04-ngram

HK Turesson

2024-09-23

Contents

Chapter 1

About

This book is based on lecture notes for the Natural Language Processing (NLP) course at the Master of Management in AI (MMAI) 5400 at the Schulich School of Business.

How the book was written:

1. Over the years, I built the course content as lecture slides.
2. I recorded myself giving the lecture.
3. The sound recording was transcribed using a speech to text (Whisper, open source?)
4. Regular expression was used to clean up and split the text into sections.
5. Each section was fed into an LLM with a prompt like `Re-write the following text in the style of a concise and not-too-technical introductory textbook in NLP. Format the output in latex.`
6. The returned texts were checked, corrected, figures were added and combined into lecture notes.
7. These lecture notes were provided to the students together with the slides as course materials.
8. The students were encouraged to edit, comment, ask for clarification, suggest additions etc. Students who contributed in this way were listed as chapter co-authors.
9. These lecture notes became the chapters making up this book.

1.1 Usage

Chapter 2

Natural Language Processing Tasks in Business

Written by Hjalmar K Turesson and Le Chat Mistral_ Large 2

2.1 Goal

The goal with this book is that it should prepare the students for the business world and allow them to do the following:

1. Identify the NLP task(s) that a particular business problem depends on.
2. Given the NLP task(s), identify their requisites.
3. Decide whether it's feasible or not.
4. If feasible, execute the tasks.
5. Given a business problem the students should be able to recognize the underlying NLP/computational task(s).
6. Given the task(s) they should be able to estimate what resources are needed to perform the task (e.g. data and compute).
7. Given the resources they should be able to evaluate the task(s) and tell whether it makes business sense to spend the resources.
8. Combine simple tasks into a composite task. For example, combining extracting aspect-opinion pairs, identifying aspect categories and finally sentiment analysis to perform aspect based sentiment analysis.

2.2 NLP Tasks

NLP can be thought of as a collection of tasks concerned with machine processing of natural language. Tasks are what you have to do to solve NLP-related business problems. It's often your job to recognize what task to apply given a business problem since problem might not mention the task. Thus, you need to know what the NLP tasks are and in what situations they apply.

The tasks are presented in **bold** and example methods or use cases below in *italics*.

- **Gather data**
  - Methods
    - *Web scraping* (example): Automatically download text from websites.
    - *Web APIs* (example): Download text for websites via APIs.
- **Text normalization**
  - Methods
    - *Case normalization* (example): For example, turn all characters in the text to lowercase.
    - *Tokenization* (example): Convert running text (a long string) into a sequence of tokens.
    - *Part-of-speech tagging* (example): Assign a grammatical category (such as noun, verb, adjective, etc.) to each word in a sentence.
    - *Stop word removal* (example): Remove uninformative words (i.e. stop words) such as articles (the, an) and conjunctions (and, but). Whether a word is informative or not depends on the downstream task(s).
    - *Word normalization* (example): Convert words to a standardized format or choosing a single, normal form for words with multiple variations. *Stemming* and *lemmatization* are word normalization methods.
  - Use cases
- **Language Modelling**
  - Methods
    - *N-gram language models* (example): Statistical models of the sequence of words in language computed from the frequencies of word co-occurrences.
    - *Neural language models* (example): Recurrent neural networks and Large language models.

    - * *Auto complete*: N-gram language models can be used to predict the next word or spelling in auto complete or spelling helpers.
  - Use cases
    - * *Text embedding* (use case): Neural language models can be used to encode, i.e. embed, text into a dense vector useful for downstream tasks.

- **Text Classification**
  - Methods
    - * *Sentiment Analysis* (example): Classify texts as expressing positive or negative sentimens.
  - Use cases
    - * *Brand Monitoring* (use case): Analyze social media posts and reviews to understand public sentiment towards a brand or product.
    - * *Spam Detection* (use case): Identify and filter out spam emails or messages to keep communication channels clean.
    - * *Incident ticket routing* (use case): Classify support tickets so that they can be automatically sent to the correct team.

- **Topic Modeling/Text Clustering**
  - Methods
    - * *Content Categorization* (example): Automatically categorize documents or articles into predefined topics for better organization and retrieval.
  - Use cases
    - * *Market Research* (use case): Analyze large volumes of text data to identify emerging trends and topics in customer feedback or social media.
    - * *Customer Segmentation* (use case): Group customers based on their preferences and behaviors to tailor marketing strategies.
    - * *Document Organization* (use case): Cluster similar documents together for better organization and retrieval.

- **Parsing**
  - Methods
    - * *Dependency parsing* (example): Describe the syntactic structure of a sentence in terms of the words (or lemmas) and an associated set of directed binary grammatical relations that hold among those words.
    - * *Part-of-speech tagging* (example): Assign a grammatical category (such as noun, verb, adjective, etc.) to each word in a sentence.

    - * *Semantic Analysis* (example): Understand the meaning of sentences by analyzing the dependencies between words, useful in advanced search engines and knowledge graphs.

  - Use cases
    - * *Information Extraction* (use case): Extract structured information from unstructured text, such as identifying relationships between entities in legal documents or contracts.

- **Named Entity Recognition (NER)**
  - Example methods
    - * *Customer Relationship Management*: Extract and organize information about customers, such as names, addresses, and contact details, from emails or forms.
    - * *Compliance Monitoring*: Identify and monitor mentions of regulated entities or sensitive information in communications to ensure compliance with regulations.
- **Summarization**
  - Methods
  - Use cases
    - * *Automated Reporting*: Generate concise summaries of lengthy documents, such as financial reports or meeting minutes, to save time and improve efficiency.
    - * *News Aggregation*: Provide users with summarized versions of news articles from various sources.
    - * *Executive Summaries*: Provide executives with concise summaries of lengthy reports or documents to facilitate decision-making.
- **Question Answering**
  - Methods
  - Use cases
    - * *Customer Support*: Develop chatbots that can answer customer queries based on a knowledge base of FAQs or product documentation.
    - * *Information Retrieval*: Enable users to ask natural language questions and receive accurate answers from large datasets, useful in legal research or medical diagnosis.
- **Machine Translation**
  - Methods
  - Use cases

    - * *Global Communication*: Translate documents, websites, and customer communications into multiple languages to reach a global audience.
    - * *Localization*: Adapt products and services to different languages and cultures to improve user experience and market penetration.

- **Text Generation**
  - Example methods
    - * *Content Creation*: Automatically generate blog posts, social media updates, or product descriptions to save time and resources.
    - * *Personalized Communication*: Create personalized emails or messages for customers based on their preferences and behavior.

- **Aspect-Based Sentiment Analysis** (composite)
  - Example methods
    - * *Product Improvement*: Identify specific aspects of a product that customers like or dislike to inform product development and improvement.
    - * *Service Enhancement*: Understand customer opinions on different aspects of a service to enhance service quality.
- **Text Similarity/comparison**
  - Methods
    - * *Information retrieval*: Find similar documents.
  - Use cases
    - * *Plagiarism Detection*: Identify similarities between documents to detect plagiarism in academic or professional writing.
    - * *Duplicate Content Detection*: Ensure that content is unique and not duplicated across different platforms or documents.
- **Text Annotation**
  - Example methods
    - * *Data Labeling*: Annotate text data with labels or tags to create training datasets for machine learning models.
    - * *Knowledge Graph Construction*: Annotate text with semantic information to build knowledge graphs for advanced search and analysis.
- **Text Segmentation**
  - Methods
    - * *Content Structuring*: Segment text into meaningful sections or paragraphs to improve readability and organization.

    * *Data Extraction*: Extract specific segments of text, such as headings, subheadings, or key points, for further analysis.

- Use cases

2.2.1 What do you need to know about a task?

- What type business problems can it be applied to?
- What are the inputs (i.e. what type of data)?
- What is the output?
- What other tasks are involved?
- How good is the state of the art?
- What model/algorithms are involved?
- What resources are required during deployment (estimate)?
- Compute, data privacy, etc.
- What resources are required for development (estimate)?
- Knowledge, time, compute, data, etc.

Given this knowledge you can acheive the goals set out above.

2.3 Why is NLP difficult?

NLP is challenging for a multitude of reasons, but first and foremost is the context dependence of natural language. Below is list of some of the challenges.

- **Context dependence**: The meaning of natural language is often context-dependent, making it crucial to consider the surrounding words, phrases, paragraphs and so on, when understanding a sentence.
    - *Bat*: Context-dependent. Can mean a nocturnal flying mammal, a piece of sports equipment, or the act of batting.
- **Ambiguity**: Natural language expressions can have multiple meanings, which can lead to confusion and errors in machine understanding. This phenomenon is called "*crash blossoms*."
    - "*I saw the man with the telescope*": Ambiguous. Could mean I used a telescope to see the man, or I saw a man who had a telescope.
- **An infinite number of possible sentences**: There are, in principle, an infinite number of possible sentences in natural language, making it impossible to list all sentence meaning pairs.

  - "*The cat sat on the mat*": Infinite sentences. The sentence can be extended to "*The cat sat on the big mat*," "*The big cat sat on the mat*," "*The big cat sat on the small mat*", "*The big cat sat on the small mat in the house*," and so on.

- **Non-standard language**: Natural language changes constantly and contains many non-standard forms, such as slang, contractions, and misspellings, which can make it difficult for machines to understand.
  - "*Lit*", "*On fleek*": Non-standard language. Slang terms can be difficult for machines to understand, especially if the terms are newer than the training data.

- **Ambiguous segmentation**: Segmenting words and phrases in natural language can be ambiguous, leading to different interpretations of the same sentence.
  - "**I saw her duck**": This could be segmented as "I saw her (and she) duck(ed)" meaning I saw a person who ducked. Alternatively, it could be segmented as "I saw her duck," meaning I saw the duck that belongs to her. The correct interpretation depends on the context.

- **Idioms**: Idiomatic expressions can be challenging to understand for machines, as their meanings are not always straightforward.
  - "*Kick the bucket*," "*Let the cat out of the bag*": Idioms. Meanings aren't directly related to their individual words.

- **Neologisms**: Newly coined words and expressions, such as "*unfriend*," "*retweet*," and "*bromance*," can be difficult for algorithms to understand.
  - "*Selfie*", "*Blog*", "*Vlog*": Neologisms. New words can be hard for machines to understand if they're not updated with these new terms.

- **Difficult entity names**: Some entity names can be tricky to identify, especially when they consist of multiple words or have similarities with other independently meaningful words. Examples include "**a bug's life**" (a movie title) and "**let it be**" (an album by the Beatles).
  - "*Apple's sales have increased*": Difficult entity names. Could refer to the fruit's sales or the tech company's sales, depending on context.

Chapter 3

Regular Expressions and Web Scraping

Written by Hjalmar K Turesson, ChatGPT, Darren Singh, Krystaleah Ramkissoon and Le Chat Mistral_ Large 2

3.1 Introduction

WRITE A BRIEF INTRO HERE!

3.1.1 Content

- **The data type, string**: We'll start by discussing the data type, which is string. Strings are sequences of characters and crucial to understand when working with text processing.
- **Regular Expressions (RegEx)**: Next, we'll explore regular expressions, a powerful tool for manipulating and matching strings. They allow you to define patterns to efficiently find and extract specific information from text.
- **Web Scraping**: Moving on, we'll delve into web scraping, a common method for extracting data from websites. Web scraping is essential for gathering information from the web, which can be useful for various purposes.

Thus, well cover the fundamentals of working with strings, manipulating them using regular expressions and extracting data from the web through web scraping.

3.2 Strings

The data type string is a sequence of characters with some character encoding like ASCII or UTF8. ASCII, the American Standard Code for Information Interchange is a character encoding where each character is encoded by 7 bits, resulting in $2^7 = 128$ unique characters. This covers the English alphabet (upper and lower case), the digits 0 to 9 and a few other symbols such as the punctuation symbols (`.`, `?`, `!`). However, 128 characters does not cover characters in many non-English languages. To address this limitation, Unicode was introduced, which associates characters with unique code points. UTF8 is a variable-width (i.e. not fixed at a certain number of bits) encoding of Unicode, using one to four bytes to encode all 1,112,064 Unicode characters. This flexibility allows encoding a broader range of characters, making it more suitable for various languages.

3.2.1 Strings in Python

In Python, the string data type is called `str`, and it holds Unicode strings. Strings in Python can be delimited using single (e.g.`'a string'`), double (e.g.`"another string"`), or triple (e.g.`'''yet another string'''`) quotes. Different delimiters can be nested (e.g.`'''"'a string' inside a string" inside another string'''`), allowing flexibility in string construction (copy and paste the examples into Ipython or a Jupyter notebook to test).

3.2.1.1 Triple quoted strings

In Python, strings can be delimited by triple single (`'''`) or triple double (`"""`) quotation marks. These are called triple quoted strings. Within triple quoted strings, both single and double quotes can be included without the need for escaping (see Escape sequences below). These strings are especially useful for creating multi-line strings, such as help texts in Python files, where you can include both types of quotes without any issues.

For example:

```
def partition_data(data_splits, data, kind='drug'):
    """
    data_splits : data_splits should sum to 1
    data        : The data to split
    kind        : Optional. "pair" or "drugs".
                    "pair" (splits on pairs, DeepDTA-style) or
                    "drugs" (splits on the unique drugs)

    Assume that drugs are novel and searched for while proteins are known
```

```
    partition data on the drugs so that drugs in train are not in valid or
    test, and drugs in valid are not in test.
    """
    ...
```

3.2.1.2 Escape sequences

Escape sequences in Python strings involve using the backslash (\) character as the *escape character*. The backslash indicates the beginning of an escape sequence, which allows certain characters to *escape* their normal meanings (how they are interpreted by Python). For instance, a double quotation mark inside a double quotation-delimited string can be escaped with a backslash, allowing it to be treated as a literal character and not as a delimiter.

Examples Try pasting the following into Ipython or Jupyter:

```
print(""a string" inside another string")
```

Does it work? If not, why?.

Try also this:

```
print("\"a string\" inside another string")
```

Does this work? If not, why?

Another example of an escape sequence is `'\n'` where the character `'n'` escapes it's normal meaning and instead is interpreted as a newline character.

To see what effect `'\n'` has try:

```
print("Hello\nworld!")
```

The escape character can be printed by using it upon itself (i.e. `print('\\')` will print \).

3.2.1.3 Raw strings

In Python, raw strings are denoted by a lowercase `'r'` or uppercase `'R'` prefix. In raw strings, escape sequences are not translated. This is particularly useful when working with regular expressions because backslashes have special meanings in regular expressions (see Regular expressions below). By using raw strings, you can avoid the need to escape escape characters, thus simplifying the use of regular expressions.

Example Try:

```
print(r"Hello\nworld!")
```

3.2.1.4 Operating on strings

Python provides multiple functions and methods to operate on strings, among them `len("a string")` that returns the length of the string (8), `"a string".capitalize()` which returns a capitalized string (`'A string'`), and `str.replace("old", "new')` which replaces the sub-string `"old"` with the string `"new"` (e.g. `"a string".replace("string", "ping")` will return `'a ping'`).

For a complete list of string operations, type `help(str)` at a Python prompt (e.g. IPython or Jupyter).

3.3 Regular Expressions

A *regular expression* (also regex or regexp) is a powerful tool for extracting information from strings. A regex is a sequence of characters that specifies a match pattern in a text (i.e. a string). Usually such patterns are used by string-searching algorithms for "find" or "find and replace" operations on strings, or for input validation. For instance, you can use regex to extract prices of items from a given text automatically. Regex is supported by many programming languages and platforms, including SQL, Python, R and JavaScript. Text editors and integrated development environments (IDEs) often support regex searching. Regex is a crucial tool in the data scientists toolbox.

3.3.1 What are Regular Expressions?

Regex is crucial tool in data science and is an algebraic notation for characterizing sets of strings.

3.3.2 For what are they used?

Regular expressions have multiple uses in in programming and data manipulation. Here are some key uses with explanations:

- **Search and Replace**:
  - *Explanation*: Regex can search for specific patterns within text and replace them with other strings.

  - *Example*: Replacing all occurrences of a word in a document.

- **Data Extraction**:
  - *Explanation*: Regex can extract specific parts of a string based on a pattern, for example numbers, dates, or other specific patterns from text. This makes regex useful for data extraction and analysis from various sources like code, log files, and spreadsheets.
  - *Example*: Extracting all phone numbers from a block of text.

- **Input Validation in Forms**:
  - *Explanation*: In web development, regex is used to validate user input in forms before submission.
  - *Example*: Ensuring a username contains only alphanumeric characters.

- **String Parsing**:
  - *Explanation**: Regex can be used to parse and break down complex strings into more manageable parts.
  - *Example**: Parsing a log file to extract timestamps, IP addresses, and error messages.

- **Syntax Highlighting**:
  - *Explanation*: In text editors and IDEs, regex is used to identify and highlight different parts of code syntax.
  - *Example*: Highlighting keywords, comments, and string literals in a programming language.

- **Web Scraping**:
  - *Explanation*: Regex can be used to extract information from HTML or XML documents downloaded from webpages, thus helping to gather information efficiently.
  - *Example*: Extracting all URLs from a webpage.

- **Data Cleaning and Wrangling**:
  - *Explanation*: Regex can be used to clean and standardize data by removing unwanted characters or formatting inconsistencies.
  - *Example*: Removing all non-alphanumeric characters from a string.

- **arsing and Tokenization**:
  - *Explanation*: Regex can be used to process and manipulate large amounts of text data and can be used to break down text into meaningful units or tokens, which is essential for NLP and parsing tasks.
  - *Example*: Splitting a large text file into individual sentences or words (see Tokenization).

- **Security**:

  - *Explanation*: Regex can validate user inputs such as emails, usernames, and passwords, ensuring they meet specific criteria or patterns and it can be used to detect and filter out potentially harmful input, such as SQL injection attempts.
  - *Example*: Validating user input to prevent injection attacks.

- **Configuration Files**:
  - *Explanation*: Regex can be used to parse and manipulate configuration files.
  - *Example*: Updating values in a configuration file based on specific patterns.

- **Code Refactoring**:
  - *Explanation*: Regex can be used to find and replace patterns in code, making it easier to refactor large codebases.
  - *Example*: Renaming a variable across multiple files.

These uses demonstrate the versatility and power of regular expressions in various domains, from data processing and validation to security and text analysis.

3.3.3 How to get started with regular expressions

For basic usage of regular expressions, you need to grasp the fundamental concepts and syntax. Here are the key requirements to get started:

- **Understanding the Basics of Regex Syntax**:
  - *Literals*: Characters that match themselves exactly, e.g., `a`, `b`, `1`, `2`.
  - *Metacharacters*: Special characters that have a specific meaning in regex, such as:
    `.` (dot): Matches any single character except newline (`'\n'`).
    `^` (caret): Matches the start of a string.
    `$` (dollar sign): Matches the end of a string.
    `*` (asterisk): Matches 0 or more of the preceding element.
    `+` (plus): Matches 1 or more of the preceding element.
    `?` (question mark): Matches 0 or 1 of the preceding element.
    `[]` (square brackets): Defines a character class, e.g., `[abc]` matches 'a', 'b', or 'c'.
    `|` (pipe): Acts as a logical OR, e.g., `a|b` matches 'a' or 'b'.
    `()` (parentheses): Groups subpatterns and creates capturing groups.

- **Character Classes**:
  - *Predefined Character Classes*: Familiarize yourself with common character classes like:
    `\d`: Matches any digit, equivalent to `[0-9]`.

`\w`: Matches any word character, equivalent to `[a-zA-Z0-9_]`.
`\s`: Matches any whitespace character, equivalent to `[ \t\r\n\f\v]`.

- *Negated Character Classes*: Understand how to negate character classes, e.g.:
  `\D`: Matches any non-digit character. `\W`: Matches any character that's not a letter, digit, or underscore. `\S`: Matches any non-whitespace character.

- **Quantifiers**:
  - *Basic Quantifiers*: Learn how to use quantifiers to specify the number of occurrences:
    - `*`: 0 or more.
    - `+`: 1 or more.
    - `?`: 0 or 1.
    - `{}`: Exact number or range, e.g., `{3}` (exactly 3), `{3,}` (3 or more), `{3,5}` (between 3 and 5).

- **Anchors**:
  - *Start and End Anchors*: Understand how to use `^` and `$` to match the start and end of a string, respectively.

- **Escaping Special Characters**:
  - *Escape Sequences*: Learn how to escape special characters using the backslash \, e.g., `\.` to match a literal dot.

- **Basic Usage in a Programming Language**:
  - *Regex Functions*: Familiarize yourself with the basic regex functions in your preferred programming language, such as:
    `match()`: Checks if a string matches a pattern. `search()`: Searches for a pattern within a string. `findall()`: Finds all occurrences of a pattern in a string. `sub()`: Replaces occurrences of a pattern with a replacement string.

- **Online Regex Testers**:
  - *Regex Tools*: Utilize online regex testers like Regex101, Regexr, or Debuggex to practice writing patterns, test their behavior, and understand the matching process.

- **Practice with Simple Examples**:
  - *Basic Patterns*: Start with simple patterns and gradually move to more complex ones, e.g.:
    - Matching a specific word: `cat`.
    - Matching a digit: `\d`.
    - Matching a word character: `\w`.
    - Matching an email pattern: `\w+@\w+\.\w+`.

3.3.4 Examples

Here are a few examples of regex usage for Python. These examples demonstrate the basic usage of regular expressions, starting with simple patterns and gradually increasing in complexity. By practicing with these examples, you can build a solid foundation in regex and move on to more advanced patterns as you become more comfortable.

Matching a Specific Word
Objective: Match the word "cat" in a string.

Regex Pattern: `cat`

Explanation: This pattern simply matches the exact sequence of characters "cat".

