonly used words (e.g., "the", "is") that add little meaning.
- **Stemming** – reducing words to their root form (e.g., "playing" → "play").
- **Lemmatization** – converting words to their base dictionary form with context (e.g., "better" → "good").

These steps ensure that the text is standardized and ready for numerical transformation.

1.2.2 Step 2: Text Representation (Part II)

The next step focuses on representing text numerically through vectorization techniques. Methods include:

- **Bag of Words (BoW)**
- **TF-IDF (Term Frequency–Inverse Document Frequency)**
- **N-grams** (unigrams, bigrams, trigrams)

These techniques convert text into sparse numerical vectors, allowing models to capture word frequency and co-occurrence.

1.2.3 Step 3: Advanced Text Representation (Part III)

To capture deeper semantic meaning, more advanced techniques are applied:

- **Word2Vec** – converts words into dense vector embeddings that preserve semantic relationships.
- **Average Word2Vec** – averages word embeddings to represent sentences or documents.

Unlike Bag of Words or TF-IDF, these methods provide richer, context-aware representations.

1.2.4 Step 4: Deep Learning Architectures

NLP tasks often benefit from deep learning architectures that capture sequential dependencies:

- **RNN (Recurrent Neural Networks)**
- **LSTM (Long Short-Term Memory)**
- **GRU (Gated Recurrent Unit)**

These models are widely applied to text-related tasks such as spam classification, text summarization, and machine translation.

1.2.5 Step 5: Word Embeddings and Contextual Representations

In addition to Word2Vec, more sophisticated embedding techniques are used:

- **Custom Word Embeddings** – embeddings trained on domain-specific data.
- **Pre-trained Embeddings** – embeddings trained on large corpora (e.g., GloVe).

These embeddings provide context-rich vector representations of text.

1.2.6 Step 6: Transformers and BERT

The most advanced stage involves modern transformer architectures:

- **Transformers** – self-attention mechanisms for capturing global dependencies in text.
- **BERT (Bidirectional Encoder Representations from Transformers)** – pre-trained contextual embeddings that achieve state-of-the-art results across NLP tasks.

These models significantly improve accuracy but also require greater computational resources.

1.3 Libraries and Tools

At different stages of the roadmap, different libraries are commonly used:

- **Machine Learning (Steps 1–3)**: NLTK, spaCy
- **Deep Learning (Steps 4–6)**: TensorFlow, PyTorch

For example, NLTK provides tools for tokenization, stemming, and stopword removal, while spaCy offers advanced preprocessing capabilities. TensorFlow and PyTorch are widely used for training deep learning models.

1.4 Summary

The NLP roadmap progresses from basic preprocessing to advanced deep learning techniques:

1. Clean and preprocess text.
2. Convert text into numerical vectors using classical methods (BoW, TF-IDF).
3. Use advanced embeddings such as Word2Vec.
4. Apply deep learning models such as RNN, LSTM, GRU.
5. Utilize word embeddings and contextual representations.
6. Implement transformer-based models like BERT for state-of-the-art performance.

This progression ensures that learners can gradually build their understanding, from simple machine learning pipelines to cutting-edge deep learning models for NLP.

2 Applications of Natural Language Processing

Natural Language Processing (NLP) has become an integral part of our daily lives. Its applications span across communication, translation, information retrieval, and virtual assistance. This chapter presents some of the most common and impactful use cases of NLP.

2.1 Spell Checking and Auto-Completion

Modern email platforms such as Gmail use NLP to automatically correct spelling mistakes and provide auto-completion suggestions. For example:

- If a user types an incorrect word, the system automatically corrects the spelling.
- While composing an email, predictive text suggestions help complete sentences or phrases efficiently.

This reduces errors and improves productivity.

2.2 Smart Replies

Social platforms like LinkedIn provide *automated reply suggestions*. When receiving a message, the system generates short and contextually relevant responses such as "Thank you" or "Sounds good." This feature leverages NLP models to save time by allowing users to respond with a single click.

2.3 Machine Translation

Machine translation tools, such as Google Translate, are widely used to convert text from one language to another. For instance:

- "How are you?" in English can be translated to Hindi as "Kya haal hai?"
- Similar conversions are possible across dozens of languages.

Additionally, platforms like LinkedIn often provide inline translation options for posts written in foreign languages, enhancing accessibility for global users.

2.4 Information Retrieval: Text-to-Image and Text-to-Video

Search engines such as Google integrate NLP for text-based queries:

- A text query like "Johnna" can retrieve not only web pages but also associated images and videos.
- The system uses semantic understanding to map text queries to relevant multimedia content.

This represents an application of NLP in **text-to-image** and **text-to-video** retrieval.

2.5 NLP in Industry: Hugging Face

Hugging Face is a leading organization providing open-source NLP models and frameworks. Their platform hosts thousands of pre-trained models for tasks such as:

- Question Answering
- Summarization
- Text Classification
- Machine Translation

For example:

- Over 2300 models are available for Question Answering.
- More than 500 models exist for Summarization.
- Hundreds of models support tasks like Sentiment Analysis and Translation.

Several leading companies—including Google AI, Microsoft, Grammarly, and Intel—integrate Hugging Face solutions into their workflows.

2.6 Virtual Assistants

Voice-activated assistants such as **Alexa** and **Google Assistant** make extensive use of NLP. They are capable of:

- Recognizing user queries through speech recognition.
- Retrieving relevant information (e.g., checking calendar appointments).
- Controlling smart devices such as lights and air conditioners.

These assistants showcase how NLP is integrated with everyday technologies to provide seamless human-computer interaction.

2.7 Summary

NLP applications are embedded into numerous daily activities:

1. Spell correction and auto-completion in emails.
2. Automated replies in messaging platforms.
3. Language translation across platforms.
4. Information retrieval in search engines.
5. Industry-scale pre-trained models provided by Hugging Face.
6. Virtual assistants for task automation and smart device integration.

Through these applications, it is clear that NLP is not just a theoretical concept but a powerful technology that enhances communication and productivity in everyday life. In the following chapters, we will explore the techniques that enable these applications, starting from text preprocessing to advanced deep learning architectures.

3 Basic Terminologies and Tokenization in NLP

In this chapter, we introduce some fundamental terminologies used in Natural Language Processing (NLP). These concepts will be repeatedly encountered throughout our study, so it is essential to understand them clearly. The key topics covered in this chapter include:

- Corpus
- Document
- Vocabulary
- Words
- Tokenization

3.1 Basic Terminologies

3.1.1 Corpus

A **corpus** refers to a large collection of text. It may consist of paragraphs, sentences, or even entire documents. For example, consider the following paragraph:

> *My name is John. I have an interest in teaching Machine Learning, NLP, and Deep Learning.*

This paragraph as a whole is considered a corpus.

3.1.2 Documents

A **document** refers to an individual sentence or text unit within a corpus. For example, from the above corpus, we can extract the following documents:

- Document 1: *My name is John.*
- Document 2: *I have an interest in teaching Machine Learning, NLP, and Deep Learning.*

3.1.3 Vocabulary

The **vocabulary** refers to the set of unique words in a corpus. For example, in the two sentences above, the unique words include *my, name, is, John, have, interest, teaching, machine, learning, NLP, deep.*

3.1.4 Words

Words are the smallest text units (tokens) present in a corpus. These may repeat across different documents.

3.2 Tokenization

3.2.1 Definition

Tokenization is the process of splitting a corpus into smaller units, known as tokens. Depending on the context, these tokens may represent:

- Sentences (Sentence Tokenization)
- Words (Word Tokenization)

3.2.2 Example: Sentence Tokenization

Consider the following paragraph:

> *My name is John. I am also a YouTuber.*

After applying sentence tokenization, the corpus is divided into:

- Sentence 1: *My name is John.*
- Sentence 2: *I am also a YouTuber.*

3.2.3 Example: Word Tokenization

Now consider applying word tokenization on the sentence:

> *My name is John.*

The result is:

- Tokens: *My*, *name*, *is*, *John*

3.2.4 Illustrative Example

Consider the paragraph:

> *I like to drink apple juice. My friend likes mango juice.*

- Total Words: 11
- Unique Words (Vocabulary): 9

$$\text{Vocabulary} = \{\text{I, like, to, drink, apple, juice, my, friend, mango}\}$$

Thus, vocabulary size refers to the number of unique words present in the corpus.

3.3 Python Implementation of Tokenization

In practice, tokenization can be easily performed using libraries such as `NLTK`. For example:

```python
from nltk.tokenize import sent_tokenize, word_tokenize

text = "My name is John. I am also a YouTuber."

# Sentence Tokenization
sentences = sent_tokenize(text)
print("Sentences:", sentences)

# Word Tokenization
words = word_tokenize(text)
print("Words:", words)
```

3.4 Conclusion

In this chapter, we introduced the key concepts of corpus, documents, vocabulary, and words. We also explored **tokenization**, the essential preprocessing step in NLP, which converts text into smaller units (sentences or words). These foundations are critical, as each token will later be transformed into numerical representations for machine learning models.

4 Tokenization in Natural Language Processing

In the previous chapter, we introduced the fundamental concepts of Natural Language Processing (NLP), including terminology such as *corpus*, *paragraph*, *vocabulary*, and *words*. In this chapter, we will move towards practical implementation and demonstrate how tokenization can be performed using popular NLP libraries in Python.

4.1 Libraries for Tokenization

Two of the most widely used libraries for NLP are:

- **NLTK (Natural Language Toolkit):** A comprehensive library for working with human language data in Python, supporting tasks such as tokenization, stemming, lemmatization, and part-of-speech tagging.
- **spaCy:** A fast and efficient open-source NLP library designed for production-level applications, also supporting tokenization and advanced linguistic features.