Example Usage:

```
import re

text = "The cat is sitting on the mat."
pattern = r"cat"

matches = re.findall(pattern, text)
print(matches)  # Output: ['cat']
```

Matching a Digit

Objective: Match any digit in a string.

Regex Pattern: `\d`

Explanation: The `\d` pattern matches any single digit (0-9).

Example Usage:

```
import re

text = "The price is $123.45."
pattern = r"\d"

matches = re.findall(pattern, text)
print(matches)  # Output: ['1', '2', '3', '4', '5']
```

Matching an Email Address

Objective: Match a simple email address pattern.

Regex Pattern: `\w+@\w+\.\w+`

Explanation:
`\w+`: Matches one or more word characters (letters, digits, and underscores).
`@`: Matches the "@" symbol.
`\w+`: Matches one or more word characters.
`\.`: Matches a literal dot.
`\w+`: Matches one or more word characters.

Example Usage:

```
import re

text = "Contact us at support@example.com or info@domain.org."
pattern = r"\w+@\w+\.\w+"

matches = re.findall(pattern, text)
print(matches)  # Output: ['support@example.com', 'info@domain.org']
```

Matching a Date in the Format MM/DD/YYYY

Objective: Match a date in the format MM/DD/YYYY.

Regex Pattern: `\d{2}/\d{2}/\d{4}`

Explanation:
`\d{2}`: Matches exactly two digits.
`/`: Matches the "/" symbol.
`\d{2}`: Matches exactly two digits.
`/`: Matches the "/" symbol.
`\d{4}`: Matches exactly four digits.

Example Usage:

```
import re

text = "The event is scheduled for 12/31/2023 and 01/01/2024."
pattern = r"\d{2}/\d{2}/\d{4}"

matches = re.findall(pattern, text)
print(matches)  # Output: ['12/31/2023', '01/01/2024']
```

3.4 Web Scraping

Web scraping, also referred to as *screen scraping*, *web data extraction*, or *web harvesting*, is a technique employed to retrieve data from web pages. While data can be retrieved manually, automating the process through web scraping enhances speed, efficiency, and accuracy. To navigate the world of web scraping,

it is crucial to gain insight into how web pages are structured and presented (see Understanding Web Page Structure).

3.4.1 What is Web Scraping?

Web scraping is the systematic process of extracting data from websites or web pages. It involves fetching and parsing the HTML or other markup languages of web pages to gather specific information, such as text, images, links, or structured data.

Web scraping automation employs software tools and scripts to perform data extraction tasks swiftly and efficiently. This automated approach significantly reduces the time and effort required compared to manual data retrieval methods.

3.4.2 For what is Web Scraping used?

Web scraping have multiple uses, including:

- **Data Collection**: Gathering data for research, analysis, or business insights.
- **Market Research**: Monitoring competitors, pricing, and trends.
- **Content Aggregation**: Collecting news articles, reviews, or product listings.
- **Lead Generation**: Extracting contact information from websites.
- **Automated Testing**: Testing website functionality and performance.
- **Price Comparison**: Tracking prices of products or services across multiple websites.

3.4.3 Understanding Web Page Structure

To excel in web scraping, one should possess a fundamental understanding of how web pages are structured and presented. Knowledge of HTML, CSS, and the Document Object Model (DOM) is essential. These technologies govern how web content is organized and displayed in browsers, and web scrapers interact with them to extract data. DOM is a standardized programming interface for the structuring of HTML and XML documents. The purpose of this is to find a way to logically represent documents within computers. It depicts the structure of the document in a tree structure in which every node is an independent,

selectable object. Because of this structure, one also refers to this type of displayed web project version as a DOM tree.

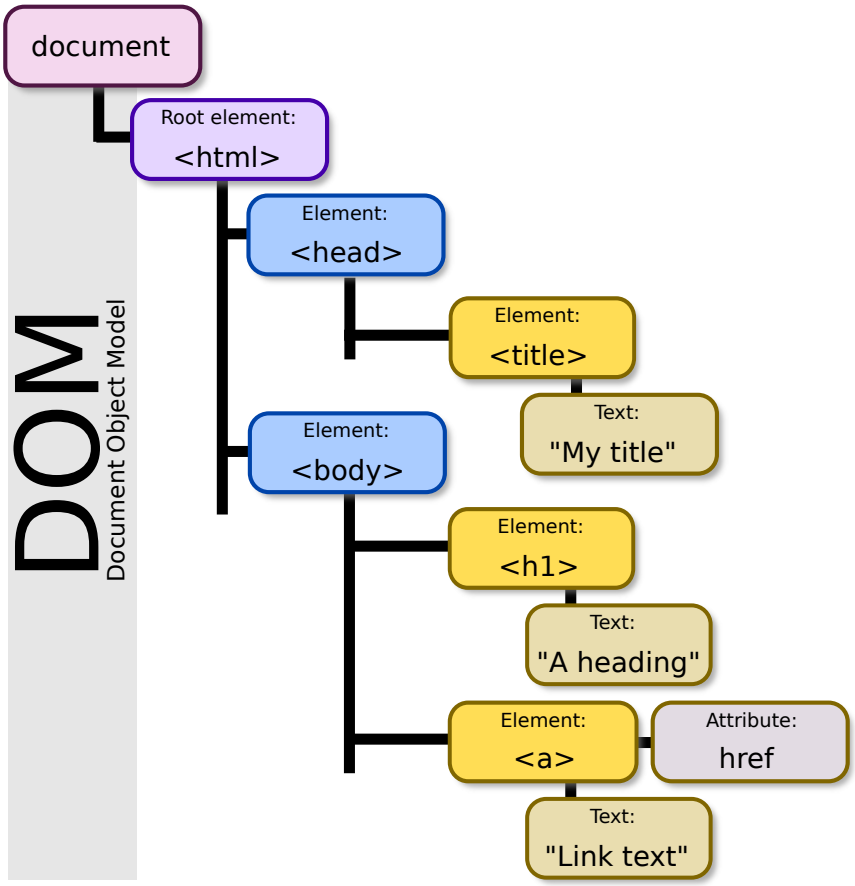

Figure 3.1: Figure 1. Birger Eriksson, CC BY-SA 3.0 https://creativecommons.org/licenses/by-sa/3.0, via Wikimedia Commons.

3.4.3.1 XML and HTML documents

XML and HTML documents are text files written in markup languages. "XML" stands for Extensible Markup Language, while "HTML" stands for Hypertext Markup Language. A markup language is used to annotate a document in a way that visually distinguishes formatting and structure from the actual text content. Examples of markup languages include LaTeX, Scribe, XML, HTML, and Markdown - the markup language this book is written in. XML and HTML are quite similar, with HTML 5 evolving to become a dialect of XML. Both markup languages play a crucial role in structuring and formatting web content.

To view the source code (e.g. the raw HTML code) of a web page, in most browsers, you can right-click on it and select 'View page source.'

3.4.3.1.1 Basic XML document structure XML documents adhere to a structured format where *elements* (indicated by _element *nodes*), *attributes* (indicated by *attribute nodes*), and *text content* (indicated by *text nodes*) are organized hierarchically, forming a tree-like structure with the document as the *root node*. Properly nested elements and adherence to case sensitivity are essential principles in XML document structure.

- **Root Node**: The XML document is organized as a tree, with the document itself serving as the root of the tree.
- **Element Nodes**: Elements are the building blocks of an XML document and must have both opening and closing tags. The opening tag is enclosed in angle brackets `'<>'`, and the closing tag is represented as an angle bracket, forward slash, and closing angle bracket `'</>'`. Tags are case-sensitive.
- **Proper Nesting**: Elements can be nested within other elements, creating a hierarchical structure. Proper nesting is crucial for maintaining the document's integrity.
- **Attribute Nodes**: Elements may contain attributes which provide additional information. Attributes must have values associated with them.
- **Text Nodes**: Text nodes refer to the content enclosed within the opening and closing tags of elements. They represent the actual data or text contained within the document.

3.4.3.1.2 Basic HTML Tags This is a list of a small subset of HTML tags. Typically, the text for the title/headline goes between the opening and closing tags.

`<h1></h1>`: A headline or header tag used to create titles or headlines in documents. For example `<h1>This is the title!</h1>`.

`<p></p>`: A paragraph tag used to create a paragraph within a document, the paragraph text appears between the opening and closing tag. For example `<p>This the the beginning of a long paragraph ... and this is the end.<p>`.

`<table></table>`: A table tag used to create a table within a document. For example `<table><tr><td>Hjalmar</td> <td>Kosmos</td><td>Nat</td></tr></table>` (see below for the meanings of `<tr></tr>` and `<td></td>`).

`<tr></tr>`: A table row tag used to create a row within a table within a document. See example under `<table></table>`.

`<td></td>`: A table cell tag used to fill in a cell within a table within a document. See example under `<table></table>`.

3.4.3.1.3 Learn more You can learn and explore more about HTML tags on W3Schools HTML Tutorial.

3.4.4 Web scraping with Python

3.4.4.1 Tools

- **Beautiful Soup**:
  - Beautiful Soup is used for extracting data from web pages that have already been downloaded.
  - To use Beautiful Soup, you first download a web page using Python libraries like `requests` or `urllib3`.
  - It's ideal for working with static content and is a good starting point for beginners to understand web scraping.
  - For a tutorial, see Beautiful Soup: Build a Webscraper With Python.
- **Scrapy**:
  - Scrapy is a comprehensive web scraping library for Python.
  - It handles the entire web scraping process, including downloading, cleaning, and saving data.
  - Scrapy is robust, highly customizable, and suitable for complex web scraping tasks.
  - There are several helpful tutorials available for learning Scrapy, for example the Scrapy Tutorial included in the documentation.
- **Selenium**:
  - Selenium is designed for scraping dynamic web pages where content changes with user interaction (technically, it's constructed at runtime).
  - Selenium simulates a web browser, allowing you to interact with web elements like buttons and links.
  - Selenium essentially acts as a headless web browser, making it versatile for dynamic content scraping.
  - It is useful (necessary) for scraping content that requires actions like scrolling or clicking.
  - Selenium is more difficult to install and use than Beautiful Soup and Scrapy.

In summary, Scrapy provides an end-to-end solution for web scraping, including downloading, cleaning, and saving data. Beautiful Soup is used for static content extraction from downloaded web pages, while Selenium is essential for scraping

dynamic content where user interactions are needed. These tools cater to various web scraping scenarios, making Python a powerful choice for web data extraction tasks.

3.4.5 Legal and Ethical Considerations

While web scraping offers powerful capabilities, it is important to respect legal and ethical boundaries. Some websites may have terms of service that prohibit scraping, and scraping sensitive or personal data without consent may raise ethical concerns.

3.4.5.1 Global Variations in Legality

The legality of web scraping differs from one jurisdiction to another. Generally, web scraping may violate the terms of service of certain websites. However, the enforceability of these terms remains ambiguous.

3.4.5.2 Basic Principles

In general, the act of automatically downloading a web page and extracting information from it is not illegal. This process is akin to viewing a web page through a browser, which is a legal activity.

3.4.5.3 Copyright and Commercial Use

Legal issues can arise if the scraped data is protected by copyright and used for commercial purposes. For instance, Mongohouse.com scraped listings from the Toronto Real Estate Board (TREB), republished them on its own site, and generated revenue through real estate-related advertisements, effectively competing with TREB. This activity was deemed illegal.

3.4.5.4 Best Practices (Not Legal Advice)

- **Read Terms and Conditions**: Always review the terms and conditions of the websites you intend to scrape.
- **Seek Clarification**: If you are unsure about the legality, seek permission or clarification.
- **Avoid Overloading Servers**: Be cautious with frequent requests to small websites to avoid causing a Denial of Service (DoS) attack.
- **Scrape During Off-Peak Hours**: Conduct scraping activities during periods of low traffic to minimize the impact on the website's performance.

- **Pause Between Requests**: Implement pauses between requests, for example, using Python's `time.sleep(num_seconds)` function.

These guidelines are intended to provide general advice and do not constitute legal advice. Always consult with a legal professional for specific guidance.

3.4.6 An alternative to Web Scraping: Web APIs

Instead of web scraping, another method for obtaining structured data from websites is to utilize Web APIs (Application Programming Interfaces). Web APIs provide a structured and often more reliable way to access data from websites compared to web scraping. Many websites offer APIs that allow developers to access and retrieve data in a structured and organized manner. Examples of websites with APIs include Facebook, YouTube, cryptocurrency exchanges (e.g. Coinsquare), PubMed, and many more. It's important to refer to API documentation and use the appropriate tools to interact with the API effectively.

3.4.6.1 How to start using web APIs

1. **Documentation**: Start by searching for documentation related to the API you are interested in. You can usually find this by searching for the website name followed by "API" on popular search engines.

2. **API Requests**: Use libraries like `requests` in Python to make HTTP requests to the API. These requests can include parameters to specify the data you want to retrieve.

3. **API-Specific Packages**: In some cases, you may find specialized Python packages or libraries designed for interacting with specific APIs. These packages can simplify the process of working with the API.

3.4.6.2 Example: Getting Data from a Public Weather API

Below is a basic example of how to get data from a web API using Python's `requests` library to make HTTP requests and the `json` library to handle the JSON response. You can adapt this script to interact with other APIs by changing the URL and handling the response accordingly.

We'll use the OpenWeatherMap API to get the current weather data for a specific city. For the example to work you will need to get an API key from OpenWeather and replace `'your_api_key_here'` with it.

```
import requests

# Define the API endpoint and parameters
api_key = 'your_api_key_here'  # Replace with your actual API key
city = 'Toronto'
url = f'http://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}'

# Make the GET request to the API
response = requests.get(url)

# Check if the request was successful
if response.status_code == 200:
    # Parse the JSON response
    data = response.json()

    # Extract and print relevant information
    weather_description = data['weather'][0]['description']
    # The temperature in Kelvin, convert to Celsius using `temperature - 273.15`.
    temperature = data['main']['temp']

    humidity = data['main']['humidity']

    print(f'Weather in {city}: {weather_description}')
    print(f'Temperature: {temperature} K')
    print(f'Humidity: {humidity}%')
else:
    print(f'Error: {response.status_code}')
```

3.4.6.2.1 Explanation

1. **Import the `requests` library**: This library allows you to send HTTP requests.
2. **Define the API endpoint and parameters**:
   - `api_key`: Your API key from OpenWeatherMap.
   - `city`: The city for which you want to get the weather data.
   - `url`: The API endpoint URL with the city and API key as query parameters.
3. **Make the GET request**: Use `requests.get(url)` to send a GET request to the API.
4. **Check the response status**: Ensure the request was successful (status code 200).
5. **Parse the JSON response**: Use `response.json()` to parse the JSON data returned by the API.

6. **Extract and print relevant information**: Access specific fields in the JSON data to get the weather description, temperature, and humidity.

3.5 Abbreviations

- NLP - Natural Language Processing
- ML - Machine Learning
- RegEx - Regular Expressions
- ASCII - American Standard Code for Information Interchange
- UTF8 - Unicode Transformation Format – 8-bit
- IDE - Integrated Development Environment
- HTML - HyperText Markup Language
- CSS - Cascading Style Sheets
- DOM - Document Object Model
- XML - eXtensible Markup language
- DoS attack - Denial-of-service attack
- API - Application Programming Interface
- NER - Named entity recognition
- BPE - Byte-Pair Encoding
- BERT - Bidirectional Encoder Representations from Transformers
- GPT - Generative Pretrained Transformer

Terminology

3.5.0.1 Strings and regular expressions

String: A string is a sequence of characters which can represent text or data. In computing, strings are a fundamental data type used for various purposes, including text processing and data manipulation.

Character encoding: Character encoding refers to the method used to represent characters in a computer system. Common character coding schemes include ASCII and UTF-8, which map characters to numeric values.

Unicode: Unicode is a character coding standard that specifies a unique numeric code point for each character, symbol, or glyph from various writing systems worldwide. It allows for the representation of a wide range of characters, including those from non-English languages.

Escape Sequence: An escape sequence is a combination of characters, typically starting with an escape character (often a backslash '\'), that represents a special character or control code within a string. Escape sequences are used to include characters that might have special meanings, such as escaping quotes within a string.

Regular Expression: A regular expression, often abbreviated as "regex," is a powerful pattern-matching language used to define search patterns within strings. It enables precise and flexible text matching and manipulation by specifying rules for finding and extracting specific patterns or substrings within larger texts.

Web scraping

Web Scraping: Web scraping is the process of programmatically extracting data from websites. It enables the retrieval of specific information from web pages, automating data collection for various purposes.

Screen Scraping: Screen scraping is a synonym for web scraping and refers to the practice of extracting data displayed on computer screens. It involves capturing and parsing visual data from websites or applications.

Web Data Extraction: Web data extraction, sometimes referred to as web harvesting, is the broader process of collecting data from the web, including web scraping. It encompasses techniques for systematically gathering information from websites.

Markup Language: A markup language, such as XML (Extensible Markup Language) or HTML (Hypertext Markup Language), is a system for annotating documents to distinguish text content from structural elements and formatting.

XML and HTML: XML and HTML are markup languages commonly used for structuring and formatting web content. XML is known for its extensibility, while HTML is designed for creating web pages with hypertext links.

XML Tree: An XML document is organized hierarchically, forming an XML tree. This tree structure consists of elements, attributes, and text nodes, creating a structured representation of data.

Element Attributes: Elements in XML or HTML can contain attributes that provide additional information about the element. Attributes are typically name-value pairs and enhance the element's properties. These could be names, spans, classes, id's, etc.

Text Nodes: Text nodes represent the actual content enclosed within the opening and closing tags of elements in XML or HTML. They contain the textual data within the document.

HTML Tags: HTML documents use tags, enclosed in angle brackets, to define elements and their attributes. Tags play a crucial role in structuring and formatting web content.

Web API: Web APIs are interfaces provided by websites or web services that allow developers to access and retrieve structured data in a programmatic manner. They offer an alternative to web scraping for data extraction.

Chapter 4

Text Normalization

Written by Hjalmar K Turesson, ChatGPT, Darren Singh and Krystaleah Ramkissoon

4.1 Introduction

The previous chapter introduced the data type holding texts – string (see Strings), and a method to gather texts for NLP tasks – web scraping (see Web scraping). However, a raw text – a sequence of characters stored in a string – is not a suitable input to downstream ML or other NLP tasks. First we need to pre-process the text. Text preprocessing, aka *text normalization*, involves splitting the strings into words (see Tokenization), removing punctuation, converting text to lowercase (*case normalization*), and handling special characters to make the data consistent and suitable for analysis. What further steps to take depends on the type of downstream NLP tasks, but stop word removal (see Stop Words) and word normalization (see Word Normalization) are common, especially when the subsequent NLP tasks are based on the bag-of-words (BoW) representation (see Bag-of-Words). Text normalization often trades information in return for improved generalization. This is especially true for word normalization and stop word removal.

4.1.1 Steps of text normalization

1. Tokenizing
2. Normalizing word formats
    - All words to lowercase
    - Plurals to singulars
    - Removing stop words

   - Stemming
   - Lemmatizing
3. Segmenting (parsing) sentences
   - Dependency parsing

4.1.2 Content

- **Tokenization**: We'll start by discussing different ways for splitting continuous text (a sequence of characters) into individual tokens (e.g. words).
- **Word Normalization** is the process of converting words to a canonical form, such as their base form or lemma. This task is crucial for NLP applications like information retrieval and text mining, as it helps to reduce the vocabulary size and improve the accuracy of matching and retrieval.
- **Stop words removal** involves removing common words (like "the," "and," "a") from a text document to improve the efficiency and accuracy of subsequent NLP tasks.

4.2 Tokenization

Tokenization is a fundamental process in NLP that involves breaking down the continuous running text (i.e. a string) into discrete units or tokens. These tokens serve as the building blocks for various downstream NLP tasks, such as dependency parsing and machine learning. This chapter explores the concept of tokenization and its different levels. In the context of NLP, a piece of text is often represented as a continuous string, a sequence of characters. This string comprises words, punctuation marks, spaces, and other elements that together form the text's structure. The primary objective of tokenization is to transform this continuous string into a structured list of tokens, typically representing words. This structured format is more amenable to analysis and facilitates subsequent NLP tasks. By default, Python treats each character as a token and allows for strings to be indexed similarly to arrays in which each element is a character.

4.2.1 Three Levels of Tokenization

Tokenization can be applied at three distinct levels:

- **Word Tokenization**: At the word level, tokenization breaks the text into individual words or word-like units. For instance, the sentence "Despite fair coding" would be tokenized into tokens like ["Despite," "fair,"

"coding"]. Word tokenization is commonly used for many NLP tasks as it provides meaningful units for analysis.

- **Subword Tokenization**: Subword tokenization takes a more granular approach, breaking text into smaller units that may not necessarily be complete words. An example of subword tokenization might involve splitting "unhappiness" into subword tokens like ["un", "h", "appiness"]. Subword tokenization is valuable in morphologically rich languages and scenarios where words can be deconstructed into smaller, meaningful parts.
- **Character Level Tokenization**: At the character level, each character in the text is treated as an individual token. For example, the word "tokenization" would be tokenized into ["t", "o", "k", "e", "n", "i", "z", "a", "t", "i", "o", "n"]. Character level tokenization is useful in specific cases, such as analyzing character-level patterns in text.

In summary, tokenization is a critical step in NLP, enabling the transformation of text into structured tokens that are essential for various analytical and processing tasks. The choice of tokenization level depends on the specific requirements of the NLP application and the linguistic characteristics of the text being analyzed.

4.2.1.1 Word Tokenization

Word tokenization is a crucial step in NLP that involves segmenting text into individual words or tokens. These tokens serve as the foundation for various downstream NLP tasks. However, word tokenization is not without its challenges, including issues related to punctuation, spaces, and efficiency.

Handling Punctuation Punctuation marks, such as periods and commas, pose a challenge in word tokenization. While periods often indicate the end of a sentence, they can also signify a decimal point in numerical values like "3.1416." Tokenizing periods in the first instance is logical, but it can lead to undesirable outcomes in the second, where "3.1416" should remain a single token, not broken down into "3", "." and "1416". Determining the appropriate treatment of punctuation depends on the specific NLP task at hand.

Space as a Word Boundary Whitespace, typically spaces, often serves as an indicator of word boundaries. However, this rule is not universal. Consider the phrases "New York University Library" and "new York University Library." In the first sentence, "New", "York" and "University" should all be combined into a single token, recognizing the university New York University. In the second sentence, "new", "York" and "University" should be two separate tokens, indicating a university named York University. Named entity recognition (NER) can be a valuable tool to address this challenge and aid in tokenization by identifying entities within the text.

Speed and Efficiency Efficiency is paramount in tokenization since it is a preprocessing step. Delays in tokenization can significantly impact the overall processing pipeline's speed. To ensure swift tokenization, many implementations rely on regular expressions, which offer a balance between accuracy and efficiency.

Conclusion Word tokenization is essential in NLP, but it comes with nuances related to punctuation, spaces, and efficiency. Handling punctuation and whitespace effectively requires consideration of the specific NLP task and the use of techniques like NER. Moreover, using efficient implementations, often based on regular expressions, helps maintain a streamlined processing pipeline. Ultimately, careful word tokenization is crucial for accurate and efficient NLP analysis.

4.2.1.2 Character-Level Tokenization

Character-level tokenization is an unconventional but noteworthy approach to segmenting text. While it may not be as prevalent as word-level tokenization, it offers distinct advantages, especially in addressing the challenge of handling unknown or rare words.

The challenge of unknown words In traditional word-level tokenization, words are the primary units. However, this approach can encounter issues when dealing with words that occur very infrequently or even just once in a dataset. These are called out-of-vocabulary (OOV) words. OOVs pose challenges for ML models, particularly when they appear in a test set but were not encountered during training.

Character-level tokenization as a solution Character-level tokenization offers an alternative solution to the problem with OOVs. In character-level tokenization, each individual character within a text becomes a token. For example, the sentence "my name is" would be broken down into tokens like ["m", "y", " ","n", "a", "m", "e", " ", "i", "s"].

When character-level tokenization makes sense Character-level tokenization may not be suitable for all NLP tasks, especially those involving bag-of-words models. However, it finds its strength in sequence models, which are designed to work with sequences of characters. This approach becomes particularly valuable when dealing with situations where word boundaries are not clear or when addressing the challenge of rare or unknown words.

Conclusion In summary, while character-level tokenization may not be as common as word-level tokenization, it presents an innovative solution to handling unknown or rare words in NLP tasks. This approach's effectiveness depends on the specific task at hand, and it plays a significant role in sequence models, which we will delve into further in later discussions.

4.2.1.3 Subword tokenization: bridging the gap between words and characters

Subword tokenization is an innovative approach aimed at addressing the challenge of handling unknown or rare words in NLP. While defining tokens as words can lead to numerous unknown words, subword tokenization seeks to find the largest meaningful segments within the text while minimizing the impact of rare or unfamiliar tokens.

The challenge of unknown words Tokenizing text based on full words can pose difficulties when dealing with words that are infrequent or entirely unknown in a given dataset. These words can disrupt the effectiveness of NLP models, particularly during testing when previously unseen words appear.

The Goal of Subword Tokenization Subword tokenization aims to find meaningful chunks of text that minimize the occurrence of unknown or rare tokens. By doing so, it reduces the complexity of token combinations and facilitates effective NLP analysis.

Chunking Text In subword tokenization, the text is divided into chunks that can include limited-word tokens (complete words), word combinations (e.g., "York University"), and parts of words (e.g., "un", "st" and "ing"). The key is to create meaningful units that help maintain the text's linguistic and semantic integrity. Large chunks/tokens are important to decrease the number of combinations of tokens (limit combinatorial explosion).

Common Algorithms Two common algorithms for subword tokenization are "*Byte Pair Encoding*" (BPE) and "*WordPiece*." These algorithms are further described in the book "Speech and Language Processing." BPE is notably used for models in the GPT series of LLMs, featuring a vocabulary size of around 50,000 tokens. On the other hand, WordPiece is employed in the BERT family of models and typically has a vocabulary size of approximately 30,000 tokens.

Subword tokenization bridges the gap between full words and individual characters, offering a solution to the challenge of handling rare or unknown words in NLP tasks. By segmenting text into meaningful subword tokens it simplifies token combinations and enhances the robustness of NLP models. This approach

is especially valuable in scenarios where the vocabulary is diverse and includes a wide range of terms.

The BPE Algorithm: A Vocabulary Building Approach Step-by-step

1. Vocabulary Size Determination: The BPE algorithm begins by setting a predefined vocabulary size. In the case of GPT, this size is typically chosen as 50,000.
2. Collect a big corpus: A substantial corpus of text is collected.
3. Tokenization: Tokenize the corpus into words.
4. Add an end token: Add the end token underscore ("_") to each word.
5. Splitting words into characters: The words are further segmented into individual characters
6. Initial vocabulary: Create an initial vocabulary consisting of all unique characters including the end token. The tokens in the vocabulary are referred to as symbols.
7. Symbol pair merging: The core of the BPE algorithm involves merging the most frequent symbol pairs. When two symbols are frequently found together, they are merged into a single new symbol. Note that this step requires you to check throughout your entire corpus what pair of tokens (from your vocabulary) occur most frequently together in your corpus. If they appear once within a word that appears 5 times within your corpus, then that raises the count of that token pair to 5. In the image above, the token pair of 'r' and '_' is the most frequent and occurs 9 times in the corpus.
8. Extend the vocabulary: The new symbol is then added to the vocabulary.
9. Iterate steps 7 and 8: Continue iteration steps 7 and 8 and repeatedly merge symbol pairs until the vocabulary size reaches the predetermined limit from Step 1 or until all symbols have been merged.

WordPiece as a Variation WordPiece is a similar approach to BPE but uses language model likelihood and employs a "start of word" token instead of the "end of word" token.

4.2.1.4 Conclusion

BPE is a powerful algorithm for building vocabularies in NLP. It breaks down words into smaller units, allowing for efficient representation and analysis of text. The algorithm's flexibility makes it suitable for various NLP models and tasks, making it a valuable tool in the field of NLP.

Word-level tokens face problems with rare or unseen words aka out-of-vocabulary words (OOV). Character level tokens face no problems with OOV, but are too fine-grained and miss important information, they also require

sequence models. Sub-word tokens both are not too fine-grained and do not face problems with OOV.

4.3 Word Normalization

Word normalization is the process of representing words in a standardized format or choosing a single, normal form for words with multiple variations. This standardization ensures consistency and can be valuable in various NLP applications, such as information retrieval. E.g.

4.3.1 Normalization Techniques

Two common techniques for word normalization are stemming and lemmatization. The goal of methods is to reduce a word's inflectional and sometimes derivational forms to a common base form.

4.3.1.1 Stemming

Stemming is a process that reduces different forms of a word to its word stem. Essentially cuts the ends of words off For example, the word "students" might be reduced to "student" through stemming. Stemming is a quick and simple technique but may produce non-English words. The Porter's stemmer is a well-known and widely used stemming algorithm.

4.3.1.2 Lemmatization

Lemmatization is a more sophisticated than that takes into account vocabulary and morphological analysis to determine a word's base or dictionary form (i.e. the lemma). For instance, "better" might be reduced to "good" through lemmatization. Lemmatization is more accurate but can be slower than stemming. In lemmatization, the base form of a word is referred to as its "lemma." Understanding and identifying lemmas is important in achieving precise word normalization.

4.4 Stop Words

Stop words are common words in a language that do not carry significant meaning or information for a particular NLP task. Examples of stop words include "the," "was," "it," "there," and "but." These words are ubiquitous in nearly all text but often lack specificity and relevance to the core message of the text. Stop

word removal can reduce the dimensionality of the data, improve the efficiency of certain NLP algorithms, and enhance the accuracy of tasks like text classification and information retrieval. Stop word removal is primarily used as a preprocessing step when using bag-of-words representations (see Bag-of-Words).

Abbreviations

- NER - Named entity recognition
- BPE - Byte-Pair Encoding
- BERT - Bidirectional Encoder Representations from Transformers
- GPT - Generative Pretrained Transformer

Terminology

Word: A word is a single, distinct, and meaningful element of speech or writing. Depending on the context and task, words can include punctuation marks such as periods, commas, question marks, and similar characters.

Token or term: A token or term refers to an individual component of a text. It can be a character, a sequence of characters, a word, or a sequence of words. Tokens are the building blocks of text analysis.

Sentence: A sentence is a sequence of words that conveys a complete and coherent meaning. Sentences are fundamental units of communication and play a crucial role in text analysis.

Document: A document is an individual piece of text that is organized into sentences. Documents can vary in length and content, ranging from short texts like emails to longer ones such as research articles. A document is the unit of input in NLP-based ML, corresponding to a single row in a tabular dataset or an image in an image dataset.

Corpus: A corpus is a text or speech dataset made up of a collection of documents or speech recordings that are gathered and organized for analysis. Examples of datasets include the Brown Corpus, which comprises a million words sampled from 500 English texts of different genres, providing a diverse set of linguistic data.

Lemma: A lemma is the canonical, dictionary, or citation form of a set of words that share the same stem or base. It represents the major part of speech and the same word sense. For example, “run” is the lemma among the words “run,” “runs,” “ran,” and “running.”

Word form: A word form is the full, inflected, or derived form of a word. It encompasses all variations of a word based on factors like tense, number, and

grammatical role. For instance, “they run,” “she runs,” “he ran,” and “I am running” are all different word forms of the verb “run.”

Chapter 5

Language Modelling with N-grams

Written by Hjalmar K Turesson, ChatGPT, Krystaleah Ramkissoon and Darren Singh

5.1 Introduction

This chapter will first introduce into language modelling in general and N-gram language models in particular.

5.1.1 Content

- Introduction to Language Models and their role in NLP.
- N-gram Language Models.
- Limitations Of N-Gram Language Models.
- Generating Text With A Language Model.
- Evaluation of Language Models.

5.2 Introduction to Language Models

Language models assign probabilities to word sequences, like sentences.

$$P(w_1, w_2, ..., w_n) \tag{5.1}$$

The above equation should be read as *the probability of a sequence starting with word w_1, followed by word w_2 and so on until word w_n.*

There are two basic types language models: N-gram language models (simple and based on the frequencies of short sequences of words) and neural language models (advanced, using neural networks like recurrent neural networks and transformers).

Neural language models are autoregressively trained (trained to predict the next word in a sequence) for sequence modelling (see chapter 12).

5.2.1 When are language models useful?

Language models are useful in various applications, including:

- *Human-Computer Interface*: Used in chatbots and other human-computer interaction systems.
- *Optical Character Recognition* (OCR): Helps read handwritten text automatically.
- *Spelling and Grammar Corrections*: Assists in identifying and fixing errors in text.
- *Machine Translation*: Facilitates automatic translation between languages.
- *Augmentative and Alternative Communication Systems*: Enhances communication for individuals with speech disabilities, like Stephen Hawking.
- *Text Generation*: Generates text, as demonstrated by chatbots like ChatGPT.

5.2.2 The Probability of a Sequence

The probability of an N-word long sentence is estimated from a large corpus. It is the ratio of the number of times the N-word long sequence occurs and how many times all N-word long sequences occur.

$$P(w_1, w_2, ..., w_n) = \frac{C(w_1, w_2, ..., w_n)}{C(all\, n\, word\, long\, sequences)} \tag{5.2}$$

While this seems straightforward, the number of all N-word long sequences grows exponentially with N, and the task quickly becomes computationally intractable. For instance, given a vocabulary of 100,000 words, the number of different two-word sequences is 10 billion ($100,000^2$ or $10,000,000,000$) and of five-word sequences, there is 10 septillion (i.e. $100,000^5$, $10,000,000,000,000,000,000,000,000$ or 10^{24}), which rapidly becomes impractical to count.

To address this, the **chain rule of probability** (no relation to the chain rule in calculus used for derivatives of compound functions) is used. It breaks down the probability of a sequence into a product of conditional probabilities. That is, the probability of each word given the preceding words.

The chain rule makes it more manageable to estimate probabilities for longer sequences without having to count all possible combinations. For example, $P(w_3|w_1, w_2)$ is the probability of word w_3 occurring right after words w_1 and w_2. It is read as *the probability of word* w_3 *given words* w_1 *and* w_2.

Thus, the equation below states the probability of word w_1 multiplied by the probability of word w_2 following word w_1, multiplied by the probability of word w_3 following words w_1 and w_2 (in that order), and so on up to the probablitiy of word w_n following all the previous words $(w_1, w_2, ..., w_{n-1})$.

$$P(w_1, w_2, ..., w_n) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)...P(w_n|w_1, ..., w_{n-1}) \quad (5.3)$$

5.3 N-gram language models

N-gram language models are simple models that relies on the frequenices of sequences of words. An N-gram is a sequence of N words, such as:

- **Unigram** (1-gram): A single word (e.g., "wake").
- **Bigram** (2-gram): A sequence of two words (e.g., "wake up").
- **Trigram** (3-gram): A sequence of three words (e.g., "wake up this").

N-grams are not only used for language models but also as features in Bag-of-words models (see Text classification with Bag-of-Word models).

5.3.1 The probablity of an N-gram

Let's compute the probability of a specific bigram. For this, we start with a micro-corpus with only three sentences ("I am Sam," "Sam I am," "I do not like green eggs and ham"), and a bigram, for example, "I" given the start token. To do this, we count the occurrences of the sequence "<s>" (the start token) followed by "I" and divide it by the total number of occurrences of "<s>."

In this micro-corpus, the sequence "<s> I" occurs two times, and "<s>" alone occurs three times. So, the probability of the bigram "I" given <s> is 2/3. Or, put another way, there is a 0.667 probability that "`I`" begins a sentence.

5.3.2 The N-gram language model

To build a language model and estimate the probability of a sequence of words, we typically use the chain rule of probability. However, when dealing with longer sequences, such as a ten-word sentence, calculating the probability of the last word given all preceding words (a deca-gram) becomes challenging due to the vast number of possibilities.

To make this more manageable, we approximate the full product of all the conditional probabilities up to the length of the sequence with a product of lower-order N-gram probabilities. So, in a bigram language model, we approximate the sequence probability as the product of bigram probabilities, and in a trigram language model, we use trigram probabilities, and so on for N-gram models. For instance, a trigram language model:

$$P(w_{1:n}) \approx \prod_{t=1}^{n} P(w_t|w_{t-2:t-1}) \tag{5.4}$$

In such a bigram language model, only pairs of consecutive words are considered. That is, the highest factor in this model is the bigram itself. Similarly, a trigram language model focuses on triplets of consecutive words.

5.4 Generating Sentences With a Language Model

To generate a sequence with a bigram language model we follow these steps:

1. Start with a *start token*, <s>.
2. Randomly draw a new word w_2 according to $P(w^i|\text{<s>}\forall i \in V)$.
3. Append the word w_2 to the sequence.
4. Randomly draw the next word according to $P(w_t^i|w_{t-1})\forall i \in V$.
5. Repeat steps 3 and 4.
6. End when the *end token*, </s>, is drawn.

The start (<s>) and end (</s>) tokens are part of the vocabulary V.

The sequence, starting from the start token and ending with the end token, constitutes the generated text.

A bigram language model based with a small vocabulary can be shown as a table. For example, a bigram language model with the vocabulary $V = \{\text{go}, [text< /s >, \text{here}, \text{we}, \text{<s>}$ might looks as follows.

$w_t \setminus w_{t-1}$	go	/<s>	here	we	<s>
go	0.05	0	0.35	0.2	0.15
</s>	0.4	0	0.3	0.3	0.05
here	0.25	0	0.3	0.4	0.35
we	0.3	0	0.05	0.2	0.45
<s>	0	0	0	0	0

Note that in the table above, the rows represent the current word (w_t), and the columns represent the previous word (w_{t-1}) (an analogous statement is that the rows represent the next word and the columns represent the current word). The table states that the probability of the start token being followed by the end token is 0.05, and the probability of the start token being followed by another start token is 0 because this is not possible.

5.4.1 Limitations of N-gram language models

5.4.1.1 Low-order N-gram language models

Would a good bi-gram model assign the following "sentence" high or low probability?

It is it is it is it is it is it is it is it is it is.

Answer: Yes it would since both "it is" and "is it" are commonly occurring and thus, probable phrases.

However, would a good tri-gram model assign the following same "sentence" high or low probability?

Answer: No it wouldn't since neither "it is it" nor "is it is" are probable phrases.

This demonstrates that for good language models, higher-order N-grams needs to be included.

5.4.1.2 Higher-order N-gram language models

The limitations of N-grams become apparent when dealing with higher-order N-grams (e.g., three, four, or five-word sequences). These higher-order N-grams will capture on longer-range dependencies between words, and thus be better models of language. However, to obtain meaningful probability estimates for N-grams, it's essential that these sequences occur not only in the test data but also in the training data and with high-enough frequencies for their estimates to be reliable. If a particular N-gram, like "wake up," doesn't occur in the training corpus, it will have a count of zero, resulting in an undefined probability. As the

N-gram order increases, the number of unique possible sequences grows exponentially (combinatorial explosion), requiring an exponentially bigger corpus for reasonable probability estimates. Therefore, N-grams are limited by the data requirement, and they struggle with rare or unseen sequences. Exponentially more data is required to get non-zero counts of all/nearly all sequences.

5.4.1.3 Natural language is generative

Natural language is generative, allowing us to create novel sentences that still make sense. Consider, for instance, the following sentences:

Is, angry Norway and the small suicidal adorable lemming.

and

The suicidal Norway lemming is small, angry and adorable.

In both pairs of sentences above, one sentence is clearly more natural-sounding than the other. Assuming you haven't encountered either if these sentences before, this demonstrates that we can generate and comprehend novel sentences. This ability to comprehend and create previously unheard sentences highlights the flexibility and creativity of natural language. However, novel word sequences are a problem for N-gram-based language models.

5.4.1.4 Zero-counts

When you encounter sequences, such as bigrams, that are missing from the training data set (i.e., they have zero counts), you face two main problems:

- **Zero Probability**: Since these sequences never occurred in the training data, their probabilities are undefined, and you cannot assign probabilities to them.
- **Evaluation Complexity**: Due to the undefined probabilities, evaluating these sequences becomes problematic, as the perplexity (see The Evaluation Metric for Language Models) is undefined.

What should we do with sequences (e.g. tri-grams) missing from the training set? For example:

Training set

bigram continuation	count
denied the *allegations*	5
denied the *speculation*	2
denied the *rumors*	1

bigram continuation	count
denied the *report*	1

Test set

bigram continuation
denied the *offer*
denied the *loan*

To address this issue, you can employ techniques like *smoothing* or *back-off* methods. Smoothing methods involve redistributing some probability mass from seen sequences to unseen ones, making it possible to assign non-zero probabilities to previously unseen sequences. Back-off methods, on the other hand, involve using lower-order N-grams (e.g., unigrams or bigrams) when higher-order N-grams are missing. These techniques help mitigate the challenges posed by zero counts in N-gram language models.

5.4.1.4.1 Dealing with zero-counts Dealing with zero counts in language models can be addressed through smoothing techniques:

- **Add-One Smoothing** (Laplace Smoothing): Add one to all N-gram counts, including those with zero counts. Then, normalize by the total number of N-gram counts. While intuitive, it may not perform well when dealing with many zero counts, as too much probability mass is shifted to these zeros, resulting in poor estimates.
- **Back-Off Smoothing**: When counts for higher-order N-grams are missing, use the counts of a lower-order N-gram. For example, if the trigram count for "wake up this" is missing, use the bigram counts for "wake up" and "up this" instead. To improve accuracy, you can employ a weighted average over all N-grams (unigrams, bigrams, trigrams) in place of the missing higher-order N-gram. This technique is called interpolation.

Both smoothing methods help mitigate the problem of zero counts in N-gram language models.

5.4.1.4.2 Dealing with unknown words Dealing with unknown words in language models is done by either replacing all words not in the vocabulary with the unknown-word token, <UNK>, or replacing all words below some counts with <UNK>. The later is motivated by poor probability estimates of words with very low counts.

5.4.2 Google N-grams

Google released its N-grams dataset in 2006, which included up to 5-grams. This dataset was generated from a, at the time, massive corpus of one trillion words of running text. It was based on all five-word sequences that appeared at least 40 times, excluding those appearing less than 40 times. Given the enormity of this dataset, especially in the context of 2006, when computing power was less advanced, a fast and efficient smoothing method was essential for its handling.

5.5 Evaluating Language Models

Language models can be evaluated through two main approaches: extrinsic evaluation and intrinsic evaluation.

- **Extrinsic Evaluation**: In this approach, the language model is deployed within an application or task, and its impact on the application's performance is measured. For example, in the task of speech recognition, two different language models can be trained and used to recognize spoken words. The results are then compared to determine which language model improves the task's performance. Extrinsic evaluation assesses the model's real-world utility.
- **Intrinsic Evaluation**: In this approach, an evaluation metric that is independent of the application is used to assess the language model's quality. The model is evaluated on a test set, which typically consists of well-written human text, such as Wikipedia or newspaper articles. The best language model is the one that assigns the highest probability (or lowest perplexity, see The Evaluation Metric for Language Models) to the test set. Perplexity is the normalized inverse of probability and is used as an indicator of model quality.

Both extrinsic and intrinsic evaluations provide insights into the performance and quality of language models in different ways, with extrinsic evaluations focusing on real-world application performance and intrinsic evaluations examining the model's language modelling capabilities on specific datasets.