As an exercise, readers are encouraged to compare the features and differences between NLTK and spaCy in terms of usability, speed, and functionality.

4.2 Installing NLTK

NLTK can be installed directly using pip:

```
pip install nltk
```

Once installed, it can be imported and used within a Python environment such as Jupyter Notebook.

4.3 Defining a Corpus

We begin by creating a small sample corpus (paragraph) for tokenization:

```
corpus = """Hello, welcome to NLP tutorials.
John Nayak NLP tutorials.
Please do watch the entire course to become an expert in NLP!"""
print(corpus)
```

Here, the variable `corpus` represents a paragraph consisting of multiple sentences.

4.4 Sentence Tokenization

Sentence tokenization involves splitting a paragraph into individual sentences. This can be achieved using NLTK's `sent_tokenize` function:

```
from nltk.tokenize import sent_tokenize

sentences = sent_tokenize(corpus)
print(sentences)
```

The function separates text based on punctuation marks such as full stops and exclamation marks, returning a list of sentences.

4.5 Word Tokenization

Word tokenization involves splitting a sentence or paragraph into words. This can be done using the `word_tokenize` function:

```
from nltk.tokenize import word_tokenize

words = word_tokenize(corpus)
print(words)
```

This separates the text into individual words and punctuation symbols. For example, commas and exclamation marks are treated as independent tokens.

4.6 Alternative Tokenizers

Apart from the default word and sentence tokenizers, NLTK provides additional tokenizers that handle punctuation differently.

4.6.1 WordPunctTokenizer

The WordPunctTokenizer separates punctuation more aggressively:

```
from nltk.tokenize import WordPunctTokenizer

tokenizer = WordPunctTokenizer()
tokens = tokenizer.tokenize(corpus)
print(tokens)
```

Here, contractions such as *"it's"* are split into `["it", "'", "s"]`.

4.6.2 TreebankWordTokenizer

The Treebank tokenizer follows conventions used in the Penn Treebank corpus:

```
from nltk.tokenize import TreebankWordTokenizer

tokenizer = TreebankWordTokenizer()
tokens = tokenizer.tokenize(corpus)
print(tokens)
```

This tokenizer handles punctuation differently, retaining certain marks (e.g., periods) as part of the preceding word rather than as a separate token.

4.7 Discussion

Each tokenizer has its advantages depending on the application. For general-purpose NLP tasks, `sent_tokenize` and `word_tokenize` are commonly used. For specialized preprocessing, alternative tokenizers such as WordPunctTokenizer or TreebankWordTokenizer can be employed.

4.8 Conclusion

In this chapter, we explored the concept of tokenization, the process of dividing text into smaller units such as sentences and words. Using NLTK, we demonstrated multiple approaches to tokenization and compared their behavior. In the next chapter, we will build upon this knowledge and explore text preprocessing techniques essential for NLP pipelines.

5 Text Preprocessing: Stemming in NLP

5.1 Introduction

In the previous chapters, we discussed tokenization, where a paragraph can be split into sentences, and sentences into words. Tokenization is a fundamental step in text preprocessing for Natural Language Processing (NLP).

In this chapter, we will focus on **stemming**, an essential process in text preprocessing.

5.2 What is Stemming?

According to Wikipedia, *stemming is the process of reducing a word to its word stem, base, or root form—generally a written word form known as a lemma.* Stemming helps to normalize words so that variations of a word are treated as a single term.

For example, words like `eating`, `eats`, and `eaten` all have the same stem: `eat`. Similarly, `going`, `gone`, and `goes` share the stem `go`.

5.2.1 Importance of Stemming

Stemming is particularly useful in problems such as sentiment analysis or text classification. It reduces the number of unique words (features) and helps in creating efficient models.

5.3 Stemming in Python using NLTK

NLTK provides multiple stemming algorithms. We will explore three common ones:

- Porter Stemmer
- Regex Stemmer
- Snowball Stemmer

5.3.1 Porter Stemmer

Porter Stemmer is one of the most widely used stemming algorithms in NLP.

```
from nltk.stem import PorterStemmer

words = ["eating", "eats", "eaten", "writing", "writes", "programming", "
    programs", "history", "finally", "finalized"]
stemmer = PorterStemmer()
```

```
5
6 for word in words:
7     print(f"Word: {word} -> Stemmed: {stemmer.stem(word)}")
```

Listing 5.1: Porter Stemmer Example

Observation: Words like `eating`, `eats` are correctly stemmed to `eat`. However, some words such as `history` remain unchanged, and words like `finally` get stemmed to `final`. This is a limitation of Porter Stemmer.

5.3.2 Regex Stemmer

Regex Stemmer allows stemming using custom regular expressions.

```
1 from nltk.stem import RegexpStemmer
2
3 # Initialize regex stemmer to remove suffixes
4 reg_stemmer = RegexpStemmer('ing$|s$|e$|able$')
5
6 words = ["eating", "plays", "createable"]
7 for word in words:
8     print(f"Word: {word} -> Stemmed: {reg_stemmer.stem(word)}")
```

Listing 5.2: Regex Stemmer Example

Observation: The stemmer removes specified suffixes based on the regular expression.

5.3.3 Snowball Stemmer

Snowball Stemmer is an improvement over Porter Stemmer. It produces more accurate stems for many words.

```
1 from nltk.stem import SnowballStemmer
2
3 words = ["eating", "eats", "eaten", "writing", "writes", "programming", "
      programs", "history", "fairly", "sportingly"]
4 snowball_stemmer = SnowballStemmer("english")
5
6 for word in words:
7     print(f"Word: {word} -> Stemmed: {snowball_stemmer.stem(word)}")
```

Listing 5.3: Snowball Stemmer Example

Observation: Snowball Stemmer gives better results for words like `fairly` → `fair` and `sportingly` → `sport`. Some words, such as `history` or `goes`, may still not be stemmed perfectly.

5.4 Limitations of Stemming

- Stemming may produce stems that are not real words.
- Some words may lose their original meaning after stemming.

- In applications like chatbots, lemmatization is preferred as it uses a dictionary to return correct root forms.

5.5 Summary

In this chapter, we covered:

- The definition and importance of stemming.
- Stemming using Porter Stemmer, Regex Stemmer, and Snowball Stemmer.
- Advantages and limitations of stemming.

Stemming is crucial in text preprocessing to reduce vocabulary size and improve model efficiency. In the next chapter, we will explore **lemmatization**, which overcomes some of the limitations of stemming.

6 Text Preprocessing: Lemmatization in NLP

6.1 Introduction

In the previous chapter, we discussed **stemming**, the process of reducing a word to its word stem. While stemming is useful, it has limitations: sometimes the stemmed word loses its original meaning.

To overcome these limitations, we use **lemmatization**, which returns the **root word** (lemma) that is meaningful.

6.2 What is Lemmatization?

Lemmatization is similar to stemming, but instead of producing a stem, it produces a proper, meaningful root word. For example:

- `eating, eats, eaten` → `eat`
- `goes` → `go`
- `history` → `history`

The main advantage of lemmatization is that it preserves the meaning of the word.

6.2.1 WordNet Lemmatizer

NLTK provides a `WordNetLemmatizer` class, which uses the WordNet corpus to find correct lemmas. This makes lemmatization more accurate than stemming.

6.3 Implementation in Python

6.3.1 Importing and Initializing

```
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()
```

Listing 6.1: Importing WordNet Lemmatizer

6.3.2 Basic Lemmatization

```
word = "going"
# Default is noun ('n')
print(lemmatizer.lemmatize(word)) # Output: 'going'

# With part-of-speech tag as verb ('v')
print(lemmatizer.lemmatize(word, pos='v')) # Output: 'go'
```

Listing 6.2: Basic Lemmatization Example

6.3.3 Part-of-Speech Tags

- `n` : noun
- `v` : verb
- `a` : adjective
- `r` : adverb

Selecting the correct POS tag ensures accurate lemmatization.

6.3.4 Lemmatizing Multiple Words

```
words = ["eating", "eats", "eaten", "writing", "writes", "programming", "
    programs", "history", "finally"]

# Lemmatize as nouns
for word in words:
    print(f"Word: {word} -> Lemma (noun): {lemmatizer.lemmatize(word)}")

# Lemmatize as verbs
for word in words:
    print(f"Word: {word} -> Lemma (verb): {lemmatizer.lemmatize(word, pos='v
        ')}")
```

Listing 6.3: Lemmatizing a List of Words

Observation:

- Words like `eating`, `eats`, `writing`, `writes` are correctly lemmatized to their root forms (`eat`, `write`) when using the verb POS tag.
- Words like `history` or `finally` remain unchanged, which preserves their correct meaning.

6.4 Advantages of Lemmatization over Stemming

- Produces meaningful words rather than arbitrary stems.
- Preserves grammatical correctness.

- Useful for applications requiring precise language understanding, such as:
    - Q&A chatbots
    - Text summarization
    - Information retrieval systems

6.5 Summary

- Lemmatization returns the root **lemma** of a word rather than a stem.
- The WordNet Lemmatizer uses a dictionary-based approach for accurate results.
- Selecting the correct part-of-speech tag is essential for proper lemmatization.
- Unlike stemming, lemmatization maintains the meaning of words, making it suitable for advanced NLP tasks.

7 Stopwords in Natural Language Processing

In this chapter, we will continue our discussion on text pre-processing techniques in Natural Language Processing (NLP). So far, we have covered tokenization, stemming, and lemmatization. Now, we move on to another important concept: **Stopwords**.