5.5.1 The Evaluation Metric for Language Models

The evaluation metric for language models is often expressed as the *perplexity** (PP), which is the inverse probability of the test set normalized by the number of words in the test set. Here's how it's calculated:

1. Using the language model calculate the probability of the test set, which is a sequence of words.

2. Take the inverse of that probability, which results in a larger number for easier comparison.
3. Normalize this value by the length of the test set (the number of tokens). The exponent (N) in the calculation represents the size of the test set.

$$W_{test} = w_1, w_2, ..., w_N \tag{5.5}$$

$$PP(W_{test}) = \sqrt[N]{\frac{1}{P(w_1, w_2, ..., w_N)}} \tag{5.6}$$

This normalization is crucial because it ensures that the metric remains comparable regardless of the length of the test set. It prevents shorter sequences from being systematically assigned higher probabilities than longer ones. This approach also helps avoid numeric overflow or underflow issues.

5.5.2 Language Model Generalization

The effectiveness of a language model depends on the data it was fitted to, as the model will reflect the characteristics of the training corpus. To achieve better generalization for your task, there are a couple of options:

- **Domain-Specific Training Data**: Train the language model on a corpus that is similar to the intended application. For example, if you want a language model for legal document generation, train it on legal texts. If you want it to write in the style of Shakespeare, train it on Shakespearean works. This approach ensures the model is more attuned to the specific language and context you need.
- **Large and Diverse Corpus**: Train a high-capacity model on a vast and diverse corpus that covers a wide range of topics, styles, and languages. Models like GPT-3 are trained on massive datasets containing various types of text, allowing them to learn different writing styles and languages. With sufficient capacity, the model can adapt to various contexts and styles.

The choice between these approaches depends on your specific use case and the level of generalization required for your language model.

5.6 Terminology

- **Language Model**: A model that assigns probabilities to sequences of words (sentences or text) to predict the likelihood of a word given the preceding words.

- **Vocabulary**: The set of all unique words or tokens in a given corpus.
- **N-gram**: A sequence of N consecutive words or tokens.
- **N-gram Language Model**: A language model that uses N-gram probabilities to estimate the probability of a sequence.
- **Perplexity**: A metric used to evaluate the performance of language models, measuring how well the model predicts a given dataset.
- **Extrinsic Evaluation**: The evaluation of a language model's performance within a specific application or task to measure its real-world utility.
- **Intrinsic Evaluation**: The evaluation of a language model's quality using metrics that are independent of any specific application.
- **Chain Rule of Probability**: A principle in probability theory used to calculate the probability of a sequence of events by breaking it down into a product of conditional probabilities.
- **LLM**: Large language model.

Chapter 6

Text classification with Bag-of-Word models

Written by Hjalmar K Turesson, ChatGPT, Darren Singh and Krystaleah Ramkissoon

6.1 Introduction

We explore the basics of text classification using bag of words (BoW) models and delve into various aspects of text representation, comparison, and data storage in the context of NLP.

6.1.1 Content

- Introduction to Bag of Words Text Classification and Sentiment Analysis.
- Bag of Words Text Representation, including discussions on Information Retrieval, Vector Comparison, and the Curse of Dimensionality.
- Data Storage for NLP.

6.1.2 Text classification

6.1.2.1 Applications

Text classification has various applications, including:

- **Spam Protection**: This was an early application using naive Bayes classifiers to distinguish between spam and non-spam (ham) emails.

- **Sentiment Analysis**: It classifies text as expressing positive or negative sentiment towards a subject, often using reviews, political texts, or articles.
- **Language Detection**: Identifying the language of a given text, commonly used as an initial step in NLP pipelines.
- **Topic Classification**: Sorting text into categories based on topics, such as research articles by subject or routing incident tickets to support teams based on the user's problem type.

6.1.2.2 Different ways of classifying texts

To illustrate different ways of classifying texts consider sentiment analysis.

- **Keyword Counting** (Simplest): This approach involves counting predefined positive and negative keywords in a text. For example, if there are more negative keywords than positive ones, it's labelled as expressing a negative sentiment. It's straightforward but inflexible, manual, and not very accurate.
- **BoW-based Text Classification**: This method uses ML to classify sentiment based on labelled data. It's relatively quick and flexible but doesn't consider word order or syntax.
- **Text Embeddings**: This advanced approach yields state-of-the-art results. It involves training models on large datasets, making it computationally intensive and is deep learning based. It's sensitive to word order and syntax but requires labelled data.

6.2 Bag-of-words text representation

Text can be represented for machine learning in two primary ways:

- **BoW**: This method counts the frequency of words or tokens in a document, representing the document as a long and sparse vector. It is suitable for long documents where word order is less important, but it ignores word order.
- **Text Embeddings**: Text embeddings represent documents as sequences of tokens (words, sub-words, or characters) that are transformed into dense vectors. This approach is commonly used in neural network models and is sensitive to word order, making it effective for shorter documents like single sentences. Individual words, sentences, paragraphs and longer texts can all be embedded in this way.

6.2.1 Bag-of-Words

Text can be represented using the BoW model in three ways of increasing complexity:

- **Binary Count**: This is the simplest representation, where words are represented as either present (true) or absent (false) in a document.
- **Term Frequency** (TF): In this representation, the frequency of each word in a document is counted to create a richer representation.
- **Term Frequency-Inverse Document Frequency** (TF-IDF): TF-IDF further enhances the representation by weighting the term frequency based on how unique a word is to a given document.

6.2.1.1 Binary count

In the binary count representation, words from a document are tokenized and put into a "bag" or collection without any specific order. Then, the vocabulary, which consists of all unique words in a corpus, is used to count whether each term occurs in the document. If a word is present, it's marked as true; if not, it's marked as false.

For example, given the document:

The strange but compelling Bacurau, plunging us into the hinterlands of Brazil, is the kind of modern-day Western that Sam Pekinpah might have made if he were a) Brazilian and b) alive.

- Parts of a review of Bacurau, written by Brian Viner & published in the Daily Mail (UK)

Term	Bin-count
bacurau	True
brazil	True
compelling	True
flowers	False
hinterlands	True
love	False
of	True
plunging	True
strange	True
the	True
western	True

This results in a binary representation of the document based on the presence or absence of words from the vocabulary.

6.2.1.2 Term Frequency

Term	count	TF
bacurau	1	1/30
brazil	1	1/30
compelling	1	1/30
flowers	0	0
hinterlands	1	1/30
love	0	0
of	2	1/15
plunging	1	1/30
strange	1	1/30
the	3	1/10
western	1	1/30

In the TF representation, instead of just noting whether a word occurs or not, we count how often it occurs in a document. For example, based on the document above: To calculate the term frequency, we divide the number of times a given word occurs by the total number of words in the document (plus 1). This gives us a representation of how frequently each word appears in the document.

TF measures how often a word appears in a document, calculated as the raw count of term t in document d, divided by the number of words in the document plus one. Sometimes, to prevent skewing by very high counts or exponential distributions, TF is squashed using a logarithmic function.

Raw TF of term t in document d: $TF(t, d) = C(t, d)/(N + 1)$.

Squash frequencies with log: $TF(t, d) = \log 10(C(t, d)/(N + 1))$.

Note that the denominator is the total number of words in the document plus 1, this is so that the term frequency of a word in an empty document equals 0.

However, the goal is to represent a term's relevance to a document. Words that frequently appear in a document are considered important. For example, if a document mentions words like "cats," "fish," and "cat food" frequently, we assume the document is about cats. To avoid bias from common words like "the," "is," and "of," which appear in many different types of documents (see Stop Words), they are often removed from the vocabulary to not overly influence the term frequency.

6.2.2 Raw Term Frequency

In text analysis, common words like "the", "it", and "they" often dominate the count but don't provide much insight into the document's content. While

high frequency words often are important, those appearing too frequently can become unimportant. The issue with raw term frequency is that it's skewed and not very discriminative, as common and unspecific words can have high frequencies.

Therefore, it's crucial to balance the frequency of words in a document and their uniqueness. The Term Frequency-Inverse Document Frequency (TF-IDF) method captures this balance. It combines Term Frequency (how often a term appears in a document) and Inverse Document Frequency (how few documents a term occurs in). The TF-IDF value increases with the word's frequency in the document but decreases if the word is common to many documents in the corpus.

6.2.2.1 Inverse Document Frequency

The IDF measures how unique or specific a word is to a document. It's calculated as the base-10 logarithm of the total number of documents divided by the number of documents containing a specific term.

For common words that appear in all documents, like "the" or "off," IDF is zero (e.g., $N_{doc} = 100$, $C(t) = 100$, $IDF = \log 10(100/100) = \log 10(1) = 0$).

For unique words that appear in only a few documents, IDF is high because the term occurs in a small fraction of the documents (e.g., $N_{doc} = 100$, $C(t) = 1$, $IDF = \log 10(100/1) = \log 10(100) = 2$).

In summary, the fewer documents containing a term, the higher its IDF, and the more documents containing it, the lower the IDF. A word appearing in many documents implies less importance since it is not a unique identifier.

6.2.2.2 TF-IDF (Term Frequency-Inverse Document Frequency)

TF-IDF is a composite measure that combines two crucial aspects to determine the importance of words within a document.

- **TF**: Measures how frequently a word appears in a document.
- **IDF**: Measures how unique a word is to a document across all documents in the corpus.

By multiplying these two terms, TF-IDF balances the importance of a word based on both its frequency in a document and its uniqueness within the corpus. Frequent words specific to only a small number of documents receive a high TF-IDF score, indicating their significance within those documents.

- Frequent words occurring in all documents receive a low TF-IDF score.

- Infrequent words appearing in only a small number of documents also receive low TF-IDF scores.
- Frequent words occurring in a small number of documents receive a high TF-IDF score.

TF-IDF is a powerful tool in text analysis and information retrieval, helping to identify words that are both frequently used within a document and distinctive to that document within a larger collection.

6.2.2.3 Simple information retrieval and document comparison

In *information retrieval* (IR), the goal is to find the document (d) in a collection of documents (D) that best matches a query (q). Both the query and documents can be represented as vectors based on word counts in a common vocabulary.

Imagine these vectors as points in a space with the number of dimensions equal to the vocabulary size.

Example: Four documents and a mini vocabulary with the words "battle," "good," "fool," and "wit." We can think of each of these documents as a vector in a four-dimensional space with each axis representing the count of a word. Since some documents are longer than others the vectors magnitude differ. However, documents with a similar distribution of words are represented by vectors pointing in the same direction. Thus, to measure the similarity between documents, we need a method that measures similarity of direction but not magnitude.

The *cosine similarity* measures the angle between two vectors, indicating how similar their directions are. It is insensitive to vector length, making it ideal for comparing documents of varying lengths. If the angle between two vectors is close to 0° (the vectors are nearly on top of each other), then the cosine argument approaches a maximum (1), which means the documents are very alike. If the angle between the two vectors is approximately 90° (the vectors are orthogonal), the cosine argument approaches a minimum (0), meaning that the documents are not alike.

Thus, the cosine similarity allows us to measure document similarity based on their direction in a multi-dimensional space, making it a useful tool in information retrieval and document comparison.

6.2.3 Hapax Legomenon words and sparsity

In real corpora, vocabularies can consist of millions of words, and many of these words appear only once, known as Hapax Legomenon words. This leads to high-dimensional and sparse document vectors with many zeros. Dealing with such high dimensionality poses challenges, including increased computational

requirements, memory usage, and difficulties in machine learning due to the curse of dimensionality.

Most ML algorithms need the space to be (relatively) densely populated to fit the parameters. But, with increasing numbers of dimensions the amount of data required for the same density increases exponentially.

6.2.4 The curse of dimensionality

The curse of dimensionality in ML refers to the problem that many ML algorithms require data points to densely populate the space to estimate parameters accurately. In statistics, having more samples results in better parameter estimates, like the mean, with 30 samples often considered a rule of thumb for reasonable estimates. However, in high-dimensional spaces, the curse of dimensionality becomes apparent. The more dimensions in the dataset, the more weights/parameters are generally required in the model. As the number of dimensions increases, the amount of data required to maintain the same data density increases exponentially. With only 100 data points in one dimension, you can get good parameter estimates, but in two, three, or more dimensions, the data becomes exponentially sparser, making parameter estimation more challenging.

6.2.5 BoW for text classification

TF-IDF can be used for text classification by treating the transformed word counts as features. Each document is represented by its vector of TF-IDF values, and a target label is assigned to each document (e.g., comedy, drama). The BoW representation and corresponding labels are then then used to fit learners like Naive Bayes, logistic regression, support vector machines, and decision trees to get text classification models. Generally learners that work well with many features and few examples are preffered.

6.2.6 Limitation of bag-of-words

The BoW approach has several limitations:

- **Vocabulary Size**: The vocabulary must be kept small to avoid very sparse document vectors, which can limit its ability to represent a wide range of words.
- **Sparsity**: Sparse representations are computationally complex and can struggle to capture subtle information due to the many zero values in the vectors.

- **Lack of Word Meaning Differentiation**: BoW representations treat all words as equally distant from each other, meaning that the BoW representation doesn't capture nuances in word meanings, such as synonyms and antonyms. For example, synonyms like "exhausted" & "fatigued" should be closer in the representational space than antonyms like "exhausted" and "replenished". But, with BoW they are not.
- **Sentence meaning**: BoW disregards word order, resulting in a critical loss of information for tasks like sentiment analysis, where the meaning can change based on word order. For example, "this is interesting" vs "is this interesting".

These limitations can impact the model's ability to generalize effectively and differentiate between sentences with similar words but different meanings.

6.2.6.1 A BoW Hack

One way to address the issue of word order in text classification, especially for short sentences, is to incorporate bigrams or higher-order N-grams into the analysis (see N-gram language models). For example, the sentence "Bag of words is not good" might be classified as positive because of the word "good," even though the phrase "not good" is negative. Adding bigrams can resolve this. By adding bigrams or n-grams, the classifier can learn to consider word sequences and capture more nuanced meanings. However, it's essential to note that including n-grams will increase the dimensionality of the feature space, which can have computational and memory implications.

6.2.7 Data storage for NLP

When it comes to data storage for NLP, several factors should be considered:

- **Convenience**: The storage format should be easy to inspect, read, and write. It should also have good support for the programming language you are working with.
- **Metadata and Structured Data**: Consider whether you need to store additional information like class labels, targets, dates, or any other metadata associated with the text documents.
- **Handling Special Characters**: Be aware of how the chosen file format handles special characters. For example, in CSV files (Comma-Separated Values), sentences containing commas may be misinterpreted as new columns, leading to data-splitting issues.
- **Input/Output Speed**: For large datasets, especially when dealing with usage data and deep learning, input/output speed is crucial. Efficient data reading and writing are essential, especially if you need to continuously train models on large datasets.

- **Batch Processing**: Deep learning often requires reading data in batches during training, particularly when the dataset is too large to fit into RAM or GPU memory. The chosen storage format should allow you to load and process data one batch at a time.

For normal-sized datasets in NLP, plain text files (e.g., .txt) are convenient, easy to read and write, and widely supported. However, they lack metadata. Another option is to use CSV or TSV (tab-separated values) files, which can store metadata along with the data. However, they can encounter issues with special characters. JSON (JavaScript Object Notation) is a format suitable for structured data, offering good support and no special character issues. It can be less convenient to read directly but provides the advantage of storing metadata.

For very large datasets, you can compress text files using gzip, but this doesn't solve the RAM issue since the files need to be extracted for processing. To handle massive datasets that don't fit in memory, you need to read data in batches during deep learning training. Different deep learning frameworks offer specific data formats for this purpose, like FastAI's TextDataBunch, TensorFlow's TextLineDataset, and PyTorch's custom dataset formats.

In summary, the choice of data storage format depends on your dataset size, metadata needs, special character considerations, and the requirements of your machine learning framework.

6.3 Terminology

- **Term Frequency (TF)**: It measures how frequently a term appears in a document. It's the raw count of a term in a document.
- **Bag of Words (BoW)**: A text representation method that counts the frequency of words in a document, ignoring word order and context.
- **Inverse Document Frequency (IDF)**: A measure of how unique or specific a word is across a collection of documents. It's calculated as the logarithm of the total number of documents divided by the number of documents containing the word.
- **Term Frequency-Inverse Document Frequency (TF-IDF)**: A composite metric that combines TF and IDF to assess the importance of a term in a document. It helps in information retrieval and text classification.
- **Topic Classification**: The task of categorizing text documents into predefined topics or categories, such as news articles into sports, politics, or entertainment.
- **Curse of Dimensionality**: Refers to the exponential increase in data volume and computational complexity as the number of dimensions (features) in a dataset grows.

- **Information Retrieval (IR)**: The process of finding relevant documents or information from a large collection based on a user query. It often involves comparing document vectors to measure similarity.
- **Cosine Similarity**: A similarity measure that quantifies the cosine of the angle between two vectors. It's commonly used to assess how similar two document vectors are in text classification and information retrieval tasks.

Chapter 7

Topic Modeling & Latent Dirichlet Allocation

Written by Hjalmar K Turesson, Stable Beluga 2, Darren Singh and Krystaleah Ramkissoon

7.1 Introduction

We will discuss topic modelling, specifically, latent Dirichlet allocation (LDA). We will first explain topic models and their applications, then delve into the principles of LDA. This will involve understanding the Dirichlet distribution and combining it with our comprehension of topic modelling to arrive at LDA. Next, we will explore how to learn an LDA topic model and look at its practical applications in NLP. Lastly, we will discuss methods for evaluating topic models to ensure accurate and effective performance.

7.1.1 Content

7.2 Topic Modelling

7.2.1 The Bag-of-words (BoW) Text Representation

The BoW representation (e.g. TF-IDF) is often used to analyze text data. However, The size of the vocabulary ($|V|$) can easily be $> 10\ 000$, which may pose challenges for machine learning (ML). In standard, non-NLP ML, this can be addressed with dimensionality reduction techniques such as Principal Component Analysis (PCA). In NLP, specific methods like Latent Semantic Analysis

(LSA) and LDA are utilized for the same purpose. Today, we will focus on the latter - LDA - for Topic Modelling as a means to handle a large number of features in text data for ML.

7.2.2 What is a Topic Model?

Topic modelling is an unsupervised learning approach for identifying abstract topics within a given text corpus. The topics are not explicitly labelled (otherwise, it would be supervised learning), allowing the algorithm to discover them. Intuitively, a document discussing a particular subject should contain certain keywords or phrases with varying frequencies. For instance, 'dog' and 'bone' should be more frequent in texts about dogs, while 'cat' and 'meow' should be more common in texts about cats. In topic models, the produced topics are clusters of related words occurring weighted by their frequency of occurrence. In addition, topic modelling treats documents as mixtures of topics in different proportions. For instance, a document may be 80% about cats and 20% about dogs since cats and dogs are both pets.

In the visualization above the vertical axis represents various terms and the horizontal axis represents different documents. A darker square represents that a word has a high frequency in that document. Thus a group of dark squares implies that these documents share similar topics, since they share similar words with high frequencies within them. This is a visualization of clusters bring created. Furthermore in LDA a topic is a collection of similar words that appear frequently. Typically we do not know how many topics our corpus has and we have to treat it like a hyperparameter (e.g. educated guessing and hyperparameter tuning).

7.2.3 Applications of Topic Modelling

Topic modelling finds applications in various aspects of NLP. A common application includes text mining, where large amounts of text data are organized into distinct clusters or topics. Another application is in recommendation systems, where documents are grouped by similarity to suggest relevant content to users based on their preferences. Additionally, topic modelling can be used for dynamic text analysis to track the evolution of topics over time. Originally, topic modelling was introduced in population genetics as a method for scaling probabilistic models of genetic variation among millions of humans.

7.3 Latent Dirichlet Allocation

LDA was originally introduced in population genetics by J. K. Pritchard, M. Stephens and P. Donnelly in 2000 and, later in 2003, applied to NLP and text

mining by David Blei, Andrew Ng and Michael I. Jordan. LDA decomposes a term-frequency matrix into a document-topic matrix and a topic-term matrix. This approximates the term-frequency matrix by two smaller matrices, representing the probability distribution of topics given a document and the distribution of terms given a topic. By decomposing the term-frequency matrix, LDA can effectively model the relationships between documents and topics, improving our ability to understand and analyze text data. Decomposes a term-frequency matrix (terms in rows & documents in columns) into the product of a tall and skinny and a short and wide matrix.

7.3.1 A Simple Description of LDA

LDA can be intuitively understood as having two layers of aggregation. The first layer represents the distribution of topics, such as finance news, weather news, and political news. The second layer involves the distribution of words within each topic, for instance, terms like "sunny" and "cloud" in weather news and "money" and "stock" in finance news.

These two layers correspond to the document-topic matrix and the topic-term matrix in LDA. By decomposing the term-frequency matrix into these two matrices, we can gain insights into the relationships between documents and topics, enabling effective analysis of text data.

7.3.2 Term-frequency

The aim of LDA is to approximate the term-frequency matrix, document-topic matrix and topic-term matrix. To approximate the frequency of a particular term in a document, we perform the following steps:

1. Select a document (e.g., Document 2) and a term of interest (e.g., CO2).
2. Find the probability of different topics for the given document (a row vector).
3. Multiply the probability vector by the distribution of the term of interest over the topics (a column vector).

The result of this multiplication will provide an approximation of the frequency of the term CO2 in Document 2.

Notice that the matrix on the right has many more elements (500,000) than the matrices on the left (together: 4500 elements).

We can represent the probability of a term appearing in a document as a function of two other probabilities:

- The probability of a term (t) appearing in a specific topic (z): $P(t|z)$.

- The probability of a topic (z) occurring in a given document (d): $P(z|d)$.

The probability of a term appearing in a document is then equal to the sum of the probabilities that topics occurring in the document multiplied by the probabilities of the term appearing in the topics: $P(t|d) \approx \sum_z P(z|d)P(t|z)$.

This captures the underlying relationships between terms, topics, and documents, and serves as the basis for the LDA model.

- t - term/token/word
- d - document
- z - topic

7.3.3 The Dirichlet Distribution

The Dirichlet distribution ($\text{Dir}(\alpha)$) is a fundamental component of the LDA model. $\text{Dir}(\alpha)$ is actually not a single distribution, but a family of multivariate probability distributions that generalizes the univariate beta distribution. It is parameterized by a vector called alpha (α), which controls the shape of the distribution.

The length of α is equal to the dimensionality of the distribution; for example, for a probability distribution over 10 topics, the length of α should be 10. If all elements in α are equal, we have a symmetric distribution with an equal distance between the peaks and the corners. If the elements are unequal, the distribution will be asymmetric, with the peaks closer to one or two corners.

If the elements in α are less than 1, we have a concave distribution with the peaks at the corners and the lowest points in the middle. Conversely, if the elements in α are greater than 1, we have a convex distribution with the peaks in the middle and the lowest points at the corners. The vector α, therefore, plays a crucial role in controlling the shape of the distribution used in LDA to estimate the probability of terms given topics and the probability of topics given documents. In LDA we would like to have a concave distribution for the topic-document classification, as we would not want the most probable point in the distribution to be documents made up of an equal mixture of all topics. Instead the most probable points should be documents that belong to a single or a few topics.

α: a vector of the same dimensionality as the distribution

- E.g. for a probability distribution over 10 topics, $|\alpha| = 10$
- If all elements in α are equal $\rightarrow$ symmetric distribution
- Unequal elements $\rightarrow$ asymmetric distribution
- Elements $< 1 \rightarrow$ peak in corners (concave)
- Elements $> 1 \rightarrow$ peak in middle (convex)

7.3.3.1 The Dirichlet Distribution and LDA

In LDA, the Dirichlet probability distribution is utilized to model the relationship between topics and documents and between terms and topics. When used in LDA, we assume that:

1. Topics are represented as corners on the distribution. For instance, in a three-dimensional Dirichlet distribution, each corner represents one of the three topics.
2. A point on the distribution indicates the probability of a particular combination of topics. For example, a point at the corner represents a document with a 100% probability of the corresponding topic and a 0% probability for the other topics. 3. A point in the middle of the distribution represents an equal combination of all topics.
3. A document is more likely to be a mix of a small subset of the topics rather than an even mix of all possible topics.

This leads to a concave distribution, where the peaks (i.e. highest probabilities) are in the corners. This favors the assignment of one or a few topics per document. If, in contrast, documents were assumed to be evenly mixed possible topics, a convex distribution with the highest probability in the middle would be a better choice.

7.3.3.2 Simplex

A Dirichlet distribution is an example of a simplex. A simplex is a generalization of a triangle that allows for any combination of two topics to be equally probable. It is a shape with one edge connecting every corner, and all edges have equal lengths. In the context of LDA and the Dirichlet distribution, a simplex helps represent the relationship between topics in a document, where each corner represents a specific topic.

7.3.4 Putting the Pieces Together

Combining the concepts of a topic model and the Dirichlet distribution we arrive at the LDA model where a document is composed of n topics, which can be represented as a point in an n-dimensional Dirichlet distribution ($P(z|d)$). Similarly, a topic is made up of k words that are represented as a point in second, k-dimensional, Dirichlet distribution ($P(t|z)$).

To better understand, let's consider the example of three topics: science, politics, and sports. We have a document that consists of 80% science, 20% politics, and 0% sports. In the n-dimensional Dirichlet distribution, the corners represent the three topics: science, politics, and sports.

This document's point would be found on the edge between science and politics but closer to science (0.2 from the science corner and 0.8 from the politics), as the document has a higher proportion of science content.

The topic science would in a similar way be represented by a point k-dimensional Dirichlet distribution, where $k = 9$ in the example below.

7.3.5 LDA Assumptions

There are several assumptions that form the basis of the LDA model. These assumptions are as follows:

1. Documents covering similar topics contain similar words. Although this assumption may not always hold true, such as in cases where documents are in different languages or written by authors from different time periods, it is generally reasonable.
2. Documents can be modelled as probability distributions over latent topics. Although documents are not literal probability distributions, this assumption is useful and effective in many cases. It considers documents as unordered collections of words sampled from probability distributions over latent topics.
3. Topics are probability distributions over words. Again, this assumption does not always hold, such as when the same words reflect different topics.

While these three assumptions don't always hold, they provide a reasonable foundation for the LDA model.

7.3.6 Learning an LDA Model

We want to find $P(t|z)$ and $P(z|d)$ that best approximates the TF-matrix. I.e. we want to maximize the likelihood of $P(t|z)$ and $P(z|d)$.

Learning an LDA model involves determining the probability of topics over documents and the probability of terms over topics. This model is generally learned through some version of the Expectation-maximization (EM) algorithm. An LDA model is a generative model that can generate fake documents that, in turn, can be compared to how closely they approximate the term-frequency matrix of the training corpus.

Generate a fake document (d_1):

1. Pick a random point in the Dirichlet distribution over topics $rightarrow$ topic distribution for the fake document. E.g. $P(z|d_1) = [0.8, 0.2, 0]$

2. Based on this topic distribution, sample one topic per word in d_1. E.g. if d_1 1 has 5 words, approximately 4 (80%) of them will be from Science and 1 (20%) will be from Politics
3. For each topic pick a random point in the Dirichlet distribution over words (not shown). E.g. $P(t|z = sci) = [0.2, 0.3, 0.1, 0.2, 0.1, 0.025, 0.025, 0.025, 0.025]$
4. Populate the fake document (d_1) with words by sampling the topic vector from step 3, according to $P(t|z = sci)$

Using these two probability distributions, a fake document is created by sampling words according to the topic and term probability distributions. This fake document is then compared to the real corpus (i.e. the term-frequency matrix). By updating the two probability distributions, the next generated fake document will be a better fit for the data, and the process continues until the model is refined and the generated documents closely match the real data.

Learning LDA distributions involves an iterative learning algorithm that starts with randomly initializing n-topic distributions and k-word distributions. These distributions are represented as points in the Dirichlet space.

The process then proceeds as follows:

1. Sample topics and words distributions to populate fake documents with words, as previously explained.
2. Compare the fake documents with the real documents, and measure their similarity.
3. Update the topic and word distributions to make the fake documents more similar to the real documents.

This process is iterated until there is no longer any improvement in the similarity between the fake and real documents.

7.3.7 An Application of LDA

An application of LDA is to measure document similarity, which can be useful for recommendation systems. Consider the example of an online library of magazines with thousands of documents. If a user has expressed interest in a particular magazine, LDA can be employed to recommend other magazines based on their historical interests and similar topics.

7.3.7.1 Fitting the Model

To measure document similarity using LDA, we must first fit a model to data. In this case, we can use Wikipedia articles as the dataset to fit the model. After pre-processing the data by removing metadata and retaining articles, we

can normalize the data by retaining words in the middle 90th percentile (drop highest and lowest 5%) and using TF-IDF and N-grams for text representation.

Next, we can fit the LDA model to this TF-IDF matrix and evaluate and tune the hyperparameters. With the fitted model, we can take all the articles or magazines in our library and obtain their topic distributions. This probability vector over the topics will represent a document (i.e. a magazine). Thus, applying the LDA model to our documents we obtain a probability vector of topics for each document (e.g. a 20-dimensional vector if we set the number of topics to 20).

7.3.7.2 The Similarity Metric

To assess the similarity between documents, we can use a distance measure like the Jensen-Shannon divergence. We can then recommend other magazines based on their similarity to a user's interests and historical reading preferences.

Jensen-Shannon divergence is a method to measure the distance between two probability distributions. It returns a value between 0 and 1, where 0 indicates the two distributions are the same, and 1 signifies they are completely different.

$$JSD(P_{d_1} \| P_{d_2}) = \sqrt{\frac{D_{KL}(P_{d_1} \| M) + D_{KL}(P_{d_2} \| M)}{2}} \tag{7.1}$$

Where D_{KL} is the Kullback-Leibler Divergence and $M = \frac{P_{d_1} + P_{d_2}}{2}$.

Using the Jensen-Shannon distance, we can calculate the similarity between two documents by considering their differences and mean probabilities. Now, equipped with this measure of similarity, we can determine which magazines or documents are most closely related to a user's interests based on their historical reading preferences and make appropriate recommendations.

7.3.8 Evaluation of LDA and Topic Models

The evaluation of LDA and topic models presents a challenge, as unsupervised learning does not provide the ground truth. Consequently, it becomes difficult to determine if the identified topics align with the actual subject matter. To address this problem, researchers must make assumptions and employ automated evaluation methods, human evaluation techniques, or a combination of both. Automated evaluation techniques are important for rapid development (i.e. model selection), but for a final evaluation, before deployment, human evaluation provides a more precise and contextual understanding of the performance.

7.3.8.1 Manual Evaluation

The manual evaluation of LDA and topic models involves human subjects who assess the quality of the generated topics and their associated word distributions. There are two main approaches: word intrusion and topic intrusion.

7.3.8.1.1 Word Intrusion Word intrusion focuses on assessing whether the probability of a word belonging to a specific topic is accurate. The process involves selecting the five most probable words for a given topic, as well as the least probable word that is a top word in another topic. Human subjects then determine which word is the intruding word, i.e., the one that does not fit well with the given topic.

Let's consider an example for each of the four generated topics:

floppy	alphabet	computer	processor	memory	disk
molecule	education	study	university	school	student
linguistics	actor	film	comedy	director	movie
islands	island	bird	coast	portuguese	mainland

The human evaluators guess which word is the intruding word while not knowing what the actual intruding word is supposed to be. The evaluation metric is model precision, which is the fraction of subjects agreeing with the model.

In summary, word intrusion evaluation helps to assess the relevance of words assigned to specific topics, using human judgment to identify intruding words and determine the overall quality of the topic model.

7.3.8.1.2 Topic Intrusion Topic intrusion is used to evaluate the probability of topics given documents. To accomplish this, the most probable words from the most probable topics are selected for a given document and shown together with the most probable word from a low-probability topic.

Human subjects are then shown a snippet of the document and asked to identify the intruding low-probability topic. For example, consider the snippet about Douglas Richard Hofstadter:

Douglas Richard Hofstadter (born February 15, 1945 in New York, New York) is an American academic whose resarch focuses on consciousness, thinking and creativity. He is best known for ", first published in"

student	school	study	education	research	university	science	learn
human	life	scientific	scientist	experiment	work	idea	
play	role	good	actor	star	career	show	performance
write	work	book	publish	life	friend	influence	father

Given this snippet, three rows of words are sampled from the most probable topics, and one row of words is sampled from the least probable topic. The human subject's task is to determine which row represents the intruding low-probability topic.

In this case, the least probable topic, represented by words such as "play role," "good," and "actor," can be identified as intruding because there is no indication in the snippet that Hofstadter is associated with acting or plays. The other three topics, focused on education, science, and writing, are more relevant to the snippet content.

In summary, topic intrusion helps assess the accuracy and relevance of topics assigned to specific documents, using human judgment to identify intruding topics and determine the overall quality of the topic model.

7.3.8.2 Automatic Evaluation

Automatic evaluation for LDA and topic models is based on the assumption that words from the same topic should occur in the same documents. This evaluation can be conducted either by using the training corpus, known as intrinsic evaluation or by using an external corpus like Wikipedia, called extrinsic evaluation.

In automatic evaluation, topic coherence is an essential factor. Topic coherence is the summed scores assigned to pairs of words. Different scoring methods can be used, such as the extrinsic UCI score and the intrinsic UMass score. These scores are used to assess the quality of the generated topics based on pairwise scores of the words representing a topic.

For the extrinsic UCI score, we take the log of the probability of two words appearing together on Wikipedia divided by the product of their individual probabilities. This score uses Wikipedia as an external corpus and is thus extrinsic (provided the model was not trained on Wikipedia).

$score_{UCI} = \log \frac{P(t_i,t_j)}{P(t_i)P(t_j)}$,

where $P(t_i) = \frac{D_{Wikipedia}(t_i)}{D_{Wikipedia}}$ and $P(t_i,t_j) = \frac{D_{Wikipedia}(t_i,t_j)}{D_{Wikipedia}}$.

The intrinsic UMass score is computed from the training data. The number of documents containing both words in a pair is divided by the number of documents containing the first word in the pair. We then take the log of the result to obtain the intrinsic UMass score.

$score_{UMass}(t_i,t_j) = \log \frac{D(t_i,t_J)+1}{D(t_i)}$

Both scores assign a single-valued score to each word pair, and the coherence is derived by summing up these values for all word pairs in a given topic.

$Coherence = \sum_{i<j} score(t_i,t_j)$

7.3.9 Shortcomings of LDA and Alternatives

The limitations of LDA and topic models can be attributed to the BoW approach, which has several drawbacks. One of the primary challenges faced by LDA is the loss of word order, which becomes particularly problematic for short documents like single sentences. As a result, it becomes difficult to create good topic models for such documents.

Another shortcoming also stems from the BoW representation, which does not provide an effective way to represent word relatedness, such as synonyms and antonyms.

However, there are alternatives to BoW-based topic modelling. One of these alternatives involves topic modelling in embedding spaces. By encoding documents using neural networks, each document can be represented as a dense vector, allowing for clustering into topics using a wide selection of powerful clustering algorithms.

7.4 Terminology

- Topic model
- Latent Dirichlet Allocation (LDA)
- Dirichlet distribution Dir(α)
- Alpha (α) with respect to a Dirichlet distribution
- Expectation-maximization (EM) algorithm
- Jensen-Shannon divergence
- Word intrusion
- Topic intrusion
- Extrinsic evaluation
- Intrinsic evaluation
- Topic coherence
- UCI score
- UMass score

Chapter 8

Part-of-speech Tagging and Dependency Parsing

Written by Hjalmar K Turesson, Stable Beluga, Stable Chat, Krystaleah Ramkissoon and Darren Singh

REMAKE THE LECTURE. FOCUS IT ON FEATURE ENGINEERING WITH POS AND DEPENDENCY PARSING AS FEATURES (remove the POS and Dependency parsing algorithms, probably keep evaluation methods). SEE: https://spacy.io/usage/linguistic-features AND https://vtechworks.lib.vt.edu/server/api/core/bitstreams/398ffe2f-4690-4c0a-96b7-f17ef913b878/content USE SPACY FOR THE TUTORIAL: https://explosion.ai/blog/sp-global-commodities or https://explosion.ai/blog/nesta-skills and https://github.com/nestauk/ojd_daps_skills

8.1 Introduction

This lecture will discuss part-of-speech (POS) tagging. In NLP, the goal is to analyze and understand human language in order to perform various tasks such as text classification, machine translation, and speech recognition. One of the fundamental tasks in NLP is POS tagging, which involves assigning the correct part of speech to each word in a given sentence. This process is essential for understanding the grammatical structure of the text and is useful for various NLP applications.

Dependency parsing is an NLP technique. We will cover the uses of parsing, followed by a discussion on parse trees, shallow parsing, dependency structure, and the motivation for dependency parsing. Next, we will delve into dependency grammar, transition-based dependency parsing, and evaluation of dependency parsers.

How to use PoS tags and dependency parsing to augmentj BoW representation. Feature construction, feature engineering

8.1.1 Contents

8.1.2 Sentence-level NLP Tasks

There are several sentence-level NLP tasks that we will briefly introduce. Co-reference resolution. In this task, we identify clusters of mentions in a text that refer to the same underlying real-world entities. For example, in the sentence "I voted for Trudeau because he was most aligned with my values, she said," we need to determine that 'I', 'my' and 'she' belong to the same real-world entity, and 'Trudeau' and 'he' belong to another real-world entity. Dependency parsing (see MMAI 5400 lecture 6), which aims to describe the syntactic structure of a sentence in terms of the words and associated binary grammatical relationships. This information is crucial for understanding the meaning of sentences and is helpful in tasks like co-reference resolution, question answering, and information extraction. An example of dependency parsing is determining the relationships between the words in the sentence "I prefer the evening train through Ottawa," where the subject 'I' prefers 'the train,' and 'evening' and 'through Ottawa' provide additional information about the train.

In conclusion, part-of-speech tagging, co-reference resolution, and dependency parsing are essential sentence-level tasks in NLP. They contribute to a better understanding of natural language and are fundamental for many NLP applications. Part-of-speech Tagging POS tagging, also known as grammatical tagging, is a critical NLP task. It involves determining the word class or syntactic category for each word in a given text. In English, common parts of speech include noun, verb, adjective, adverb, pronoun, preposition, and conjunction. Since many words have more than one possible part of speech, POS tagging is a disambiguation task. Accurate POS tagging is important in many applications, such as named entity recognition, sentiment analysis, question answering, word sense disambiguation, and grammar and spell checkers. An Example Consider the sentence, "Bobo had a red nose." Here, 'Bobo' is a proper noun (like a person's name), 'had' is a verb in past tense, 'a' is a determinant, 'red' is an adjective, and 'nose' is a singular noun. This is an example of a POS-tagged sentence.

Ambiguity in POS Tagging However, POS tagging can be ambiguous because many words have multiple possible parts of speech. For instance, the word 'back' has six different parts of speech: adjective, singular noun, verb in past tense, verb in present tense, particle, and adverb.

Here are examples demonstrating the various parts of speech for the word 'back': "Earnings took a back seat." (Adjective) "A small building in the back." (Singular noun) "A clear majority of senators back the bill." (Verb in past tense)

"Dave began to back toward the door." (Verb in present tense) "Enable the country to buy back about debt." (Particle) "I was 21 back then." (Adverb)

Other examples of ambiguous words include 'down,' 'put,' and 'sit.' As you can see, numerous words can have multiple parts of speech, making POS tagging a challenging but important task in NLP. The Penn Treebank POS Tag Set The Penn Treebank POS tag set consists of 45 tags. The table (first image) below shows a subset of these tags, while the complete list can be found in Figure 8.2 of the textbook "Part of Speech and Language Processing." (second image).

Part-of-speech Tagging Algorithms We will explore two different POS tagging algorithms. For the sake of illustration and simplicity, we will use only three simplified tags:

1. N (Noun): This tag is used for nouns, such as house, spike, rock, llama, and Hjalmar. We have combined several noun-related tags into this single category.
2. M (Modal Verb): This tag represents modal verbs, such as will, could, would, should, and may.
3. V (Verb): This tag is for verbs like run, jump, skip, and ride. We have combined several verb-related tags into this single category.

It is important to note that these simplified tags are used for demonstration purposes only, and the actual POS tagging algorithms will consider a much larger set of tags. The Baseline Algorithm: The Most Frequent Tag The baseline algorithm for POS tagging is the most frequent tag algorithm based on frequency counts from a given corpus.

Let's consider an example dataset consisting of two sentences:

In this dataset, the words are tagged as nouns (N) and verbs (V). To apply the most frequent tag algorithm, we create a table with the counts of each word's parts of speech across the entire dataset:

We use this table to tag a new sentence: "Liv saw Will." The table shows that "Liv" is always a noun in the dataset, so we assign the noun (N) tag in the new sentence. "Saw" is always a verb in the dataset, so we assign it the verb (V) part of speech in the new sentence. Lastly, "Will" is always a noun in the dataset, so we assign the noun (N) part of speech in the new sentence.

The resulting part-of-speech tagged sentence is Liv [N] saw [V] Will [N]. The most frequent tag algorithm achieves an accuracy of around 92.34% on the Wall Street Journal Corpus. However, more advanced POS taggers achieve around 97% accuracy, indicating that there is still room for improvement in POS tagging algorithms.

The baseline or most frequent tag algorithm is a widely used approach in POS tagging. It is especially useful to compare other, possibly better algorithms. A limit of the algorithm becomes apparent when disambiguating words with

multiple parts of speech. For example, consider this (slightly bigger) mini-corpus:

Here, the word "will," functions as both a noun and a modal verb. However, a most-frequent-tag algorithm would always tag "will" as a modal verb because that is the most frequent tag for that term in the corpus.

Improving on the Baseline Algorithm: The Hidden Markov Model Tagger We can improve on the baseline algorithm by introducing a Hidden Markov Model (HMM) to overcome this limitation. An HMM is a generative Markov model in which the system being modelled is assumed to be a Markov process with unobservable ("hidden") states. In the context of POS tagging, the HMM functions as a probabilistic classifier that determines the most likely sequence of tags given a sentence. This approach takes into account the context in which the words appear, allowing for a more accurate determination of their parts of speech.

The HMM is capable of generating a probability distribution over possible sequences of labels, making it well-suited for modelling the inherent uncertainty present in natural language. By incorporating the HMM into the baseline algorithm, we can improve the accuracy of POS tagging, as it can more effectively disambiguate words with multiple parts of speech. The Markov Assumption An HMM operates based on the Markov assumption, which states that the current state depends only on the previous state and not on any states beyond that. In the context of POS tagging, the likelihood of a word having a certain part of speech depends on its immediate preceding context, as opposed to considering the entire sentence.

When applying the Markov assumption in HMMs, we consider a sequence of state variables, q1, q2, q3, up to qi. The Markov assumption implies that the probability of the next state, qi, given all the preceding states, q1 to qi-1, is equal to the probability of qi given just one preceding state, qi-1.

Formally, the Markov assumption can be stated as follows:

$P(qi \mid q1, q2, \ldots, qi\text{-}1) = P(qi \mid qi\text{-}1)$.