7.1 Introduction to Stopwords

Stopwords are commonly used words in a language that often carry little semantic meaning in the context of many NLP tasks. Examples in English include words such as "the", "is", "in", "on", "and", "to". These words are generally removed during text preprocessing because they do not contribute significantly to the meaning of the text when performing tasks such as classification or sentiment analysis.

However, one must be cautious — in certain contexts, stopwords like "not" can carry significant importance (e.g., in sentiment analysis).

7.2 Importance of Stopwords Removal

Consider a motivational speech by Dr. A.P.J. Abdul Kalam. While analyzing such text, words like "the", "is", "have" appear frequently but do not add much to the semantic meaning. Removing them reduces noise and allows models to focus on the more meaningful words. This results in:

- Reduced dimensionality of the dataset.
- Faster training and prediction times.
- Improved model performance for tasks where such words are not relevant.

7.3 Working with Stopwords in NLTK

We will now use the Python NLTK library to explore stopwords and how to remove them.

7.3.1 Importing Required Libraries

```
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk.tokenize import sent_tokenize, word_tokenize

```

```
# Download stopwords if not already downloaded
nltk.download('stopwords')
nltk.download('punkt')
```

7.3.2 Viewing Stopwords

NLTK provides predefined stopwords for many languages. Let us check the English stopwords.

```
stop_words = set(stopwords.words('english'))
print(len(stop_words))
print(list(stop_words)[:20]) # print first 20 stopwords
```

7.3.3 Applying Stopwords Removal with Stemming

We will apply both stopword removal and stemming on a sample paragraph.

```
paragraph = """
I have three visions for India. In 3000 years of our history, people from
    all
over the world have come and invaded us. From Alexander onwards, the Greeks
    ,
the Turks, the Moguls, the Portuguese, the British, the French, the Dutch,
all of them came and looted us, captured our minds. """

# Initialize stemmer
stemmer = PorterStemmer()

# Sentence tokenization
sentences = sent_tokenize(paragraph)

# Apply stopword removal + stemming
processed_sentences = []
for sent in sentences:
    words = word_tokenize(sent)
    filtered_words = [stemmer.stem(w.lower()) for w in words if w.lower()
        not in stop_words]
    processed_sentences.append(' '.join(filtered_words))

print(processed_sentences)
```

7.3.4 Output

After applying stopword removal and stemming, the sentences are shortened and reduced to their root forms. For example:

```
['vision india .', '3000 year histori , peopl world come invad us .', ...]
```

7.4 Stopwords in Different Languages

NLTK also provides stopwords for other languages such as German, French, and Arabic. For example:

```
print(stopwords.words('german')[:20])
print(stopwords.words('french')[:20])
print(stopwords.words('arabic')[:20])
```

This is useful when working with multilingual datasets.

7.5 Comparison: Stemming vs Lemmatization with Stopwords

Stemming sometimes produces truncated or unnatural words (e.g., "history" becomes "histori"). Lemmatization, on the other hand, provides linguistically correct base forms.

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

lemmatizer = WordNetLemmatizer()

processed_lemmas = []
for sent in sentences:
    words = word_tokenize(sent)
    filtered_words = [lemmatizer.lemmatize(w.lower()) for w in words if w.
        lower() not in stop_words]
    processed_lemmas.append(' '.join(filtered_words))

print(processed_lemmas)
```

7.6 Conclusion

In this chapter, we introduced stopwords and demonstrated their removal using NLTK. We also combined stopword removal with stemming and lemmatization. Stopword removal is a key pre-processing step in NLP pipelines, but it should be applied thoughtfully depending on the task.

In the next chapter, we will continue with Part-of-Speech (POS) tagging and Named Entity Recognition (NER).

8 Part-of-Speech Tagging in Natural Language Processing

In the previous chapter, we discussed stopwords and their role in text preprocessing. In this chapter, we will explore **Part-of-Speech (POS) Tagging**, a fundamental step in Natural Language Processing (NLP). POS tagging assigns a grammatical category (such as noun, verb, adjective, etc.) to each word in a sentence. This plays a crucial role in tasks like **lemmatization**, where the root form of a word depends on its POS.

8.1 Why POS Tagging is Important

Consider the word "play." It can function as:

- Noun: "The **play** was wonderful."
- Verb: "Children **play** in the park."

Without POS tagging, NLP systems may struggle to interpret such words correctly. Lemmatization especially benefits from POS tags, since "playing" (verb) reduces to "play," while "play" (noun) remains unchanged.

8.2 POS Tags in NLTK

NLTK provides a wide variety of POS tags. Some common ones include:

- **NN**: Noun, singular (e.g., "dog")
- **NNS**: Noun, plural (e.g., "dogs")
- **VB**: Verb, base form (e.g., "run")
- **VBD**: Verb, past tense (e.g., "ran")
- **JJ**: Adjective (e.g., "beautiful")
- **RB**: Adverb (e.g., "quickly")
- **PRP**: Personal pronoun (e.g., "I", "he")
- **CC**: Coordinating conjunction (e.g., "and", "but")

NLTK uses the `Penn Treebank POS Tagset`, which contains over 30 tags.

8.3 Practical Example: Taj Mahal Sentence

Let us consider the example:

> *"Taj Mahal is a beautiful monument."*

We will tokenize this sentence and apply POS tagging.

```
import nltk
from nltk.tokenize import word_tokenize

nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

sentence = "Taj Mahal is a beautiful monument"
words = word_tokenize(sentence)

pos_tags = nltk.pos_tag(words)
print(pos_tags)
```

The output may look like:

```
[('Taj', 'NNP'), ('Mahal', 'NNP'), ('is', 'VBZ'),
 ('a', 'DT'), ('beautiful', 'JJ'), ('monument', 'NN')]
```

Here:

- `NNP` = Proper noun, singular
- `VBZ` = Verb, 3rd person singular present
- `DT` = Determiner
- `JJ` = Adjective
- `NN` = Noun, singular

8.4 Applying POS Tagging on Larger Texts

Let us take a paragraph from Dr. A.P.J. Abdul Kalam's speech and apply POS tagging.

```
from nltk.corpus import stopwords
nltk.download('stopwords')

paragraph = """I have three visions for India. In 3000 years of our history
    ,
people from all over the world have come and invaded us."""

sentences = nltk.sent_tokenize(paragraph)

for sent in sentences:
    words = nltk.word_tokenize(sent)
```

```
    words = [w for w in words if w.lower() not in stopwords.words('english')
        ]
    tags = nltk.pos_tag(words)
    print(tags)
```

This code filters stopwords and then applies POS tagging. Each word is annotated with its POS category.

8.5 Common Errors and Fixes

When using `nltk.pos_tag`, you may encounter the following error:

```
LookupError: Resource 'averaged_perceptron_tagger' not found.
```

To resolve this, download the tagger with:

```
nltk.download('averaged_perceptron_tagger')
```

8.6 Conclusion

POS tagging enriches text preprocessing by identifying grammatical roles of words. This enables more accurate downstream tasks like **lemmatization, sentiment analysis, and information extraction**. In the next chapter, we will move to **Named Entity Recognition (NER)**, where we identify real-world entities such as names, places, and organizations within text.

9 Named Entity Recognition (NER)

In the previous chapters, we explored stopwords and Part-of-Speech (POS) tagging. In this chapter, we will focus on **Named Entity Recognition (NER)**. NER is the process of identifying and classifying named entities in text into predefined categories such as *person names*, *locations*, *dates*, *organizations*, and *monetary values*.

9.1 What is Named Entity Recognition?

NER extends POS tagging by identifying real-world objects in text. For example, consider the sentence:

> *"The Eiffel Tower was built from 1887 to 1889 by French engineer Gustave Eiffel."*

In this sentence:

- **Eiffel Tower** → Location
- **1887 to 1889** → Date
- **Gustave Eiffel** → Person
- **French** → Nationality/Adjective

NER allows NLP systems to automatically tag and categorize these entities, providing semantic meaning to raw text.

9.2 Common Named Entity Categories

Some common NER categories include:

- **PERSON** – Names of people (e.g., "Albert Einstein")
- **GPE** – Geopolitical entities (countries, cities, states)
- **ORG** – Organizations (companies, institutions)
- **DATE** – Dates or periods ("21st century", "1887")
- **TIME** – Specific times ("5 p.m.")
- **MONEY** – Monetary values ("$1 million")
- **PERCENT** – Percentages ("50%")

9.3 Implementing NER with NLTK

Let us implement NER on the example sentence.

9.3.1 Tokenization and POS Tagging

```
import nltk
from nltk.tokenize import word_tokenize

nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

sentence = """The Eiffel Tower was built from 1887 to 1889
by French engineer Gustave Eiffel."""

# Tokenize words
words = word_tokenize(sentence)

# POS tagging
tagged_words = nltk.pos_tag(words)
print(tagged_words)
```

9.3.2 Applying Named Entity Recognition

To perform NER, we use `ne_chunk`, which builds a parse tree of entities.

```
nltk.download('maxent_ne_chunker')
nltk.download('words')

ner_tree = nltk.ne_chunk(tagged_words)
print(ner_tree)
```

9.3.3 Visualizing Named Entities

We can also draw the parse tree of named entities.

```
ner_tree.draw()
```

This will open a graphical window showing entities like `PERSON`, `GPE`, and `DATE` mapped to the words in the sentence.

9.4 Output Example

For our Eiffel Tower sentence, NER may produce:

```
(S
  (GPE Eiffel/NNP Tower/NNP)
  was/VBD
  built/VBN
  from/IN
  (DATE 1887/CD)
```

```
to/TO
(DATE 1889/CD)
by/IN
(GPE French/JJ)
engineer/NN
(PERSON Gustave/NNP Eiffel/NNP)
./.)
```

Here:

- `GPE` – Eiffel Tower, French
- `DATE` – 1887, 1889
- `PERSON` – Gustave Eiffel

9.5 Practical Notes

- If you encounter `LookupError`, ensure that the following resources are downloaded:

```
    nltk.download('maxent_ne_chunker')
    nltk.download('words')
```

- Not all entities are perfectly recognized; results depend on the quality of the trained model.
- For advanced applications, libraries like `spaCy` and `transformers` provide more accurate NER models.

9.6 Conclusion

Named Entity Recognition (NER) enhances text understanding by identifying real-world entities such as people, places, and dates. It goes beyond simple tokenization and POS tagging, enabling applications such as **information extraction, question answering, and knowledge graph construction**. In the next chapter, we will explore more advanced NLP techniques that build upon NER.

10 Sentiment Analysis: An End-to-End NLP Workflow

In the previous chapters, we have explored fundamental concepts of Natural Language Processing (NLP) such as tokenization, stemming, lemmatization, stopwords, and named entity recognition. In this chapter, we shift our focus to a practical problem: **Sentiment Analysis**. This task involves determining whether a given piece of text expresses a positive, negative, or neutral sentiment.

10.1 Problem Statement

Sentiment Analysis is a classic NLP problem where we aim to classify text based on the emotions or opinions it conveys. Examples include:

- "This food is delicious." → Positive
- "The movie was boring." → Negative
- "It is a sunny day." → Neutral

Let us explore how the concepts we have learned so far fit into the overall life cycle of an NLP project.

10.2 Corpus and Vocabulary

When working with text data, we typically begin with documents (denoted as $d_1, d_2, d_3, \dots$). A collection of such documents forms a **corpus**. From the corpus, we can extract the set of unique words, known as the **vocabulary**. This vocabulary is central for later steps like vectorization.

10.3 Step 1: Text Pre-processing (Part I)

The first step in solving any NLP problem is **text pre-processing**, also called text cleaning. Techniques include:

1. **Tokenization** – Splitting a paragraph into sentences, or sentences into words.
2. **Lowercasing** – Converting all words to lowercase (e.g., "The" and "the" are treated as the same word).
3. **Regular Expressions (Regex)** – Removing special characters, numbers, or unwanted patterns.

Example:

```
import re
import nltk
from nltk.tokenize import word_tokenize

sentence = "The Movie was AMAZING!!! :)"

# Tokenization
tokens = word_tokenize(sentence)

# Lowercasing
tokens = [w.lower() for w in tokens]

# Regex cleaning (remove non-alphabetic characters)
cleaned_tokens = [re.sub(r'[^a-z]', '', w) for w in tokens if w.isalpha()]

print(cleaned_tokens)
# Output: ['the', 'movie', 'was', 'amazing']
```

10.4 Step 2: Text Pre-processing (Part II)

After initial cleaning, further pre-processing improves the quality of text:

1. **Stemming** – Reducing words to their root form (e.g., "playing" → "play").
2. **Lemmatization** – More accurate root word extraction using vocabulary and grammar rules.
3. **Stopwords Removal** – Eliminating common words ("is", "the", "and") that carry little semantic value.

Example:

```
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer

nltk.download('stopwords')
nltk.download('wordnet')

tokens = ['this', 'movie', 'was', 'amazing']

# Remove stopwords
filtered_tokens = [w for w in tokens if w not in stopwords.words('english')
    ]

# Apply stemming
stemmer = PorterStemmer()
stems = [stemmer.stem(w) for w in filtered_tokens]

```

```
# Apply lemmatization
lemmatizer = WordNetLemmatizer()
lemmas = [lemmatizer.lemmatize(w) for w in filtered_tokens]

print("Filtered:", filtered_tokens)
print("Stems:", stems)
print("Lemmas:", lemmas)
```

10.5 Step 3: Text to Vectors

Once the text is cleaned, the next step is to convert it into a numerical representation (**vectors**) so that machine learning algorithms can process it. Techniques include:

- **One-Hot Encoding** – Each word represented as a binary vector (inefficient for large vocabularies).
- **Bag of Words (BoW)** – Counts word frequency in each document.
- **TF-IDF (Term Frequency – Inverse Document Frequency)** – Weighs words by importance in a corpus.
- **Word2Vec** – Uses neural networks to create dense vector embeddings of words.
- **Average Word2Vec** – Aggregates word embeddings to represent entire sentences or documents.

10.6 Step 4: Model Training

The generated vectors are then passed into machine learning algorithms such as:

- Naive Bayes
- Logistic Regression
- Support Vector Machines (SVM)
- Neural Networks (for advanced models)

Pipeline Illustration:

Raw Text → Pre-processing → Vectors → Model Training → Sentiment Prediction

10.7 Step 5: Prediction and Evaluation

Finally, the trained model is evaluated on test data to measure accuracy, precision, recall, and F1-score. The model predicts the sentiment of new, unseen text.

10.8 Conclusion

In this chapter, we have outlined the complete NLP pipeline for solving a real-world task: **Sentiment Analysis**. Each of the techniques we previously studied—tokenization, stopwords, stemming, and lemmatization—play a role in preparing text for modeling. In the next chapters, we will study each vectorization method (**One-Hot Encoding, Bag of Words, TF-IDF, Word2Vec**) in detail, with both theory and implementation examples.

11 One-Hot Encoding: Representing Text as Vectors

In the previous chapter, we introduced the overall workflow of a sentiment analysis problem. After pre-processing steps such as tokenization, stopwords removal, stemming, and lemmatization, the next step is to represent text in a numerical form that can be fed into machine learning models. In this chapter, we begin our study of vectorization techniques with the simplest one: **One-Hot Encoding**.

11.1 Concept of One-Hot Encoding

The main idea of one-hot encoding is to represent each word in the vocabulary as a binary vector.

- The length of the vector equals the size of the vocabulary V.
- Each position in the vector corresponds to one unique word from the vocabulary.
- For a given word, the index corresponding to that word is marked with 1, while all other positions are 0.

Example Corpus

Let us consider the following three documents:

$$D_1: \text{“The food is good”}$$
$$D_2: \text{“The food is bad”}$$
$$D_3: \text{“Pizza is amazing”}$$

The unique vocabulary extracted from this corpus is:

$$\{\text{the, food, is, good, bad, pizza, amazing}\}$$

Thus, the vocabulary size is $V = 7$.

11.2 Word Representation

For each word, we create a binary vector of length 7. For example:

$$\begin{aligned}
\text{the} &\rightarrow [1,0,0,0,0,0,0] \\
\text{food} &\rightarrow [0,1,0,0,0,0,0] \\
\text{is} &\rightarrow [0,0,1,0,0,0,0] \\
\text{good} &\rightarrow [0,0,0,1,0,0,0] \\
\text{bad} &\rightarrow [0,0,0,0,1,0,0] \\
\text{pizza} &\rightarrow [0,0,0,0,0,1,0] \\
\text{amazing} &\rightarrow [0,0,0,0,0,0,1]
\end{aligned}$$

11.3 Sentence Representation

Now, each sentence (document) can be represented as a sequence of word vectors.

Example: Document D_1

"The food is good" consists of 4 words:

$$\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0
\end{bmatrix}$$

The shape of this representation is 4×7 (4 words, vocabulary size 7).

Example: Document D_2

"The food is bad" becomes:

$$\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0
\end{bmatrix}$$

Again, shape $= 4 \times 7$.

11.4 Implementation in Python

```python
from sklearn.preprocessing import OneHotEncoder
import numpy as np

# Define corpus
corpus = ["The food is good", "The food is bad", "Pizza is amazing"]

# Tokenize into words
tokens = [sentence.lower().split() for sentence in corpus]
vocab = sorted(set(word for sent in tokens for word in sent))

print("Vocabulary:", vocab)

```

```
# Map each word to index
word_to_index = {word: i for i, word in enumerate(vocab)}

# One-hot encode manually
def one_hot_encode(sentence, vocab, word_to_index):
    vectors = []
    for word in sentence:
        vector = [0]*len(vocab)
        vector[word_to_index[word]] = 1
        vectors.append(vector)
    return np.array(vectors)