In simpler terms, this means that we only need the information from the present state to predict the future, and past information is not relevant for making predictions. By incorporating the Markov assumption in HMMs, these models can be more efficient in processing and analyzing sequences in natural language, as they do not need to consider all the past information to predict future states. A Markov Chain of Part of Speech Tags Given a sequence of observed words, the HMM allows us to compute the probability of a sequence of tags. However, since tags are not directly observable, they are considered hidden events. We can use the observed words to infer the hidden tag sequences by applying the Markov assumption.

In this context, a Markov chain of POS tags refers to a sequence of tags, where the probability of each tag is conditioned on the preceding tag. This sequence

can be represented as a probabilistic graphical model where each tag represents a node, and the edges between the nodes have probabilities associated with them. We call these transition probabilities. The transition probability is the probability of one tag following another tag, i.e. the probability of a transition from one tag to another. For example, the probability of a noun being followed by a verb.

However, HMM POS taggers also rely on another set of probabilities: emission probabilities. These are the probabilities that a given tag "emits" or "generates" a word (see below).

By including these two sets of probabilities and utilizing the Markov assumption, the HMM POS tagger becomes more effective and accurate, as it considers the contextual relationships between words and their parts of speech. This improves the model's ability to disambiguate between words with multiple part-of-speech tags.

The Transition Probability To compute the transition probability, which is the probability of tag i given tag i-1, we can utilize a tagged corpus of sentences. By counting the frequency of transitions from one tag to another, we can derive the probabilities of these transitions. The unique tags, including start and end tokens, form a table of transition probabilities. The table is read row, then column. Meaning that if you wanted to know the probability of a noun coming after a verb, you would go into the verb row and locate the noun column (answer is 5/5).

This can be visualized as a network, where different tags are represented as nodes, and all nodes are connected to each other with edges. The weight of an edge represents the probability of transitioning from one node (tag) to another. These edges are allowed to have zero weights, indicating that a particular transition does not occur in the corpus.

Emission Probabilities Emission probabilities refer to the likelihood of a certain POS tag generating a particular word. These probabilities help us understand the relationship between a tag and its corresponding word, which is observable in the text.

Consider an HMM where the hidden states represent POS tags, and the observations are the words themselves. The aim is to compute the probability of a tag sequence given the words in the text. To do this, we introduce the concept of emission probabilities, which connect the hidden states and observations.

Emission probabilities are computed by dividing the number of occurrences of a particular word generated by a given POS tag by the total number of occurrences of that tag. For example, to find the emission probability of the word "spot" being emitted by a noun (N) tag, we count how many times "spot" is tagged as a noun in the corpus and divide that by the total number of times the noun tag appears. Therefore, the probability of the word "spot" being emitted by a noun tag is 3/10.

As with transition probabilities, we can use our labelled corpus to calculate emission probabilities. In other words, we want to find the most likely estimates of these probabilities using the observed frequency of word-tag combinations in the corpus.

To create an emission probability table, we exclude the start and end tags since they don't generate words or observations. For each POS tag in the corpus, we count the number of times a specific word follows it and divide this count by the total number of times that tag appears. The result is the emission probability of the word given the POS tag.

By considering both transition and emission probabilities, we gain a better understanding of the context and grammatical structure of natural language, thus improving the accuracy and efficiency of NLP tasks. The Full Hidden Markov Model The full HMM consists of three components: observations, emission probabilities, and transition probabilities.

Observations: These represent the sequence of words in a sentence. The goal is to find the most likely sequence of hidden states or POS tags that generated these words. Emission probabilities: These probabilities reflect the likelihood of a word being emitted by a specific POS tag. They are calculated by dividing the number of times a word follows a POS tag by the total number of occurrences of that tag. Transition probabilities: These probabilities represent the likelihood of one POS tag following another. They are estimated by counting the frequency of one tag occurring immediately after another and dividing it by the total number of occurrences of the preceding tag.

By combining these three components, a full HMM can predict the most probable sequence of POS tags that generated a given sentence. This model is particularly useful in various NLP tasks, such as POS tagging and syntactic parsing, as it provides a mathematical framework for understanding natural language's context and grammatical structure. HMM Formalism In the HMM formalism, we have the following components:

States: Represented by the set Q, which contains a sequence of n states (q1, q2, . . . , qn) or POS tags (t) in the context of POS tagging. Transition probabilities: Represented by the matrix A, which contains the probabilities of transitioning from state qi to qj. In the context for the POS tagger: P(ti|ti-1). Observations: Represented by the sequence O, containing the sequence of t observations or words (w1, w2, . . . , wn) in the case of POS tagging. Observation probabilities: Represented by the sequence B, which contains the observation probabilities or emission probabilities for each state, representing the probability of an observation w being generated by a state q. In the context of the POS tagger: P(wi|ti). Initial probability distribution: Represented by the vector Π, which contains the initial probabilities the states Q. In the POS tagging case, Π is replaced with the probability of the start token transitioning to a POS tag.

By using this HMM formalism, we can model and predict the most probable sequence of POS tags that generated a given sentence. HMM for POS: As-

sumptions The goal is to determine the maximum probability of all possible tag sequences, given the sequence of words in the sentence.

To achieve this, we use Bayes's theorem, which states that the probability of a tag sequence given the word sequence is equal to the probability of the word sequence given the tag sequence multiplied by the probability of the tag sequence divided by the probability of the word sequence.

In practice, the word sequence remains constant (our word sequence is fixed), so we can simplify this equation.

We can then make two assumptions, which are commonly used in HMMs for NLP tasks:

Assumption 1: The probability of the word sequence given the tag sequence can be approximately calculated as a product of the probabilities of each word given its corresponding tag. This implies that each word depends only on its current tag and not on preceding or following tags.

Assumption 2: The probability of a tag sequence can be approximately calculated as a product of the probabilities of each tag given its previous tag. This is known as the Markov assumption, and it states that each tag depends only on its previous tag, not on any other preceding tags.

With these assumptions, we can simplify the problem and derive the most probable sequence of POS tags that generated the given sentence.

Finding The Most Probable Tag Sequence Finding the most probable POS tag sequence is a challenge, as the number of possible tag sequences can be enormous, making it infeasible to compare all sequences directly. For instance, assuming a nine-word sentence and 45 possible tags, there are $459 = 75{,}668{,}064{,}257{,}8125$ possible tag sequences. Further, assuming 9 bytes per tag sequence, approximately 6,810,125 GB of RAM would be needed.

In our previous example, we had the sentence "Liv will spot Spot" with three possible tags for each observation.

We first focus on removing zero probability transitions and emissions to identify the optimal tag sequence. This significantly reduces the number of sequences that need to be considered.

Additionally, we can eliminate incomplete sequences, further decreasing the number of possible sequences. By doing this, we can go from 81 possible sequences to only a few (4) sequences to consider.

Finally, among the remaining sequences, we can determine the most probable one by computing the products of the transition and emission probabilities. To do this, we use the transition probability table and emission probability table from before.

The Viterbi algorithm However, even after pruning, the number of possible tag sequences can be huge. Dynamic programming techniques, such as the Viterbi algorithm, address this issue.

With dynamic programming, we can focus on computing the probabilities for the most promising sequences and ignore those that are less likely. This substantially reduces the computational complexity and allows us to identify the optimal tag sequence without comparing every possible sequence. This reduces the computational burden significantly making it possible to find the most probable tag sequence within a reasonable timeframe and without requiring immense amounts of memory.

The Viterbi algorithm is a systematic approach to finding the most probable tag sequence for a given sentence. To apply the Viterbi algorithm, we have three steps:

Evaluate from left to right: For each observation (i.e., a word) from the first to the last, we compute the probability of reaching that word. For each word, we consider all possible tags (1 through n) and keep only the best path for each tag. At the end of the sentence, there will be only one best path remaining for each tag. We use these paths to identify the most probable tag sequence.

Let's demonstrate this with the sentence "Liv will spot Spot" and the pruned sequences. We start with the word 'Liv' and consider the only possible transition: a noun following the start token, with a probability of 4/5. Additionally, we consider the emission probability of 'Liv' given a noun, which is 2/10. Since there are no alternatives, this becomes the path up to the noun.

Next, we have the word 'will', which can be a noun or a modal verb. We compute the probability of the modal given a preceding noun and the probability of a noun given a preceding noun. To decide what next step is optimal, its probability is computed as the product of its transition and emission probabilities. In this case, the upper path probability is the probability of a noun following a noun (1/10) multiplied by the probability of a noun being will (1/10). And the probability of the lower path is the probability of a modal verb following a noun (4/10) times the probability of a modal verb being will (4/4). Therefore, the lower path is the optimal path

Following the Viterbi algorithm, we continue this process for each word and tag, ultimately identifying the most probable tag sequence for the given sentence. The crucial point is when we have more than one possible path to a tag, for instance, the noun or verb in the fourth position.

The two paths leading up to the tag share the same subsequent path (e.g. , N, N, N, N, and , N, M, N, N,). This latter (common) part of the path will have the same probability in both cases. Since the probability of a sequence is a product of probabilities, the less probable path leading up to the tag can be discarded, it will always be less probable.

The same holds for the two paths to verb in the fourth position (i.e. , N, N, V, N, and , N, M, V, N,).

By evaluating from left to right for each observation, or word, and considering all possible tags for each word, we can determine the optimal path for each tag.

At the end of the sentence, we are left with only one best path remaining for each tag.

This approach requires significantly fewer computations than considering all possible sequences (NT) and, instead, becomes a polynomial-time process (N2T). This reduced complexity makes it much more feasible to compute. Dynamic Programming The Viterbi algorithm is an example of dynamic programming, a concept introduced by Richard Bellman in 1952. Dynamic programming entails breaking down a problem into smaller subproblems and solving the optimal solution for each subproblem. In the case of a POS tagger, this approach allows us to identify the most probable tag sequence without having to compare all possible tag sequences. Improving The HMM POS Tagger Improving the HMM POS tagger involves incorporating additional context and optimizing the search process to enhance accuracy and speed. One way to improve accuracy is to use wider tag contexts, such as trigrams instead of bigrams. That is, approximate the probability of a tag sequence by using the probability of a tag given the previous two tags (i.e., P(ti | ti-1, ti-2). This allows the model to better estimate the probability of a tag sequence by considering the previous two tags as well.

Another issue is that as the number of tags increases, the Viterbi algorithm, which uses the dynamic programming approach, becomes slower. We can adopt beam search techniques to address this issue to improve the algorithm's efficiency. Beam search helps speed up the Viterbi algorithm by keeping only the best beta tags instead of all possible tags. This reduces the time and computation required to find the optimal tag sequence, making the process more manageable.

Uses of Parsing There are two kinds of parsing in NLP: dependency and constituency parsing. Dependency parsing focuses on understanding the grammatical structure of a sentence and is commonly used in machine translation, sentiment analysis, and information extraction. In constituency parsing, sentences are represented as parse trees, where each node corresponds to a phrase, and edges represent parent-child relationships. Constituency parsing is used in syntactic analysis, grammar checking, and text generation. Both dependency parsing and constituency parsing are applied in co-reference resolution, which identifies expressions, such as pronouns, that refer to the same entity.

Parsing is also used in sentiment analysis to analyze the syntactic structure of a sentence, identifying the sentiment-bearing parts and improving aspect-based sentiment analysis. In question answering, another NLP application, parsing aids in recognizing relevant entities and relationships needed to retrieve the correct answer. Finally, in information extraction, parsing can extract structured information from unstructured text, including subject-verb-object relationships that are crucial when extracting events or facts. When To Use Dependency Parsing? When it comes to dependency parsing, it is primarily utilized for enhancing the performance of downstream tasks. It serves as a supportive component rather than a standalone application. Prior to 2015, parsing was commonly in-

corporated into NLP pipelines. However, with the advent of end-to-end models, dependency parsing is now more commonly used to enrich text representation, for example, by tagging a text before vectorization or as an additional feature vector.

An effective dependency parsing model necessitates extensive training on large datasets. Consequently, a custom model trained on a smaller dataset can benefit from leveraging the valuable information gained from larger datasets. This can be seen as transferring knowledge from large datasets to smaller models, thus resembling transfer learning (although technically distinct). Parse Trees Language Is Recursive Consider a set of words, such as "the", "cat", "cuddly", "door", and "by". Each word has a corresponding POS: "the" is a determiner, "cat" is a noun, "cuddly" is an adjective, "door" is another noun, and "by" is a preposition. These words can be combined to form phrases with specific categories.

For instance, we can create the phrase "the cuddly cat" by combining the determinant "the", the adjective "cuddly," and the noun "cat" to form a noun phrase. Similarly, we can create another phrase, "by the door," by combining the preposition "by" with the noun "door." These two phrases can be combined further to create a larger noun phrase, such as "the cuddly cat by the door." This demonstrates the recursive nature of language as we can continue building larger and larger phrases using the same rule repeatedly.

Parse Trees Parse trees are a way to represent the grammatical structure of sentences by breaking them down into their constituent phrases. Consider the sentence, "I shot an elephant in my pajamas." At the first level, individual words have their respective parts of speech: "I" is a pronoun, "shot" is a verb, and "an" is a determiner. "Elephant" is a noun, and "in my pajamas" is a preposition followed by a noun phrase where "my" is a pronoun, and "pajamas" is a noun.

By combining these constituents, we can construct a parse tree: "I" creates a noun phrase, "shot an elephant" is a verb phrase consisting of a verb and the noun phrase "an elephant." This verb phrase can be combined with the preposition phrase "in my pajamas" to form another verb phrase, "shot an elephant in my pajamas." Finally, the noun phrase "I" can be combined with the verb phrase to create a complete sentence, "I shot an elephant in my pajamas." In this manner, we have parsed the sentence into a parse tree, representing its grammatical structure. Shallow Parsing Shallow parsing, also known as chunking, involves grouping adjacent tokens into phrases based on their POS tags. Well-known chunks are noun, verb and prepositional phrases.

Noun phrases consist of at least two words with a noun, pronoun, or determiner as the head and may include modifiers. They function as a noun in a sentence. Examples of the form modifier noun: the man a girl the doggy in the window.

arguments verb: She can smell the pizza He has appeared on screen as an actor.

Verb phrases, on the other hand, consist of at least two words with a verb as the head and arguments that further illustrate the verb's tense, action, or tone. Examples of verb phrases include "can smell" in the sentence "She can smell the pizza" and "has appeared" in the sentence "He has appeared on screen as an actor."

Shallow parsing allows us to break down sentences into noun and verb phrases to better understand their grammatical structure. Dependency Parsing Dependency Structure Dependency structure is a representation of how words in a sentence are related to one another, specifically, which words modify or act as arguments for other words. For example, in the sentence, "Look in the large crate in the kitchen by the door," we can identify a root word (e.g., "look"), words that depend on the root (e.g., "crate"), words in turn, that depend on those words (e.g. "door") and so on. This dependency structure highlights the relationships between words in a sentence and helps in understanding the sentence's grammatical structure. Why Do We Need Dependency Parsing? Dependency parsing is essential in NLP as it helps to understand the structure of sentences and the relationships between words. Sentence structure plays a crucial role in conveying complex meanings through language and, therefore, is vital for many NLP tasks. Knowing how words are connected and function in relation to one another is the core concept of dependency parsing.

Understanding sentence structure is important for interpreting language accurately and avoiding ambiguity. Correct parsing can sometimes prevent misinterpretation, as shown in the example headline, "San Jose Cops kill man with knife." This headline is ambiguous and can be interpreted in two ways: the man who was killed had a knife or the cops used a knife to lethally stab a man. Another example is "scientists count whales from space," which can be misinterpreted as whales coming from space or scientists counting them from space using satellite images. Dependency Grammar In the field of NLP, dependency grammar is a core concept that focuses on the relationships between words in a sentence, often represented as binary asymmetric connections called dependencies (aka arrows). These relationships are typically labelled by grammatical roles like subject, determiner, prepositional object, and others. Dependency grammar has a long history, with roots tracing back to Pāṇini, a logician, Sanskrit philologist, grammarian, and revered scholar in ancient India. He lived sometime between the 6th and 4th century BCE and wrote his texts on birch bark.

The earliest known NLP dependency parser was developed in 1962 by David Hays.

Universal dependency relations are defined on resources such as the Universal Dependencies website and the Stanford Typed Dependencies manual. Some common grammatical relations in dependency parsing are shown in the table below. For complete lists consult Universal Dependencies or the Stanford Typed Dependencies manual.

Causal argument relation Description NSUBJ Nominal subject DOBJ Direct object IOBJ Indirect object CCOMP Clausal complement Nominal modifier relations Description NMOD Nominal modifier AMOD Adjectival modifier NUMMOD Numeric modifier APPOS Appositional modifier DET Determiner CASE Prepositions, postpositions & other case markers Other notable relations Description CONJ Conjunction CC Coordinating conjunction

Relations are represented by abbreviations for easy identification and understanding. Building a Dependency Parser To build a dependency parser, we need data annotated for supervised learning, for example, from existing treebanks. One valuable resource for dependency parsing is the Universal Dependencies database, which contains sentences labelled with dependencies and their types across more than 100 languages.

Using a pre-existing data set like Universal Dependencies provides several advantages. First, it saves us the effort of labelling the data ourselves. Second, it can be used for multiple parsers, allowing for reusability of labour. Third, it provides broad and systematic coverage of real-world language use, making it useful for statistical analysis and linguistic research. Lastly, it serves as a basis for evaluating parsers by benchmarking performance on a standardized data set.

By using the Universal Dependencies database or similar resources, we can effectively train our dependency parsers and measure their accuracy compared to other models. This aids in continuous improvement and ensures our parsers are performing optimally on real-world language data. The Basics of a Dependency Parser Dependency parsing involves analyzing sentences and determining the relationships between words, often represented as a tree, aka a directed acyclic graph (DAG). To create a dependency structure, we decide for each word which other words it is dependent on, following certain constraints. Only one word can be independent of the root, ensuring a clear hierarchical structure. There must be no cycles within the dependency graph, meaning that if B is dependent on A, A cannot be dependent on B. If a dependency structure meets these constraints, it forms a tree without any cycles, i.e. a DAG. A more optional constraint is that arrows cannot cross. In these cases, the dependency tree is referred to as non-projective. Note that the directions of the arrows indicate dependency. Arrows beginning at A and ending at B means that B depends on A.

Types of dependency parsers There are various methods for dependency parsing, which can be broadly categorized into four groups.

Dynamic programming: Eisner (1996) gives a clever algorithm with complexity O(n3), by producing parse items with heads at the ends rather than in the middle. Graph algorithms: In this category, minimum spanning trees are built for sentences. An example of this method is McDonald and colleagues (2005) MSTParser which scores dependencies independently using an ML classifier. Constraint satisfaction: Here, edges that don't satisfy hard constraints are eliminated. An example of such a dependency parser is one developed by Karlsson in 1990. Transition-based parsing: This type of parser selects depen-

dencies greedily guided by ML classifiers. For example, the MaltParser by Nivre and others (2008) is a popular and effective transition-based parser.

Now, let's focus on transition-based parsing, an efficient method for building dependency parsers in NLP. Transition-based Parsing Transition-based parsing, introduced by Jochem Nivre in 2008, is a simple and greedy discriminative dependency parser. This parsing method involves a sequence of actions performed by the parser to analyze the input sequence. The parser consists of four components: A stack initially containing only the root node. A buffer initially containing the input sequence. A set of dependency arcs (A) which is initially empty. A set of actions available to the parser. The Arc-standard Parser In this simplified model, the parser has three available actions: Left_arc: Asserts a head-dependent relationship between the word at the top of the stack and the word directly beneath it, removing the lower word from the stack. Right_arc: Asserts a head-dependent relationship between the second word of the stack and the word at the top, removing the word at the top. Shift: Removes the word from the front of the input buffer and puts it onto the stack. How to Select The Action In a transition-based dependency parser, the correct action is chosen using an "oracle." The oracle is often an ML classifier trained to predict the actions needed in a particular scenario. In the arc-standard, the three actions are left_arc, right_arc, and shift.

Lycurgus Consulting the oracle in Delphi (Pythia). Image credits: By Eugène Delacroix, Public Domain, https://commons.wikimedia.org/w/index.php?curid=29190439.

The inputs to the oracle are features derived from the sentence being parsed. The features are often created through feature engineering or, more recently, with deep learning methods and pre-trained language models such as BERT or RoBERTa. Earlier approaches involved more manually created features, while modern parsers take advantage of powerful neural networks and embeddings to represent linguistic phenomena. For instance, the early MaltParser employed sparse, hand-coded features, while its successor, Chen and Manning's parser from 2014, relied on deep learning-based embeddings as features. Nowadays, state-of-the-art parsers use advanced deep learning techniques, such as transformer networks, to achieve superior performance in parsing and understanding natural language.

Evaluating a Dependency Parser Dependency parsers are evaluated using two scores, the unlabelled attachment score (UAS) and the labeled attachment score (LAS). Both rely on a labelled dataset for their calculation.

Unlabelled Attachment Score Evaluating a dependency parser involves the use of the UAS. In this case, the focus is on the structure of the dependency tree and not the labels associated with the arrows. The evaluation process starts by having ground truth data, which consists of a sentence with its corresponding dependency arrows. Next, we have the dependency parser generate a predicted set of arrows for the same sentence.

To calculate the unlabeled attachment score, we count the number of correctly predicted dependencies and divide it by the total number of dependencies in the ground truth data. The unlabeled attachment score approach bears similarities to normal classification accuracy in evaluating the performance of the dependency parser. Labelled attachment score Evaluating a dependency parser using the LAS involves not only considering the dependency arrows but also the specific type of dependency present, such as a subject, root, determiner, or causal complement. To assess the performance of the parser, we start by having ground truth data that includes both the dependency arrows and the corresponding types.