for i, sent in enumerate(tokens):
    print(f"D{i+1}:\n", one_hot_encode(sent, vocab, word_to_index))
```

11.5 Advantages and Disadvantages of One-Hot Encoding

In the previous section, we introduced the concept of one-hot encoding and showed how words and sentences can be represented as binary vectors. Before moving on to more advanced techniques, it is important to understand the strengths and weaknesses of this method.

11.5.1 Advantages

- **Simplicity and Ease of Implementation:** One-hot encoding is extremely easy to implement. Libraries such as `scikit-learn` provide `OneHotEncoder`, and in `pandas`, the function `pd.get_dummies()` makes it straightforward to apply this transformation.

- **Interpretability:** Each dimension of the vector directly corresponds to a specific word in the vocabulary, making it intuitive to understand.

11.5.2 Disadvantages

Despite its simplicity, one-hot encoding suffers from several critical limitations that make it unsuitable for most NLP applications:

1. **High Dimensionality and Sparsity:** For large vocabularies (e.g., 50,000 words), each vector becomes extremely large with mostly zeros. Such *sparse matrices* are computationally inefficient and memory intensive. Sparse representations also increase the risk of **overfitting**, where a model performs well on training data but generalizes poorly to unseen data.

2. **Variable-Length Sentences:** Each word is encoded as a vector of size $|V|$, but sentences may contain different numbers of words. For example:

$$D_1 : 4 \times 7 \qquad D_2 : 4 \times 7 \qquad D_3 : 3 \times 7$$

   Machine learning models generally require fixed-size input features, which one-hot encoding does not guarantee.

3. **No Semantic Information:** One-hot vectors treat all words as independent and equidistant. For example:

$$\text{food} = [1, 0, 0], \quad \text{pizza} = [0, 1, 0], \quad \text{burger} = [0, 0, 1]$$

   Geometrically, these points are orthogonal and equally distant, even though *food* and *pizza* are semantically related. Thus, one-hot encoding fails to capture similarity or contextual meaning.

4. **Out-of-Vocabulary (OOV) Problem:** If a new word appears in the test data (e.g., "burger"), but it was not present in the training vocabulary, the model has no way to represent it. This severely limits generalization.

5. **Scalability Issues:** In real-world corpora, vocabularies are very large. With tens of thousands of unique words, one-hot encoding becomes computationally impractical.

11.5.3 Summary

- One-hot encoding is simple and interpretable.
- However, it leads to high-dimensional sparse vectors, does not capture semantic similarity, fails with unseen words, and struggles with variable-length inputs.

These disadvantages motivate the need for more efficient and informative text representation methods. In the next chapter, we will study the **Bag of Words (BoW)** approach, which addresses some of these limitations by providing fixed-length sentence representations.

11.6 Limitations

Although one-hot encoding is simple and intuitive, it has several drawbacks:

- **High Dimensionality:** Vocabulary sizes can be in the tens of thousands, leading to extremely large vectors.
- **Sparsity:** Most entries in the vector are zero, wasting storage and computation.
- **No Semantic Information:** Words like "good" and "amazing" are represented as orthogonal vectors, even though they are semantically similar.
- **Variable Length Sentences:** Different sentences lead to matrices of different shapes, which complicates direct use in models.

11.7 Conclusion

One-hot encoding provides the foundation for representing text numerically, but it is rarely used in modern NLP because of its limitations. In the next chapter, we will study improved techniques such as **Bag of Words** and **TF-IDF**, which overcome many of these issues.

12 Bag of Words (BoW) in Natural Language Processing

After discussing one-hot encoding and its limitations, we now introduce **Bag of Words (BoW)**, a simple yet powerful method for representing text as numerical vectors. This technique is widely used in tasks such as **sentiment classification**, **spam detection**, and other text classification problems.

12.1 Introduction

Bag of Words is a method that converts text into fixed-length vectors based on the presence or frequency of words. Unlike one-hot encoding, BoW considers the entire sentence as a unit rather than creating a vector for each word separately. This makes it more suitable for machine learning models.

12.2 Step 1: Collecting the Dataset

Consider a toy dataset with three positive statements:

- Sentence 1: `He is a good boy`
- Sentence 2: `She is a good girl`
- Sentence 3: `Boy and girl are good`

All sentences are labeled as **positive** (output = 1).

12.3 Step 2: Preprocessing

Preprocessing is crucial in NLP. The steps include:

1. **Lowercasing:** Convert all words to lowercase to treat "Boy" and "boy" as the same.
2. **Stopword Removal:** Remove common words such as *he, she, is, a, and, are*, which do not contribute meaningfully to classification.

After preprocessing:

- Sentence 1: `good boy`
- Sentence 2: `good girl`
- Sentence 3: `boy girl good`

12.4 Step 3: Building the Vocabulary

The vocabulary consists of all unique words in the corpus:

$$V = \{\text{good}, \text{boy}, \text{girl}\}$$

Tracking word frequencies:

- `good`: 3 times
- `boy`: 2 times
- `girl`: 2 times

12.5 Step 4: Converting Sentences into Vectors

Each sentence is represented as a vector of size $|V| = 3$.

- Sentence 1 (`good boy`) $\rightarrow [1, 1, 0]$
- Sentence 2 (`good girl`) $\rightarrow [1, 0, 1]$
- Sentence 3 (`boy girl good`) $\rightarrow [1, 1, 1]$

12.6 Step 5: Variants of Bag of Words

1. **Binary Bag of Words:** Each entry is 1 if the word is present, 0 if absent.
2. **Count-based Bag of Words:** Each entry reflects the number of times a word appears in the sentence. Example: `good good girl` $\rightarrow [2, 0, 1]$

12.7 Python Implementation

```python
from sklearn.feature_extraction.text import CountVectorizer

# Sample dataset
corpus = [
    "He is a good boy",
    "She is a good girl",
    "Boy and girl are good"
]

# Preprocess with stopword removal (optional)
vectorizer = CountVectorizer(stop_words='english')
X = vectorizer.fit_transform(corpus)

print(vectorizer.get_feature_names_out())
print(X.toarray())
```

Output:

```
['boy', 'girl', 'good']
[[1 0 1]
 [0 1 1]
 [1 1 1]]
```

12.8 Advantages and Disadvantages of Bag of Words

Now that we understand how Bag of Words converts text into vectors, it is important to discuss its strengths and limitations. This helps in comparing it with one-hot encoding and understanding its applications in NLP.

12.8.1 Advantages

1. **Simple and Intuitive:** Bag of Words is easy to implement and understand. It provides a straightforward way to convert text into numerical vectors for machine learning algorithms.

2. **Fixed-size Input for ML Algorithms:** Unlike one-hot encoding, BoW generates a fixed-length vector for every sentence, regardless of its word count. This ensures that inputs are consistent in size, which is crucial for machine learning models.

3. **Captures Word Frequency:** The vector representation can either indicate the presence of a word (binary BoW) or its frequency (count-based BoW), providing more information about the text compared to simple one-hot encoding.

12.8.2 Disadvantages

1. **Sparse Matrices:** Like one-hot encoding, BoW often results in vectors with many zeros, especially for large vocabularies. This can lead to overfitting in machine learning models if not handled properly.

2. **Loss of Word Order:** BoW does not capture the order of words in a sentence. For example, "good boy" and "boy good" are treated similarly, which can affect the semantic meaning of the sentence.

3. **Out of Vocabulary (OOV) Words:** If a new word appears in the test data that was not in the training vocabulary, it will be ignored. This can be problematic if the new word carries important meaning.

4. **Limited Semantic Understanding:** BoW only captures whether a word is present and its frequency but does not understand context or relationships between words. For example, "The food is good" and "The food is not good" will have very similar vector representations, despite having opposite meanings.

12.8.3 Illustrative Example: Semantic Issue

Consider the two sentences:

- Sentence 1: `The food is good`
- Sentence 2: `The food is not good`

After converting to BoW vectors (with stop words removed):

$$\text{Sentence 1} \rightarrow [1, 1, 1] \quad \text{Sentence 2} \rightarrow [1, 0, 1]$$

Using cosine similarity, these vectors would appear very close in vector space, implying similarity, even though their actual meanings are opposite. This illustrates the limitation of BoW in capturing semantic meaning.

12.9 Summary

- Bag of Words improves on one-hot encoding by generating fixed-length vectors for sentences and allowing frequency-based representation.
- It still suffers from sparsity, ignores word order, and has limited semantic understanding.
- Despite these drawbacks, Bag of Words is simple and effective for tasks like sentiment analysis, spam detection, and text classification.
- These limitations motivate more advanced techniques like **Word2Vec**, **GloVe**, and **embedding methods** in deep learning.

In the next section, we will implement Bag of Words practically using `scikit-learn` and see how to convert text data into feature vectors.

13 Bag of Words Practical Implementation

In this chapter, we will implement the Bag of Words model using a real-world dataset for spam classification. The dataset contains two columns: `label` (ham or spam) and `message`. We will preprocess the data and convert the text messages into vectors using the Bag of Words approach.

13.1 Reading the Dataset

We begin by importing the necessary libraries and reading the dataset:

```
import pandas as pd

# Reading the dataset with tab separator and custom column names
messages = pd.read_csv(
    'spam_classification_master/SMS_spam_collection.txt',
    sep='\t',
    names=['label', 'message']
)

# Display the first few rows
print(messages.head())
```

13.2 Data Cleaning and Preprocessing

We clean the data by removing special characters, converting text to lowercase, removing stopwords, and applying stemming:

```
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer

nltk.download('stopwords')

ps = PorterStemmer()
corpus = []

for i in range(len(messages)):
    # Remove special characters
```

```
    review = re.sub('[^a-zA-Z]', ' ', messages['message'][i])
    # Convert to lowercase
    review = review.lower()
    # Tokenize the message
    review = review.split()
    # Remove stopwords and apply stemming
    review = [ps.stem(word) for word in review if word not in stopwords.
        words('english')]
    # Join back into a single string
    review = ' '.join(review)
    corpus.append(review)

# Display cleaned corpus
print(corpus[:5])
```

13.3 Creating the Bag of Words Model

We now convert the cleaned text into vectors using `CountVectorizer`:

```
from sklearn.feature_extraction.text import CountVectorizer

# Initialize CountVectorizer
cv = CountVectorizer(max_features=2500, binary=True)

# Fit and transform the corpus
X = cv.fit_transform(corpus).toarray()

# Display the shape of the resulting feature matrix
print(X.shape)
```

The `max_features` parameter selects the top words by frequency, and `binary=True` ensures a binary Bag of Words representation.

13.4 Summary

We have successfully:

- Read and inspected a dataset.
- Cleaned and preprocessed text data.
- Converted text into numerical vectors using Bag of Words.

These vectors can now be used as input features for machine learning models to classify messages as spam or ham. As an exercise, try using lemmatization instead of stemming to potentially improve classification accuracy.

14 N-Grams in Natural Language Processing

14.1 Introduction

In the previous chapters, we learned about the Bag of Words (BoW) model for text vectorization. While BoW is powerful, it often fails to capture the contextual meaning of sentences. For example, the sentences:

- **Sentence 1:** "Food is good"
- **Sentence 2:** "Food is not good"

Although both sentences share most of their words, their meanings are completely opposite. Using only unigrams (individual words), the resulting vectors will look very similar, making it difficult for models to distinguish them.

This problem motivates the use of **N-grams**, which help capture word combinations and preserve contextual information.

14.2 Understanding N-Grams

An **N-gram** is a contiguous sequence of N items (usually words) from a given text. For example:

- **Unigram ($N = 1$):** {food, good, not}
- **Bigram ($N = 2$):** {food good, food not, not good}
- **Trigram ($N = 3$):** {food not good}

By including word pairs or triplets, N-grams help us better distinguish between sentences with opposite meanings.

14.3 Example: Unigram vs. Bigram

Let us consider the two sentences again:

- Sentence 1: "Food is good"
- Sentence 2: "Food is not good"

Unigram Representation

If we remove stopwords such as "is", our vocabulary becomes:

$$\{food, not, good\}$$

The vectors are:

$$\text{Sentence 1: } [1, 0, 1] \quad \text{Sentence 2: } [1, 1, 1]$$

The only difference is in the "not" word, making them appear quite similar in vector space.

Bigram Representation

Now, let us include bigrams:

$$\{food, not, good, food\ good, food\ not, not\ good\}$$

The vectors become:

$$\text{Sentence 1: } [1, 0, 1, 1, 0, 0]$$

$$\text{Sentence 2: } [1, 1, 1, 0, 1, 1]$$

Here, the difference is much clearer, since "food not" and "not good" appear only in the second sentence.

14.4 Using N-Grams in Scikit-Learn

Scikit-Learn provides an easy way to include N-grams using the `ngram_range` parameter in vectorizers like `CountVectorizer` and `TfidfVectorizer`.

```
from sklearn.feature_extraction.text import CountVectorizer

sentences = [
    "Food is good",
    "Food is not good"
]

# Unigram + Bigram representation
cv = CountVectorizer(ngram_range=(1,2), stop_words='english')
X = cv.fit_transform(sentences).toarray()

print(cv.get_feature_names_out())
print(X)
```

Explanation of Parameters

- `ngram_range=(1,1)`: Only unigrams.
- `ngram_range=(1,2)`: Includes unigrams and bigrams.
- `ngram_range=(1,3)`: Includes unigrams, bigrams, and trigrams.
- `ngram_range=(2,3)`: Includes only bigrams and trigrams.

14.5 Summary

- Bag of Words alone cannot capture word order or negation.
- N-grams help preserve contextual meaning by considering combinations of words.
- Unigrams, bigrams, and trigrams can be combined depending on the problem.
- In Scikit-Learn, N-grams can be implemented easily using `ngram_range`.

Thus, N-grams provide richer feature representations and allow models to better distinguish between sentences with subtle but important differences in meaning.

15 Practical Implementation of N-grams

In the previous chapter, we introduced the intuition behind N-grams and discussed why they are necessary to capture contextual meaning beyond individual words. In this chapter, we will implement N-grams practically using Python, the `nltk` library, and `scikit-learn`'s `CountVectorizer`.

We continue with the same dataset used earlier for spam classification, where the task is to classify messages as `spam` or `ham`.

15.1 Revisiting Bag of Words

Recall that in the Bag of Words (BoW) model, we extracted the most frequent words from the text corpus. For example, we might limit our features to the top 100 most frequently occurring words. These words form our vocabulary.

To check the vocabulary created by `CountVectorizer`, we can simply call the `vocabulary_` attribute:

```
from sklearn.feature_extraction.text import CountVectorizer

# Initialize CountVectorizer for top 100 words
cv = CountVectorizer(max_features=100)

# Fit and transform the corpus
X = cv.fit_transform(corpus).toarray()

# Display vocabulary
print(cv.vocabulary_)
```

This returns a dictionary where keys are words and values are their corresponding column indices in the feature matrix. For example:

```
{'ask': 0, 'babe': 1, 'call': 2, ... }
```

15.2 The `ngram_range` Parameter

The crucial parameter for N-grams in `CountVectorizer` is `ngram_range`.

- `(1,1)`: Only unigrams (single words).
- `(1,2)`: Unigrams + Bigrams (single words and pairs of consecutive words).

- `(1,3)`: Unigrams + Bigrams + Trigrams.
- `(2,2)`: Only Bigrams.
- `(2,3)`: Bigrams + Trigrams.

15.3 Implementing Unigrams and Bigrams

We begin with unigrams and then extend to bigrams:

```
# Unigrams only
cv_uni = CountVectorizer(max_features=200, ngram_range=(1,1))
X_uni = cv_uni.fit_transform(corpus).toarray()
print(list(cv_uni.vocabulary_.items())[:20])

# Unigrams + Bigrams
cv_bi = CountVectorizer(max_features=500, ngram_range=(1,2))
X_bi = cv_bi.fit_transform(corpus).toarray()
print(list(cv_bi.vocabulary_.items())[:20])
```

Output for bigrams will include combinations like `please call`, `good morning`, or `customer service`, depending on frequency in the dataset.

15.4 Exploring Higher-order N-grams

We can extend this to trigrams:

```
# Trigrams only
cv_tri = CountVectorizer(max_features=500, ngram_range=(3,3))
X_tri = cv_tri.fit_transform(corpus).toarray()
print(list(cv_tri.vocabulary_.items())[:20])
```

This will capture phrases of three words, such as `call customer service`.

15.5 Hyperparameter Tuning

The choice of `ngram_range` and `max_features` is a hyperparameter tuning task. A good practice is:

1. Start with `ngram_range=(1,1)`.
2. If accuracy is unsatisfactory, extend to `(1,2)`.
3. If necessary, try `(1,3)` or adjust `max_features`.

These combinations allow the model to capture more context, at the cost of higher dimensionality.

15.6 Summary

In this chapter, we have:

- Reviewed the Bag of Words vocabulary extraction.
- Introduced the `ngram_range` parameter in `CountVectorizer`.
- Implemented unigrams, bigrams, and trigrams.
- Highlighted the role of hyperparameters (`max_features` and `ngram_range`).

N-grams are a powerful way to capture context and improve classification accuracy when sentences differ in meaning despite having similar words. In the next chapter, we will move on to the intuition behind **TF-IDF (Term Frequency - Inverse Document Frequency)** and later implement it in Python.

16 TF-IDF Representation in NLP

In this chapter, we continue our exploration of techniques in **Natural Language Processing (NLP)**. In the previous chapter, we introduced the concept of **N-grams** as a method of text representation. We now turn our attention to another widely used and effective representation method: the **TF-IDF (Term Frequency – Inverse Document Frequency)** model.

16.1 Introduction to TF-IDF

TF-IDF is a statistical measure designed to evaluate the importance of a word within a document relative to a collection (or corpus) of documents. It consists of two fundamental components:

- **Term Frequency (TF)**: Quantifies how frequently a given term appears in a document.
- **Inverse Document Frequency (IDF)**: Assigns lower weights to commonly occurring words across documents, while assigning higher weights to rarer, more informative terms.

16.2 Illustrative Example

Consider the following preprocessed sentences (all lowercased, with stopwords removed):

- S_1: `good boy`
- S_2: `good girl`
- S_3: `boy girl good`

From these sentences, our vocabulary is defined as: `good, boy, girl`.

16.3 Term Frequency (TF)

The term frequency of a word is given by:

$$TF(t, d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

Using this formula, the TF values for our example sentences are as follows:

Sentence	good	boy	girl
S_1	$\frac{1}{2}$	$\frac{1}{2}$	0
S_2	$\frac{1}{2}$	0	$\frac{1}{2}$
S_3	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$

16.4 Inverse Document Frequency (IDF)

The IDF component adjusts for the fact that some words appear frequently across documents, thereby carrying less discriminative power. The formula is:

$$IDF(t) = \log\left(\frac{N}{1 + \text{Number of documents containing term } t}\right)$$

where N denotes the total number of documents. Here, $N = 3$.

Word	IDF Value
good	$\log\left(\frac{3}{3}\right) = 0$
boy	$\log\left(\frac{3}{2}\right)$
girl	$\log\left(\frac{3}{2}\right)$

16.5 TF-IDF Calculation

The final TF-IDF score is obtained by multiplying the two measures:

$$TF\text{-}IDF(t, d) = TF(t, d) \times IDF(t)$$

The resulting TF-IDF vectors for the three sentences are:

Sentence	good	boy	girl
S_1	0	$\frac{1}{2} \cdot \log\frac{3}{2}$	0
S_2	0	0	$\frac{1}{2} \cdot \log\frac{3}{2}$
S_3	0	$\frac{1}{3} \cdot \log\frac{3}{2}$	$\frac{1}{3} \cdot \log\frac{3}{2}$

16.6 Conclusion

This example demonstrates how TF-IDF balances **local importance** (frequency of a word within a document) with **global distinctiveness** (rarity across the corpus). Words that occur in every document, such as *good*, are assigned zero weight, while words that differentiate documents, such as *boy* and *girl*, receive higher importance.

In the next chapter, we will analyze the **advantages and limitations of TF-IDF** and compare it with more advanced vectorization approaches, such as *Word2Vec*.

17 Term Frequency–Inverse Document Frequency (TF–IDF)

17.1 Introduction

In the previous chapter, we discussed Bag-of-Words (BoW) and N-grams. While BoW represents text as fixed-length vectors of word counts, it fails to capture the relative importance of words. To overcome this limitation, a more refined technique is used, namely **Term Frequency–Inverse Document Frequency (TF–IDF)**.

TF–IDF assigns weights to words based not only on their frequency within a document but also on their relative importance across the entire corpus.

17.2 Mathematical Formulation

Let t be a term, d a document, and D the entire collection of documents.

- **Term Frequency (TF)** measures how frequently a term occurs in a document:

$$TF(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

  where $f_{t,d}$ is the raw count of term t in document d.

- **Inverse Document Frequency (IDF)** reduces the weight of common words that appear in many documents:

$$IDF(t, D) = \log\left(\frac{|D|}{1 + |\{d \in D : t \in d\}|}\right)$$

- **TF–IDF score** is given by:

$$TF\text{-}IDF(t, d, D) = TF(t, d) \times IDF(t, D)$$

17.3 Advantages of TF–IDF

1. **Simplicity:** Like BoW, the representation is intuitive and easy to implement.
2. **Fixed Input Size:** The feature vectors are still based on the vocabulary size, ensuring consistency across documents.
3. **Word Importance is Captured:** Unlike BoW, TF–IDF captures the relative importance of words. Words that appear in all documents (e.g., "good" in every sentence) are given less weight, while unique or rare words (e.g., "boy" or "girl" in specific sentences) receive higher weights. This ensures that the context is preserved, which improves model performance.

17.4 Example

Consider the following three sentences:

- Sentence 1: `Good boy`
- Sentence 2: `Good girl`
- Sentence 3: `Boy girl good`
- In a Bag-of-Words model, each word is given equal importance, represented by either 1 (present) or 0 (absent).
- In TF–IDF, the word `good` appears in all three sentences and thus receives a low weight. In contrast, `boy` and `girl`, which are more context-specific, receive higher scores.

This highlights how TF–IDF captures contextual importance beyond simple frequency.

17.5 Disadvantages of TF–IDF

1. **Sparsity:** The representation still produces high-dimensional sparse vectors with many zeros.
2. **Out-of-Vocabulary (OOV):** Words in the test set that were not present in the training vocabulary are ignored, which can reduce effectiveness.

17.6 Python Implementation

Below is an example of how TF–IDF can be implemented in Python using `scikit-learn`:

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Sample corpus
corpus = [
    "Good boy",
    "Good girl",
    "Boy girl good"
]

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

# Fit and transform the corpus
X = vectorizer.fit_transform(corpus)

# Display results
print("Vocabulary:", vectorizer.get_feature_names_out())
print("TF-IDF Matrix:\n", X.toarray())
```