We then compare this ground truth to the dependency parser's predictions, counting the number of correctly identified dependencies – both in terms of the arrow direction and the type of dependency. Finally, we divide this number by the total dependencies in the ground truth data to obtain the labelled attachment score. This metric provides a more comprehensive assessment of the parser's accuracy in predicting both the dependencies and their respective types. State of The Art State-of-the-art dependency parsers have achieved impressive results in recent years, as evidenced by the Penn Treebank task on the NLP Progress website. As of 2023, the best dependency parser attained a UAS of 97.42% and a LAS of 96.26%, approaching the ceiling performance.

These top-performing parsers typically feature large, deep neural networks and have undergone significant hyperparameter tuning to optimize their performance. For example, the current SOTA (2023-10-23) uses XLNet a transformer-based neural network architecture. In addition, instead of relying on greedy search for action selection, these advanced parsers leverage beam search. Furthermore, many of them integrate self-attention mechanisms within their networks, a technique that will be discussed in further detail in upcoming lectures.

8.2 Terminology

- Hidden Markov Model (HMM)
- Part-of-speech (POS)
- Dynamic Programming
- The Viterbi Algorithm
- Parsing
- Constituency parsing
- Dependency parsing
- Parse tree

- Shallow parsing
- Chunking
- Noun phrase
- Verb phrase
- Dependency grammar
- Dependencies
- Grammatical role
- Dependency parser
- Transition-based parser
- The arc-standard parser
- Oracle
- LAS
- UAS

Chapter 9

Named Entity Recognition

Written by Hjalmar K Turesson, HuggingChat (Llama-2-70b-chat-hf)

9.1 Introduction

9.1.1 Contents

- Introduction
  - Definition of Named-entity Recognition (NER)
  - Importance of NER in Natural Language Processing (NLP)
  - Relationship between NER and Information Extraction
- Background
  - Overview of NER tasks
  - Types of named entities (e.g., person, organization, location, date, time, etc.)
  - Challenges in NER (e.g., ambiguity, context dependence, lack of context)
- Fitting a NER Model
  - Approaches to NER (rule-based, machine learning, hybrid)
  - Features used in NER (lexical, syntactic, semantic)
  - Label encoding techniques (IO and IOB encoding)
  - Model selection and evaluation metrics (precision, recall, F1 score)
- Sequence Labeling
  - Introduction to sequence labeling
  - NER as a sequence labeling task

  - Implications for feature extraction and model training
- Evaluation of a NER Model:
  - Evaluation metrics (e.g., precision, recall, F1 score)
  - Test datasets and their characteristics
  - Results interpretation and comparison
- Conclusion
  - Summary of key points
  - Use cases and applications of NER
  - Challenges and future directions in NER research
- Named Entity Linking
  - Definition
  - Task
  - Importance
  - Challenges

9.2 Introduction

9.2.1 Proper Names and Named Entities

Proper Names: * *Definition*: Proper names are a linguistic category that refers to specific, unique individuals, places, organizations, or objects. They are often composed of multiple words and are capitalized in written language to distinguish them from common nouns. Examples include "Marie Curie," "Sheffield University," and "Apple Inc." * *Importance*: Proper names play a crucial role in natural language understanding (NLU), as they provide essential information about people, places, organizations, and objects mentioned in texts.

Named Entities: * *Definition*: Named entities are a broader category than proper names, encompassing not only names of people, places, and organizations but also dates, times, percentages, and other numerical values that can be referred to using a proper name. Examples include "January 1st," "30%," and "the European Union." * *Importance*: Identifying named entities is critical for various NLP tasks, such as question answering, information extraction, and sentiment analysis. Understanding whether a named entity refers to a person, place, organization, or another type of entity helps machines comprehend the meaning and context of a text.

9.3 Named-entity Recognition

NER is a task in NLP that involves identifying and categorizing named entities in text into predefined categories such as person, location, organization, date,

time, money, etc. NER is also known as entity identification, entity chunking, and entity extraction. It is a special case of information extraction and sequence labelling, where the goal is to assign a label to each word or phrase in a sentence corresponding to a particular entity type. NER is important in various NLP applications such as information retrieval, question answering, text summarization, machine translation, and sentiment analysis.

9.3.1 Task Description

Given a sentence or paragraph of text, the task is to identify and tag each named entity with its corresponding entity type. Common entity types include person (PER), location (LOC), organization (ORG), geopolitical entity (GPE), date (DATE), time (TIME), money (MONEY), etc. In addition to standard entity types, custom task-specific tags can be created to capture specific entities relevant to a particular application or domain.

9.3.1.1 Examples

- "John Smith" (PER)
- "New York City" (LOC)
- "Google" (ORG)
- "2022" (DATE)
- "14:30" (TIME)
- "$100" (MONEY)

9.3.1.2 Common NER tags

PER (People): This tag is used to identify the names of people, including fictional characters. *Example sentence*: "Turing is a giant of computer science." *Tagged sentence*: "Turing" (PER)

ORG (Organizations): This tag is used to identify names of organizations, including companies, sports teams, and government agencies. *Example sentence*: "The IPCC warned about the cyclone." *Tagged sentence*: "IPCC" (ORG)

LOC (Locations): This tag is used to identify names of locations, including cities, countries, and landmarks. *Example sentence*: "I'm going to visit Tokyo next week." *Tagged sentence*: "Tokyo" (LOC)

DATE (Dates): This tag is used to identify dates and times mentioned in the text. *Example sentence*: "The meeting is scheduled for March 20th at 2 PM." *Tagged sentence*: "March 20th" (DATE), "2 PM" (TIME)

TIME (Times): This tag is used to identify time mentions without a date. *Example sentence*: "I'll meet you at 5 o'clock." *Tagged sentence*: "5 o'clock" (TIME)

MONEY (Monetary amounts): This tag is used to identify the amounts of money mentioned in the text. *Example sentence*: "The new laptop costs $1,500." *Tagged sentence*: "$1,500" (MONEY)

PRODUCT (Products): This tag is used to identify the names of products, including books, movies, and software. *Example sentence*: "I hate my new iPhone." *Tagged sentence*: "iPhone" (PRODUCT)

EVENT (Events): This tag is used to identify the names of events, including concerts, festivals, and conferences. *Example sentence*: "I'm attending the NLP conference next month." *Tagged sentence*: "NLP conference" (EVENT)

WORK_OF_ART (Works of Art): This tag is used to identify the names of works of art, including books, paintings, and songs. *Example sentence*: "My favourite sculpture is the 'Spirit of Haida Gwaii'." *Tagged sentence*: " 'Spirit of Haida Gwaii' " (WORK_OF_ART)

URL (Uniform Resource Locators): This tag is used to identify URLs mentioned in text. *Example sentence*: "Check out my website at nlp.com." *Tagged sentence*: "nlp.com" (URL)

In summary, these are some of the most common NER tags used in NLP tasks. However, custom NER taggers can also be built that can assign custom tags. By accurately identifying and categorizing named entities in text, NER enables various applications such as information retrieval, question answering, and sentiment analysis.

9.3.2 NER Tagging - Find and Classify Named Entities

The goal of NER is to automatically extract and classify relevant information from texts, enabling various applications such as information retrieval, question answering, and sentiment analysis.

The process of NER typically consists of two steps:

1. Named Entity Detection (NED): Identifying named entities in the text.
2. Named Entity Classification (NEC): Classifying detected named entities into their respective categories.

Example:

"Citing high fuel prices, United Airlines said Friday it has increased fares by $6 per round trip on flights to some cities also served by lower-cost carriers. American Airlines, a unit of AMR Corp., immediately matched the move, spokesman Tim Wagner said. United, a unit of UAL Corp., said the increase took effect Thursday & applies to most routes where it competes against discount carriers, such as Chicago to Dallas & Denver to San Francisco." (Example from section 8.3 in Speech & Language Processing by Jurafsky & Martin.)

In the given example, the named entities detected are:

- United Airlines
- Friday
- $6
- American Airlines
- AMR Corp
- Tim Wagner
- United
- UAL Corp
- Thursday
- Chicago
- Dallas
- Denver
- San Francisco

These named entities can be classified into the following categories: * **Organization**: United Airlines, American Airlines, AMR Corp, UAL Corp * **Time**: Friday, Thursday * **Money**: $6 * **Person**: Tim Wagner * **Location**: Chicago, Dallas, Denver, San Francisco

Once the named entities are detected and classified, they can be linked to their corresponding entries in a knowledge base or database, enabling further processing and analysis of the text data.

9.3.3 Information Extraction

Information extraction is a subfield of NLP that focuses on extracting structured data or information from unstructured text. The goal of information extraction is to automatically retrieve relevant information from text in a format that can be easily processed and analyzed.

9.3.3.1 Types of Information Extraction

- **Entity extraction**: This involves extracting named entities such as people, organizations, and locations from text. For example, extracting the name "Schulich School of Business" and its location "Toronto, Ontario, Canada" from a sentence.
- **Relation extraction**: This involves extracting relationships between entities, such as "Person A is the CEO of Company B."
- **Event extraction**: This involves extracting events and their corresponding arguments, such as "Event: John Smith spoke at the conference" and "Argument: John Smith is the speaker."
- **Fact extraction**: This involves extracting factual information from text, such as extracting the address of a business or the price of a product.

9.3.3.2 Applications of Information Extraction

- **Search engines**: Information extraction is used by search engines like Google to extract addresses, phone numbers, and other relevant information from web pages and present them directly to users.
- **Email clients**: Gmail, for instance, uses information extraction to automatically extract event details from emails and add them to the user's calendar.
- **Medical research**: Information extraction can be applied to medical research literature to extract relevant information about drug interactions, side effects, and other medical facts.
- **Business intelligence**: Information extraction can extract financial data from company reports and news articles, such as earnings and profits.
- **Social media monitoring**: Information extraction can be used to monitor social media posts and extract relevant information about customer sentiments, opinions, and preferences.

9.3.3.3 Techniques for Information Extraction

- **Rule-based approaches**: These involve defining rules to extract specific information from text based on patterns and regular expressions.
- **Machine learning approaches**: These involve training machine learning (ML) models on labelled datasets to learn patterns and relationships in text and extract information accordingly.
- **Hybrid approaches**: These combine rule-based and ML techniques to improve the accuracy and efficiency of information extraction systems.

9.3.3.4 Challenges in Information Extraction

- **Ambiguity**: Words or phrases can have multiple meanings, making determining the correct interpretation in a given context challenging.
- **Noise**: Text can contain irrelevant or erroneous information, which can negatively impact the accuracy of information extraction systems.
- **Variability**: Information can be presented in different formats and structures, requiring flexible and adaptable information extraction systems.

9.3.3.5 Conclusion

Information extraction is a powerful tool for automating the process of extracting relevant information from text. Its applications in various fields, including search engines, email clients, medical research, business intelligence, and social media monitoring, demonstrate its potential to improve efficiency and decision-making. However, information extraction poses significant challenges, such as

ambiguity, noise, and variability, which must be addressed through advanced techniques and technologies.

9.3.4 Fitting a NER model

9.3.4.1 Label encoding

When training a NER tagger, the first step is to preprocess the text data and assign appropriate labels or tags to each word or phrase. The choice of encoding strategy in NER can impact the model's accuracy and the learning problem's complexity. Two commonly used encoding strategies are Inside-Outside (IO) and Inside-Outside-Beginning (IOB).

9.3.4.1.1 Inside-Outside Encoding With IO encoding, each word in the text is assigned a binary label indicating whether it is part of a named entity (inside) or not (outside). The number of labels depends on the number of NER tags. For example, if we have three tags (person, location, time), we will need three separate labels for each tag and one for outside, resulting in a total of $3 + 1 = 4$ labels.

For instance, in the sentence "Alice showed Bob Charles Darwin's new book," the words "Alice", "Bob", "Charles" and "Darwin" would receive the label "PER" (person) since they are both all, while "showed", "'s", "new" and "book" would receive the label "O" (outside) since they refer to non-entity concepts. "Alice [PER] showed [O] Bob [PER] Charles [PER] Darwin [PER] 's [O] new [O] book [O]"

IO encoding assigns a binary label to each word in the text, indicating whether it is part of a named entity (inside) or not (outside). With this approach, we can represent named entities without losing any information. However, the downside is that we require a larger number of labels, as each tag has its own dedicated label.

9.3.4.1.2 Inside-Outside-Beginning Encoding IOB encoding builds upon the IO encoding scheme by introducing a third label, "B" (beginning), which indicates the starting point of a named entity. This approach allows us to differentiate between adjacent named entities of the same type and capture multi-word named entities more accurately. However, the number of labels required increases dramatically, as each tag now has two dedicated labels (beginning and inside). For example, with three tags, we will need $2 \times 3 + 1 = 7$ labels.

Using our previous example, "Alice", "Bob" and "Charles" would receive the label "B-PER" (beginning person) since they mark the start of new entities, while "Darwin" would get "I-PER" (inside person) since it is part of the same

entity as "Charles". This allows us to differentiate between adjacent named entities of the same type, which is impossible with IO encoding.

"Alice [B-PER] showed [O] Bob [B-PER] Charles [B-PER] Darwin [I-PER] 's [O] new [O] book [O]"

9.3.4.1.3 Comparison of Encoding Strategies While IOB encoding provides a more accurate representation of named entities, the increased number of labels makes the learning problem harder and requires more data. On the other hand, IO encoding requires fewer labels but may lose some information. The choice of encoding strategy depends on the specific use case and the available resources.

9.3.4.2 Features for Sequence Labelling

Several types of features can be used when training a sequence labelling model. Some common features include:

- **Word context**: This feature uses the current word and its surrounding words to provide context for the labelling decision. For example, in the phrase "Schulich School of Business," the word context for the word "of" might include the words "Schulich", "School" and "Business."
- **Other predicted tags**: This feature utilizes tags, like POS tags, to inform the labelling decision.
- **Label context**: This feature considers the already assigned labels when determining the correct label for the current word. Continuing with our example, the fact that "Schulich," "School," and "Business" have been labelled as organizations could influence the label assignment for "of."
- **Word shape**: see below
- **Word substrings**: see below

9.3.4.2.1 Featureizing Words: Word Shape Some words may be rare, providing limited information for the NLP task. To address this issue, words can be featureized by mapping them to simpler representations that capture their attributes like length, capitalization, numerals, internal punctuation, and Greek letters. This approach helps encode words with useful information, even when they are infrequent. Three examples illustrate this technique:

- "Varicella-zoster": word shape: Xx-xxx
- "mRNA": word shape: xXXX
- "CPA1": word shape: XXXd

The featureization process involves considering the first and last two characters of a word as the most important. The method replaces middle letters with unique codes in a specific order:

- Capital letters are replaced with X
- Lowercase letters are replaced with x
- Integer numbers are replaced with d
- Dashes, colons, periods, etc. remain unchanged

By featureizing words in this manner, we can better represent rare words and improve the accuracy of sequence labelling models.

9.3.4.2.2 Featureizing Words: Word Substring In addition to using whole words as features for sequence labelling, another approach is to extract meaningful substrings from words. These substrings can provide valuable information about the type of word or its context. Two examples of useful substrings are:

- "oxa" is common in drug names, such as "Cotrimoxazole" and "oxacillin." Replacing the whole word with OXA creates a feature that indicates it's a drug name.
- Colon (:) is useful in movie and person names. For instance, "Alien Fury: Countdown to Invasion" and "Mad Max: Fury Road".
- "field" has pretty good precision for person names. For example, Wethersfield, Mansfield and Hadfield.

This technique is particularly helpful when dealing with scarce data where preventing rare words from being treated as unknown is crucial.

9.3.4.2.3 More Examples of Sequence Labelling Tasks These are various sequence labelling tasks in NLP, where the goal is to assign a label or tag to each element in a sequence of text. Each task requires a different set of labels or tags, depending on the specific application and the type of information being extracted.

- **Part-of-Speech Tagging**: Assign a part of speech (such as noun, verb, adjective, etc.) to each word in a sentence. *Example*: "The quick brown fox jumps over the lazy dog." *Tagged*: "The" (determiner), "quick" (adjective), "brown" (adjective), "fox" (noun), "jumps" (verb), "over" (preposition), "the" (determiner), "lazy" (adjective), "dog" (noun)
- **Named Entity Recognition**: Identify named entities (such as people, organizations, locations) in a sentence. *Example*: "John Smith is the CEO of ABC Corporation located in New York City." *Labeled*: "John Smith" (person), "ABC Corporation" (organization), "New York City" (location)
- **Text Segmentation**: Divide a running text into smaller segments, such as questions and answers. *Example*: "What is your name? My name is Juanita. Where are you from? I am from Toronto." *Segmented*: Question: "What is your name?" Answer: "My name is Juanita." Question: "Where are you from?" Answer: "I am from Toronto."

- **Word Segmentation**: Split words that are not separated by spaces, such as contractions or possessives. *Examples*: "aren't" (are + not), "isn't" (is + not), "we're" (we + are) *Labeled*: "aren't" (two words), "isn't" (two words), "we're" (two words)

9.4 Evaluation

Once we have built a named entity recognizer, it is essential to evaluate its performance to determine its accuracy and effectiveness. To do this, we need to predict the entities in a given text and compare them to the actual entities present in the text.

For instance, consider the following text: "The Schulich School of Business is the business school of York University." This text has two entities - "Schulich School of Business" and "York University". Our named entity recognizer should identify these entities correctly and assign them the appropriate labels.

Text	True	Predicted
The	- 0	- 0
Schulich	- ORG	- ORG
School	- ORG	- ORG
of	- ORG	- ORG
Business	- ORG	- ORG
is	- 0	- 0
the	- 0	- 0
business	- 0	- ORG
school	- 0	- ORG
of	- 0	- 0
York	- ORG	- ORG
University	- ORG	- ORG

To evaluate the performance of our recognizer, we can create a truth table that shows the true positive entities (correctly identified entities), false positive entities (incorrectly identified entities), true negative entities (entities that were not detected), and false negative entities (entities that were missed).

	Pos	Neg
Pos	2	1
Neg	0	4

From this table, we can calculate precision, recall, and F1 score, which provide a comprehensive assessment of the recognizer's performance.

$precision = \frac{tp}{tp+fp} = \frac{2}{3}$

Precision refers to the ratio of true positive entities to all positive predictions made by the recognizer. It measures the accuracy of the recognizer in identifying the correct entities.

$recall = \frac{tp}{tp+fn} = \frac{2}{2} = 1$

Recall, on the other hand, represents the proportion of true positive entities among all actual positive entities in the text. It evaluates the completeness of the recognizer in detecting all relevant entities.

$F_1 = 2\frac{precision \times recall}{precision+recall} = 2\frac{2/3 \times 1}{2/3+1} = 0.8$

Finally, the F1 score is the harmonic mean of precision and recall, providing a balanced measure of both factors.

By using these metrics, we can determine the strengths and weaknesses of our named entity recognizer and make necessary improvements to enhance its performance.

9.4.1 Limitations of F1 Score in NER

While the F1 score is a widely used metric to evaluate the performance of named entity recognizers, it has some limitations. One of the main issues is the problem of boundary errors.

Consider the sentence "First Bank of Chicago announced earnings. . . " where the entity's name is "First Bank of Chicago." If the recognizer mistakenly predicts "First" to be outside and only "Bank of Chicago" to be an entity, it results in two errors, a false negative and a false positive. The false negative is because the true entity spans from word one to word four, but the prediction only covers words two to four, resulting in a miss. The false positive is due to predicting an entity covering words two to four, where there isn't any.

In such cases, it's better for the recognizer to predict zero or outside for the entire phrase rather than making a partial match. Predicting zero would result in a higher F1 score, however this would then be implying that a complete miss is better than a partial match.However, the F1 score doesn't account for this and considers a partial match to be better than a complete miss. Therefore, the F1 score may not accurately reflect the performance of the recognizer in certain situations.

Despite these limitations, the F1 score remains a commonly used metric in NLP for evaluating NER taggers. It provides a useful way to balance precision and recall, but it shouldn't be the sole evaluation metric, especially when dealing with complex or nuanced naming conventions.

9.5 NER: Conclusion

9.5.1 Applications of NER

- **Aspect-Based Sentiment Analysis**: NER is used to identify the entities towards which sentiments are expressed in text data.
- **Question Answering**: NER helps identify entities mentioned in questions to provide accurate answers.
- **Web Page Tagging**: NER automatically generates links to relevant pages containing information about entities mentioned in web pages.
- **Event Detection**: NER is used to identify entities involved in events mentioned in text data.

9.5.2 Challenges

Important entities often change rapidly, such as companies changing products or politicians changing positions, requiring frequent updates in NER models. Due to this rapid change in entities, it is often essential to quickly train new NER models to keep up with the changes.

9.6 Entity Linking

- **Definition**: Entity Linking is the process of associating a mention in text with its corresponding representation in a knowledge base or database. Also known as Named Entity Disambiguation, Named Entity Normalization, or Named Entity Recognition and Disambiguation.
- **Task**: Associate named entities in text with their corresponding entries in a knowledge base or database, such as Wikipedia pages or database entries.
- **Importance**: Essential in the Semantic Web, which aims to make the web machine-readable. Useful in Information Retrieval, Content Analysis, Intelligent Tagging, Question Answering Systems, and Recommendation Systems.
- **Challenges**: *Named Variation*: Entities can be referred to in different ways, e.g., Michael Jordan, MJ, or Jordan. *Ambiguity*: A name can refer to multiple entities, e.g., Emerson - an Australian tennis player, an American writer, or a Brazilian footballer.

Chapter 10

Embeddings and Vector Semantics

Written by Hjalmar K Turesson and Stable Beluga 2

10.1 Introduction

In this ninth lecture of our NLP course – embeddings and vector semantics –, we will explore the concepts of distributed and localized representations, as well as delve into the intricacies of word meanings. We will primarily focus on word vectors, word embeddings, and text embeddings. By the end of this session, you will have a deeper understanding of these concepts, and we will also provide a tutorial to help reinforce your knowledge.