Listing 17.1: TF–IDF Example in Python

17.7 Conclusion

TF–IDF improves upon Bag-of-Words by capturing word importance across documents, allowing models to better distinguish meaningful words from common ones. Although sparsity and OOV issues still exist, TF–IDF generally provides superior performance in text representation tasks.

18 Practical Implementation of TF–IDF with Python

18.1 Introduction

In the previous chapters, we discussed the mathematical formulation, advantages, and disadvantages of TF–IDF. Now we will implement TF–IDF practically using Python. We will use a sample dataset (Spam Collection) and create TF–IDF vectors from it.

We will also use **lemmatization** instead of stemming for preprocessing to demonstrate another approach to text normalization.

18.2 Data Preprocessing

First, we load the dataset and perform standard text preprocessing:

- Convert all text to lowercase.
- Remove stopwords.
- Apply lemmatization using WordNetLemmatizer.

```
import pandas as pd
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import re

# Load dataset
data = pd.read_csv('spam_collection.csv')
corpus = []

lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))

for text in data['message']:
    # Lowercase and remove non-alphabetic characters
    text = re.sub('[^a-zA-Z]', ' ', text).lower()
    words = text.split()
    words = [lemmatizer.lemmatize(w) for w in words if w not in stop_words]
    corpus.append(' '.join(words))
```

Listing 18.1: Text Preprocessing with Lemmatization

18.3 TF–IDF Vectorization

Next, we initialize the `TfidfVectorizer` from `sklearn` and transform the corpus into TF–IDF vectors.

```
from sklearn.feature_extraction.text import TfidfVectorizer
import numpy as np

# Initialize TF-IDF Vectorizer
tfidf_vectorizer = TfidfVectorizer(max_features=100) # top 100 frequent
    words

# Fit and transform the corpus
X = tfidf_vectorizer.fit_transform(corpus)

# Convert sparse matrix to array
X_array = X.toarray()

# Display vocabulary and TF-IDF matrix
print("Vocabulary:", tfidf_vectorizer.get_feature_names_out())
print("TF-IDF Matrix:\n", X_array)
```

Listing 18.2: TF-IDF Vectorization

18.4 N-Gram TF–IDF

We can also create TF–IDF vectors for **n-grams** to capture sequences of words:

```
# Initialize TF-IDF Vectorizer with bi-grams
tfidf_vectorizer_ngram = TfidfVectorizer(max_features=100, ngram_range
    =(2,2))

# Fit and transform the corpus
X_ngram = tfidf_vectorizer_ngram.fit_transform(corpus)
X_ngram_array = X_ngram.toarray()

# Display vocabulary and TF-IDF matrix for n-grams
print("Bi-gram Vocabulary:", tfidf_vectorizer_ngram.get_feature_names_out()
    )
print("Bi-gram TF-IDF Matrix:\n", X_ngram_array)
```

Listing 18.3: N-Gram TF-IDF Vectorization

18.5 Observations

- The TF–IDF vectors contain decimal values representing the importance of words in each document.
- Unlike Bag-of-Words, TF–IDF does not use binary counts (0 or 1). It gives meaningful weight based on frequency and document occurrence.

- Using n-grams, we can capture phrases instead of individual words, which is useful for understanding context.

18.6 Conclusion

After creating the TF–IDF vectors, these can be used as input features for any machine learning algorithm. Before applying a model, standard steps like train-test split, scaling, or further feature engineering can be performed. TF–IDF provides a simple yet powerful way to convert text into meaningful numerical representations while capturing word importance and context.

19 Introduction to Word Embeddings

19.1 Overview

In natural language processing (NLP) for machine learning, **word embeddings** are a method of representing words as real-valued vectors in a continuous vector space. The idea is that words with similar meanings will be closer to each other in this vector space, enabling algorithms to capture semantic relationships.

According to Wikipedia, "Word embedding is a term used for the representation of words for text analysis, typically in the form of real-valued vectors that encode the meaning of the word such that words closer in the vector space are expected to be similar in meaning."

19.2 Motivation and Intuition

Consider three words: `happy`, `excited`, and `angry`. Using word embeddings, we can convert these words into vectors. When plotted (using dimensionality reduction techniques such as PCA), the vectors for `happy` and `excited` will be close, indicating similarity, while `angry` will be farther apart, reflecting opposite meaning.

This demonstrates that word embeddings provide a meaningful representation of words beyond simple presence or frequency counts.

19.3 Types of Word Embedding Techniques

Word embedding techniques can be broadly divided into two categories:

1. **Count or Frequency-based Methods:** These are classical techniques that convert words into vectors based on their frequency in the corpus.

    - One-Hot Encoding
    - Bag of Words
    - TF–IDF

    These methods are intuitive but suffer from several limitations such as sparsity and inability to capture semantic similarity effectively.

2. **Deep Learning Trained Models:** These methods leverage neural networks to train word representations and generally provide higher accuracy. The most widely used model is **Word2Vec**.

19.4 Word2Vec: A Deep Learning Approach

Word2Vec is a word embedding technique that efficiently converts words into vectors while addressing the limitations of count-based methods:

- Captures semantic similarity between words.
- Reduces sparsity in representations.
- Provides dense vector representations suitable for machine learning algorithms.

Word2Vec architectures include:

1. **Continuous Bag of Words (CBOW):** Predicts a target word from its surrounding context words.
2. **Skip-Gram:** Predicts surrounding context words given a target word.

Both architectures are trained using neural networks and optimize a loss function using optimizers to generate meaningful word vectors.

19.5 Summary

- All previous techniques for converting words into vectors (One-Hot Encoding, Bag of Words, TF–IDF) fall under the category of word embeddings.
- Deep learning-based embeddings like Word2Vec overcome most limitations of classical methods and provide robust representations for NLP tasks.
- In upcoming chapters, we will explore Word2Vec in detail, including its architecture, training process, and pre-trained models.

20 Word2Vec: Deep Learning-Based Word Embeddings

20.1 Overview

Word2Vec is a deep learning-based word embedding technique introduced by Google in 2013. Its primary goal is to convert words into continuous-valued vectors while preserving semantic meaning: similar words are mapped to nearby vectors, and dissimilar or opposite words are farther apart in the vector space.

Word2Vec uses neural networks to learn word associations from large corpora of text. Once trained, it can:

- Detect synonyms and antonyms
- Suggest additional words in partial sentences
- Enable semantic vector operations (e.g., `king - man + woman = queen`)

20.2 Feature Representation

Each word in the vocabulary is represented as a vector of numerical features. For example, consider a simplified vocabulary with words like `boy`, `girl`, `king`, `queen`, `apple`, `mango`.

Each word is mapped to a vector of size n (e.g., 300 dimensions in Google's pre-trained Word2Vec). These vectors encode abstract relationships based on features such as:

- Gender (e.g., `boy` = -1, `girl` = +1)
- Royalty (e.g., `king` = 0.95, `queen` = 0.96)
- Age, food category, or other latent features

Values are assigned based on the word's relationship to these features, learned automatically during training. Features are not manually labeled but emerge from the data.

20.3 Vector Arithmetic and Semantic Relationships

Word2Vec allows meaningful vector operations. For example:

$$\text{king} - \text{man} + \text{woman} \approx \text{queen}$$

This works because the vectors encode semantic relationships, and operations like subtraction and addition reflect analogies.

20.4 Cosine Similarity

To measure similarity between words, Word2Vec uses **cosine similarity**, defined as:

$$\text{cosine\_similarity}(\mathbf{v}_1, \mathbf{v}_2) = \frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{\|\mathbf{v}_1\|\|\mathbf{v}_2\|}$$

Distance between two vectors is calculated as:

$$\text{distance} = 1 - \text{cosine\_similarity}$$

- Distance near 0: vectors are very similar
- Distance near 1: vectors are dissimilar

This metric is widely used in NLP tasks, including recommendation systems, synonym detection, and semantic search.

20.5 Summary

- Word2Vec generates dense vector representations of words, capturing semantic relationships and analogies.
- Each word is represented as a high-dimensional feature vector, learned automatically from data.
- Cosine similarity is used to measure word similarity and vector closeness.
- Semantic operations such as `king - man + woman = queen` illustrate the power of Word2Vec embeddings.

In the next chapter, we will discuss **Word2Vec training architectures** (CBOW and Skip-Gram) and demonstrate practical implementation using Python.

21 Word2Vec: Continuous Bag of Words (CBOW)

21.1 Introduction

Word2Vec is a popular technique to learn vector representations of words. It comes in two main variants:

- **CBOW (Continuous Bag of Words)**: Predicts a word based on its surrounding context.
- **Skip-Gram**: Predicts the surrounding context given a word.

In this chapter, we focus on CBOW and how it is trained using a neural network.

21.2 Corpus and Window Size

Consider a simple example corpus:

```
"I neuron company is related to data science"
```

Window size (w) determines how many words around the target word are considered as input. For CBOW, we always choose an **odd number** to have a clear central word. Example: $w = 5$ (2 words before, 2 words after, 1 center word).

21.3 Generating Input-Output Pairs

Sliding the window over the corpus generates training pairs. **CBOW input**: context words **CBOW output**: center word

Context (Input)	Center (Output)
I, neuron, company, related	is
neuron, company, is, data	related
company, is, related, science	data

Table 21.1: Example CBOW input-output pairs

21.4 One-Hot Encoding

Each word in the vocabulary is represented as a one-hot vector.

- Vocabulary: {I, neuron, company, is, related, to, data, science}

- Example encodings:
  - I $\rightarrow [1, 0, 0, 0, 0, 0, 0, 0]$
  - neuron $\rightarrow [0, 1, 0, 0, 0, 0, 0, 0]$
  - company $\rightarrow [0, 0, 1, 0, 0, 0, 0, 0]$

21.5 Neural Network Architecture

- **Input layer:** Context words (one-hot encoded)
- **Hidden layer:** Size = window size (determines vector dimension)
- **Output layer:** Center word (one-hot encoded)

Fully connected neural network: every input node connects to all hidden nodes, and every hidden node connects to all output nodes.

21.5.1 Training Steps

1. Forward propagation: input $\rightarrow$ hidden $\rightarrow$ output $\rightarrow$ predicted probabilities
2. Loss function: compares predicted output $\hat{y}$ with true output y Example: if "is" is the center word, $y = [0, 0, 0, 1, 0, 0, 0, 0]$
3. Backward propagation: adjust weights to minimize loss

21.6 Word Vectors

After training, each word is represented as a vector of dimension equal to the hidden layer size. Example: window size = 5 $\rightarrow$ each word $\rightarrow$ 5-dimensional vector.

These vectors capture semantic relationships:

- Words in similar context have vectors close to each other
- Vector arithmetic: `king - man + woman` $\approx$ `queen`

21.7 Key Notes

- CBOW predicts a word from its context; Skip-Gram predicts context from a word
- Pre-trained models (e.g., Google Word2Vec) use huge corpora (3 billion words) and vector size 300
- Window size affects the dimension and quality of word vectors

22 Word2Vec: Skip-Gram Architecture

22.1 Introduction

Skip-Gram is another popular architecture of Word2Vec. Unlike CBOW, which predicts the center word from its context, Skip-Gram predicts the surrounding context words given the center word.

We will use the same example corpus:

```
"Enron company is related to data science"
```

22.2 Input-Output Pairs

In Skip-Gram, the central word becomes the input, and the surrounding words within a window size w are the outputs.

Input (Center Word)	Output (Context Words)
company	Enron, is, related, to
is	company, related, to, data
related	is, to, data, science

Table 22.1: Example Skip-Gram input-output pairs (window size = 5)

22.3 Neural Network Architecture

- **Input layer:** One-hot encoded center word vector
- **Hidden layer:** Size = window size (determines word vector dimension)
- **Output layer:** Context words (one-hot encoded)

Fully connected layers connect all nodes in one layer to all nodes in the next.

22.3.1 Weight Matrices

- Input $\rightarrow$ Hidden: $|V| \times N$ matrix, where $|V|$ = vocabulary size, N = hidden layer size
- Hidden $\rightarrow$ Output: $N \times |V|$ matrix

22.4 Training Steps

1. Forward propagation: compute hidden layer representation and predicted probabilities for each output word
2. Apply softmax at the output layer to get predicted probabilities ($\hat{y}$)
3. Compute loss between true context words (y) and predictions ($\hat{y}$)
4. Backward propagation: adjust weights to minimize loss
5. Repeat until convergence

22.5 Word Vectors

After training, each word is represented as a vector of dimension equal to the hidden layer size.

- Example: window size = 5 $\rightarrow$ each word represented by 5-dimensional vector
- Pre-trained models (e.g., Google Word2Vec) use huge corpora (3 billion words) with 300-dimensional vectors

22.6 CBOW vs Skip-Gram

- **CBOW:** Predicts center word from context, suitable for small datasets
- **Skip-Gram:** Predicts context from center word, suitable for large datasets

22.7 Improving Performance

- Increase training data
- Increase window size, which increases vector dimensions and captures more semantic relationships

22.8 Summary

Skip-Gram and CBOW are two complementary architectures in Word2Vec. Understanding fully connected neural networks, loss functions, and optimization is essential to implement them effectively.

23 Advantages of Word2Vec

23.1 Introduction

Word2Vec provides significant improvements over traditional text representation techniques like Bag of Words (BoW) and TF-IDF. Previous methods often resulted in sparse matrices with many zeros, which could lead to overfitting. Word2Vec, on the other hand, produces dense vector representations that capture semantic relationships between words.

23.2 Dense Representations

- Traditional techniques produce sparse matrices, making training machine learning models challenging.
- Word2Vec generates dense vectors, reducing the number of zeros and improving model training efficiency.

23.3 Semantic Information Capture

- Each word is represented as a vector in a continuous vector space.
- Similar words have similar vectors, enabling semantic relationships to be captured.
- Example: Words like ``honest'' and ``good'' will have vectors with high cosine similarity.

23.4 Fixed Vector Dimensions

- Word2Vec vectors have a fixed dimension, independent of vocabulary size.
- Example: Google's pre-trained Word2Vec model uses 300-dimensional vectors.
- This fixed-size representation avoids issues related to large vocabulary sizes encountered in BoW or TF-IDF.

23.5 Handling Out-of-Vocabulary Words

- Word2Vec can generalize to handle new words more effectively through semantic information capture.
- Each word is represented by a feature vector, reducing the risk of encountering completely unknown words.

23.6 Summary of Advantages

1. Dense matrix representations improve training efficiency.
2. Captures semantic relationships between words.
3. Fixed vector dimensions independent of vocabulary size.
4. Better handling of out-of-vocabulary words.
5. Generally outperforms traditional methods like BoW and TF-IDF in NLP tasks.

23.7 Next Steps

The next topic will introduce **Average Word2Vec**, which is particularly useful for solving text classification problems by aggregating word vectors to represent entire sentences or documents.

24 Practical Implementation of Word2Vec with Gensim

24.1 Introduction

In this chapter, we explore the practical implementation of Word2Vec using the Gensim library in Python. We will focus on Google's pre-trained Word2Vec model and see how it generates vector representations of words.

24.2 Setup

- Install Gensim:

```
pip install gensim
```

- Import required libraries:

```
import gensim
from gensim.models import Word2Vec, KeyedVectors
import gensim.downloader as api
```

24.3 Loading Google's Pre-trained Word2Vec Model

- Model: `word2vec-google-news-300`
- Trained on approximately 100 billion words from Google News.
- Contains 300-dimensional vectors for 3 million words and phrases.
- Load model using Gensim API:

```
wv = api.load("word2vec-google-news-300")
```

24.4 Exploring Word Vectors

- Retrieve the vector for a word:

```
vector = wv['king']
vector.shape # Output: (300,)
```

- All words are represented as 300-dimensional dense vectors.

24.5 Finding Similar Words

- Find most similar words:

```
wv.most_similar('cricket')
```

- Example output: cricketing, cricketer, test_cricket, etc.
- For `'happy'`:

```
wv.most_similar('happy')
```

Output: glad, pleased, ecstatic, overjoyed, thrilled, etc.

24.6 Measuring Similarity Between Words

- Compute cosine similarity between two words:

```
wv.similarity('hockey', 'sports') # e.g., 0.53
```

24.7 Vector Arithmetic

- Example: `king - man + woman = ?`

```
result_vector = wv['king'] - wv['man'] + wv['woman']
wv.most_similar([result_vector])
```

- Expected output: queen, monarch, princess, crown, etc.

24.8 Summary

- Pre-trained Word2Vec models provide dense 300-dimensional vectors for millions of words.
- They allow finding similar words, computing word similarity, and performing vector arithmetic.
- Gensim library provides an easy-to-use interface for loading pre-trained models and exploring word vectors.