10.1.1 Contents

10.2 Representations

10.2.1 Local Versus Distributed Representations

Local representations assign one neuron or unit per object (or feature or object), making them simple and easy to hand-code. However, they are inefficient for data or objects with componential structure, which have multiple properties, components, and sub-objects.

On the other hand, distributed representations utilize many neurons or units to represent each object, with each unit participating in the representation of multiple objects. This creates a many-to-many relationship between representation

elements and the objects being represented. While difficult to hand-code, distributed representations are well-suited for representing componential objects and are believed to be the representation scheme of our brains, specifically the neural cortex.

10.2.2 Representing Words

In traditional NLP, words are often represented as a string of characters or an index into a vocabulary list, which are examples of local representations. However, in embeddings, the meaning of a word or sentence is represented as a dense vector, exemplifying a distributed representation.

Before delving into embeddings, it is essential to understand the meaning of words. Words are like componential objects, as they possess multiple meanings and can be viewed as having a confidential structure. In the remainder of this lecture, we will focus on exploring the meaning of words and how they are represented using distributed representations such as embeddings.

10.3 Meanings of Words

10.3.1 Words Can Carry Many Types of Meanings

Words can convey various meanings, which is essential to understand in order to develop efficient language models. One such concept is polysemy, where a single word possesses multiple meanings. For example, the word “bank” can refer to a financial institution or the land alongside a river. Synonymy, on the other hand, involves multiple words that share the same meaning, such as “fast” and “quick.”

Similarity, another form of word meaning, allows for connections between words with closely related meanings. For instance, the words “happy” and “joyful” are considered similar as they both convey positive emotions. Relatedness and association refer to the relationships between words based on shared properties or context. The words “dog” and “cat” may be considered related as they are both domestic animals, despite having different meanings.

Semantic frames and roles involve understanding the underlying structure and function of words within a sentence. The term “frame” encompasses the general scenario or situation that a sentence describes, while “roles” are specific elements of the scenario, such as the subject, verb, and object.

Connotations and sentiment are also important aspects of word meaning. Connotations refer to the emotional or cultural associations attached to words, whereas sentiment refers to the positive, negative, or neutral emotions associated with a word or phrase. For instance, the word “lovely” has positive connotations and carries a sentiment of happiness.

In summary, understanding the different types of word meaning is crucial in developing accurate NLP models. By recognizing polysemy, synonymy, similarity, relatedness, semantic frame and roles, connotations, and sentiment, we can better capture the complexity and subtlety of human language.

10.3.2 Word similarity

Word similarity is an important aspect in NLP, which focuses on understanding how words are related in terms of meaning and association. Words that are not synonyms but have some degree of relatedness can be considered similar. This concept becomes particularly relevant when comparing the similarity of phrases and sentences.

To help quantify word similarity, researchers have developed datasets such as SimLex 999, which contains pairs of words along with a human-judged similarity score on a scale from 0 to 10. For instance, the words "vanish" and "disappear" are highly similar and receive a score of 9.8. On the other end of the spectrum, words like "whole" and "agreement" are quite dissimilar and earn a score of 0.3. In between these extremes, we have word pairs like "belief" and "impression," which are somewhat similar and have a score of 5.95.

By utilizing datasets like SimLex 999, NLP models can better understand the subtleties of human language and capture the nuances of word meaning. This, in turn, leads to more accurate and effective language processing systems.

10.3.3 Word Relatedness

Word relatedness, also known as association, refers to the relationship between words based on the events they participate in or their shared properties. For instance, "coffee" and "cup" are related because they are both part of drinking coffee, and "surgeon" and "scalpel" are related as they are both essential components of performing surgery.

Semantic fields, sets of words belonging to the same semantic domain or area, also play a crucial role in understanding word relations. For example, words like "waiter," "menu," "plate," "food," and "chef" belong to the semantic field of restaurants. This concept is closely related to topic modelling and its applications in natural language processing (NLP), such as Latent Dirichlet Allocation (LDA), which can help in learning semantic fields based on data.

Another type of word relation is antonymy, which involves words with opposite meanings. Examples include "hot" and "cold," "dark" and "light," and "far" and "near."

Meronomy, on the other hand, refers to the part-whole relationship between words. In this case, a "finger" is a part of a "hand," a "wheel" is a part of a "bike," and a "spoke" can be considered a part of a "wheel."

Understanding different types of word relations, such as similarity, association, semantic fields, antonymy, and meronomy, is vital in developing accurate and efficient NLP models that can effectively capture the subtleties of human language.

10.3.4 Semantic Frames and Roles

Semantic frames and roles refer to a set of words that denote different perspectives or participants involved in a specific type of event. For instance, in a commercial transaction where someone is selling an item to a buyer, the event would be the transaction and the semantic roles may include buyer, seller, goods, and money.

In the example "Sam bought the book from Ling," we can paraphrase it as "The book was sold to Sam by Ling." The same transaction is being referred to but from different perspectives. This concept is important in understanding word meaning and the relationship between words.

10.3.5 Connotation

Connotation refers to the emotional or cultural associations attached to specific words. This concept is particularly relevant when discussing the effective meaning of words, which involves their sentiment, positive or negative evaluation. Sentiment words, such as "great" and "love" versus "terrible" and "hate," carry connotations that influence the emotions associated with the words.

In 1957, Osgood et al. presented a model describing three dimensions of variation in connotation: valence, arousal, and dominance. Valence addresses the pleasantness of the stimulus, with examples being happiness and satisfaction versus unhappiness and annoyance. Arousal refers to the intensity of the emotion provoked by the stimulus, such as excitement versus relaxation. Finally, dominance represents the degree of control exerted by the stimulus, with examples including important and controlling versus odd and influenced.

Using this concept, words can be represented as three-dimensional vectors in the connotation space. For instance, the word "courageous" can have a high valence (8.05), moderate arousal (5.5), and high dominance (7.38). Similarly, a bear cub could have a high valence (8.5), low arousal (3.8), and low dominance (2.4). By understanding and quantifying these connotative aspects, NLP models can better represent and comprehend the nuanced meanings of words and language.

10.3.6 Conclusion

In this discussion, we have explored various aspects of word meaning and the complex relationships between words. It is evident that words possess multiple

dimensions of meaning, which makes it essential for NLP models to represent them adequately.

To achieve this, we must consider the different types of word relationships, such as similarity, association, semantic fields, antonymy, and meronomy. Additionally, connotations, particularly the three dimensions of variation (valence, arousal, and dominance), are crucial in understanding the richness of language and capturing the complexity of meaning.

To efficiently represent words for machine learning and NLP tasks like question answering, summarization, plagiarism detection, and dialogue generation, we need a vector representation that encapsulates the diverse forms of meaning associated with words. Vector representations that include multiple dimensions, such as the three-dimensional connotation vectors discussed earlier, may provide a more accurate and comprehensive portrayal of word meaning. By developing such representations, NLP systems can better handle language processing tasks and understand the subtleties of human language. One-hot encoding is not a sufficiently rich representation.

10.4 Word Vectors

10.4.1 The Distributional Hypothesis

The distributional hypothesis, formulated in the 1950s, states that the meaning of a word is derived from its usage within a language. As Wittgenstein famously said, "The meaning of a word is its use in language." In other words, words that appear in similar contexts tend to have similar meanings.

To understand the concept of distribution in this context, consider a word's distribution as the collection of contexts in which it is used, with each context being defined by the neighbouring words. By analyzing the distribution of words, NLP models can gain valuable insights into the relationships between words and the underlying semantic meaning within a sentence or a larger text. This approach has been widely applied in various NLP tasks, such as text classification, information extraction, and sentiment analysis, demonstrating the effectiveness of the distributional hypothesis in understanding and modeling natural language.

10.4.2 Word Embeddings Combining Two Ideas

Word embeddings are based on two key ideas: the distributional hypothesis and vector representation. The distributional hypothesis, formulated by Firth in 1957, proposes that a word's meaning can be understood by analyzing the company it keeps in various sentences. This concept serves as a source of information for the word's meaning.

The second part of the idea involves vector representation, where words are represented as dense vectors rather than one-hot vectors. For example, the word "kitchen" may be represented by a 20-element vector. This allows us to think of the meaning of the word "kitchen" as a point in a 20-dimensional space. This process is known as embedding, as it embeds the word within this multidimensional space.

In summary, word embeddings are derived from the distributional hypothesis and vector representation. By examining the context in which a word appears and encoding its meaning in a dense vector, we can better understand the relationships and meaning behind words in a larger text, as they are now points in an embedding space. This approach has found great success in various NLP tasks, demonstrating the effectiveness of combining these two ideas.

10.4.3 Word-to-word Matrix

A word-to-word matrix, also known as a term-context matrix, represents the relationships between target words and the words that co-occur with them in the same document or within a specified window. The target word can be characterized by its surrounding context, allowing us to gain insights into its meaning and usage.

Each word in this matrix is represented by a sparse vector, which considers only the relevant context words for that target word. Consequently, the entire vocabulary is represented by a sparse word-to-word matrix, showcasing the connections between words based on their contextual usage within documents.

This approach has been widely adopted in NLP as it allows researchers and practitioners to better understand the relationships between words in natural language text, subsequently improving the accuracy and performance of various NLP tasks.

10.4.4 Dense Word Embeddings

Dense word embeddings are essential for creating short vectors that effectively represent the meaning and context of words. These dense vectors are preferable over sparse vectors due to several reasons:

1. **Fewer parameters:** A 100-dimensional dense vector requires only 100 weights compared to a 50,000-dimensional sparse vector, which needs 50,000 weights. This reduction in parameters simplifies the model and improves its performance.

2. **Easier to find large datasets:** With a smaller number of parameters, it is easier to find datasets containing more examples (documents) than

parameters in the model. This helps avoid overfitting and improves overall accuracy.

3. **Better representation of synonymy:** In a sparse representation, synonyms like "car" and "automobile" may be assigned to distinct dimensions, making it difficult to model their relationship. However, in a dense representation, there is more overlap between the dimensions, which allows for a better representation of synonymy.

4. **Optimal number of dimensions:** While too few dimensions can make it challenging to separate non-overlapping words, an appropriate range of dimensions helps effectively capture the nuances of word meanings and relationships.

In conclusion, dense word embeddings play a crucial role in improving the performance and accuracy of various NLP tasks by offering an optimal number of parameters, facilitating the representation of synonymy, and allowing for better modelling of the relationships between words.

We can think of the size of the vectors as a hyperparameter. Bigger vectors result in higher accuracy but take longer to process.

10.4.5 How to Learn Embeddings

Embeddings are effectively learned using various algorithms, with one notable example introduced by Mikalovl et al. in their paper, "Distributed Representations of Words and Phrases and their Compositionality." In this work, they propose two versions of the software called word2vec, which are Skip-gram and Common Bag of Words (CBOW).

1. **Skip-gram:** This method aims to predict target words given a context word. In other words, it tries to understand the context in which a word occurs and uses that context to predict other related words. This approach captures the relationships between words and their contexts, thereby learning embeddings efficiently.

2. **Common Bag of Words (CBOW):** Conversely, CBOW aims to predict the target context word(s) given a target word. In this algorithm, the target word is at the center of the context window, and the surrounding words are used to predict the target word's meaning. This approach helps in understanding the relationships between words and their meanings.

These two versions of word2vec software serve as effective methods to learn embeddings, subsequently improving the performance and accuracy of various NLP tasks.

This is an example of supervised learning, but the labels are not explicitly provided to us, we get them by exploiting the structure of natural language.

10.4.6 Word2vec: the basic algorithm

The word2vec algorithm functions as a binary classification algorithm, where the objective is to classify word occurrences in a given context. The algorithm is based on two types of examples: positive and negative.

1. **Positive examples:** These consist of a target word and its neighbouring context words as they occur in a given text, such as articles, books, or any human-written text.

2. **Negative examples:** These are words randomly sampled from the lexicon (V) and don't co-occur with the target word in the text.

The goal of the word2vec algorithm is to learn word embeddings efficiently using these positive and negative examples. To do this, a simple two-layer neural network is trained as a classifier to differentiate between positive and negative samples. The weights of the hidden layer representing target context words serve as the learned word embeddings.

This process is an example of self-supervised learning, where the algorithm relies on unlabeled text data to learn meaningful word representations without the need for explicit human supervision or annotated data. The resulting word embeddings can improve the performance and accuracy of various NLP tasks.

10.4.7 Other Embedding Algorithms

Besides the Skip-gram and CBOW algorithms in word2vec, other embedding algorithms have emerged to effectively learn word representations. These include:

1. **GloVe:** This method, similar to word2vec, is based on matrix factorization. It uses a slightly different learning approach but still aims to capture the relationships between words and their meanings.

2. **Neural networks:** Nowadays, learning embeddings directly in a neural network is a common approach. Here, each word is initialized with a vector, and back propagation is used to update these vectors like any other weight in the network.

These alternative algorithms have contributed to improving word embeddings, with the goal of providing accurate and efficient representations that can be used in various NLP tasks.

10.4.7.1 Representation Learning

In the context of NLP, representation learning refers to the process of learning meaningful representations for words. This involves encoding words as vectors, which are numerical representations that capture their semantic and syntactic properties. These vectors can then be used in various tasks, such as language modeling, text classification, and machine translation.

Learning word embeddings, for instance, is a popular representation learning method where the goal is to find a low-dimensional vector space that captures the semantics and relatedness of words. In other words, similar words should be close to each other in the vector space, while dissimilar words should be farther apart.

Creating effective representations requires large amounts of data and significant computational resources. However, once these representations are learned, they can be transferred across multiple tasks and lead to improved performance. Transfer learning in this context involves using pre-trained word embeddings as the starting point for various NLP tasks, thereby bypassing the need to learn new representations from scratch. This can save time and resources while also providing better performance than models trained from scratch.

10.4.8 How is word similarity represented with word embeddings?

In NLP, word similarity is represented using word embeddings, which are dense vectors. Related words often have similar vectors, as observed through dimensionality reduction techniques like t-SNE. In a two-dimensional space, for example, words like “kitchen” and “sink” are clustered together and are closer to one another than they are to words like “charger,” “battery,” or “Bosch.” This clustering helps represent the similarity between words based on their topic or semantic meaning.

Moreover, vector semantics enable the study of the relationships between words, which can further enhance our understanding of word similarity. By examining the mathematical operations (e.g., adding or subtracting) that can be performed on word embeddings, we can gain insights into how words are related to each other.

10.4.9 Biases in Word Embeddings

It is important to recognize that the text used for learning word embeddings can contain various forms of biases, such as racial and gender biases. These biases are often preserved in the resulting word embeddings. As a consequence, the word embeddings may exhibit biases in their recommendations and associations,

such as considering computer programmer the closest occupation to "man" and housemaker to "woman," or pairing "father" with "doctor" and "woman" with "nurse."

However, this preserved bias in word embeddings also offers a unique opportunity to study and track changes in biases over time by examining the differences between embeddings learned from texts from different time periods. Through such studies, NLP researchers can gain valuable insights into the evolving nature of biases in language and society.

10.4.10 Evaluation

Evaluation in NLP can be done through both extrinsic and intrinsic methods. Extrinsic evaluation typically involves using word embeddings within an NLP application, with the performance being measured by their impact on the overall task. The main drawback of extrinsic evaluation is that it can be quite time-consuming.

Intrinsic evaluation, on the other hand, is more focused on assessing the quality of the word embeddings themselves. One way to do this is by testing the performance on similarity tasks, using datasets like Simlex 999 or WordSim 353 to compare and analyze the results.

Another approach for intrinsic evaluation is to test the performance on analytical tasks of the type "A is to B as C is to D". For example, in a task involving country-capital relationships, the system would need to correctly establish that "Ottawa is to Canada as Rome is to Italy".

Finally, one can also evaluate the syntactical performance of word embeddings by testing their ability to handle tasks like converting singular words to their plural form. Given "mouse" and "mice", the system should correctly define "bikes" for the word "bike". This can provide valuable insights into the capabilities and limitations of the word embeddings in an NLP system.

10.4.11 Embeddings: words, sub-word or characters?

Embeddings can be based on words, sub-words, or character tokens. When using word tokens for embeddings, the main limitation is the large number of words, making it difficult to obtain relevant training data for every word. This often leads to problems with unusual words, misspellings, neologisms, and foreign language words.

A better approach to tokenizing text for embeddings is to compose words from smaller bits, which is referred to as sub-word tokenization. This method has become standard in NLP, with techniques like byte pair encoding or wordpiece

encoding being commonly used. By breaking words down into smaller components, the system can better handle variations in language and improve the accuracy of the embeddings.

10.4.12 Pre-trained Word Embeddings

Pre-trained Word Embeddings are widely available for use in NLP applications. Some examples of pre-trained embeddings include Word2Vec embeddings trained on Wikipedia content in various languages, as well as those available through libraries like GenSim, TensorFlow, and Keras. Using pre-trained embeddings can save time and resources, as they have already been optimized for various language tasks. It is generally recommended to make use of these pre-trained embeddings instead of training your own from scratch.

10.4.13 How do the word embeddings fit in the language model pipeline?

In the language model pipeline, word embeddings play a crucial role in representing the words within the model. Firstly, the input text is tokenized using a sub-word or word tokenization method. This process produces a list of tokens that are then translated into indexes, which correspond to their positions in the vocabulary.

Next, these indexes are used to look up the corresponding embeddings in an embedding lookup table, which is arranged with the same number of rows as the vocabulary and a fixed number of columns representing the size or length of the embeddings. By looking up the embedding for a specific word with an index of 4, for example, we can retrieve its vector representation in row 4 of the table.

Once the word embeddings are obtained, they are fed into a new language model, such as a GBT (Gaussian-Bernoulli Triangle), LSTM (Long Short-Term Memory), or BERT (Bidirectional Encoder Representations from Transformers). The output from this new model is a set of indexes, which are obtained using softmax activation and represent the most likely next words in the sequence.

These indexes are then mapped back to the original vocabulary, returning the predicted tokens. The end result is a predicted text that incorporates the understanding of word relationships and context gained through the use of word embeddings in the language modeling process.

10.5 Text Embeddings

Text embeddings are not limited to words; entire sentences, paragraphs or longer texts can also be embedded. There are different ways to embed sentences, each with its own advantages and disadvantages.

One method is to average the word embeddings over the words in a sentence. While this approach loses the word order information, it is computationally efficient and does not require extensive hardware resources. This method is more effective than using a normal bag of words approach.

Another way to embed sentences is by using a sequence model like an RNN (Recurrent Neural Network) or a transformer-based model. In this case, the output is the hidden state of the RNN or the transformer, which allows us to be sensitive to word order and achieve better results. However, this approach is more computationally expensive and requires better hardware.

These sentence embeddings can be used for various purposes, such as transfer learning in text classification tasks. Additionally, they can be used for measuring sentence similarity, performing clustering, and other related tasks.

10.5.1 Sentence Similarity

Sentence similarity can be measured using a combination of text embeddings and similarity metrics. Firstly, we need to embed sentences using a sequence model such as a language model, which produces embeddings as vectors.

For example, if we have three sentences: “I took my dog for a walk”, “I took my cat for a walk”, and “It is going to rain today”, we can use a model to generate embeddings. In this case, let’s say the resulting embeddings are six-element long vectors.

Next, we need to determine a similarity metric to compare these vectors. Since the vectors are not normalized, we can use the cosine similarity to focus on the direction and not the absolute magnitude. We can apply this similarity measure pairwise to the three vectors, yielding a similarity matrix.

The similarity matrix reveals that the first and second sentences are more similar to each other than the first sentence is to the third sentence. This demonstrates a simple example of measuring sentence similarity using embeddings.

With this representation, we can further perform tasks such as clustering or similarity search for question answering. For instance, if we have a query about clustering documents using embeddings and we embed relevant forum posts on NLP, we can compute similarities and return the nearest neighbours as answers. This approach effectively acts as a question-answering system by identifying and returning the most similar documents to the given query.

Chapter 11

Text Classification with Sequence Models

Written by Hjalmar K Turesson and Stable Beluga 2

11.1 Introduction

We will delve into the realm of sentence-level text classification and explore various aspects of sentiment analysis. Our discussion will encompass single-sentence sentiment analysis as well as aspect-based sentiment analysis. We will then proceed to examine an outdated state-of-the-art sentiment analysis model. Afterward, we will introduce SetFit, a distinct method of fine-tuning sentence-level text classification embedding models tailored for sentiment analysis. Furthermore, we will touch upon tokenization techniques specifically designed for pre-trained embedding models and language models. Overall, this chapter aims to provide a comprehensive overview of the fundamental concepts and methods involved in the classification of short texts, thus equipping readers with the necessary tools to analyze, understand, and fine-tune text classification models in the field of Natural Language Processing (NLP).

11.1.1 Contents

11.2 Sentiment analysis

Sentiment analysis, also known as opinion mining, can be conducted at three different levels: document, sentence, and aspect or feature. Document-level

analysis involves examining longer texts, such as articles, whereas sentence-level analysis focuses on shorter content, such as single sentences, tweets, and reviews. Finally, aspect-level analysis predicts sentiment toward specific features or aspects of an entity, which will be explored further in our discussion. By understanding these levels of sentiment analysis, we can effectively process and interpret various forms of text to gain insights into public opinion and sentiment.

11.2.1 Document-level Sentiment Analysis

Document-level sentiment analysis has traditionally relied on BOW representations. Generally, it provides high accuracy, around 90%, for long documents due to the prevalence of strong sentiment words such as "horrible," "awesome," "great," or "bad." However, achieving a similar accuracy for single sentences or review data sets using BOW has proven challenging, with an accuracy rate of only around 80%.

Document-level sentiment analysis using bag-of-words methods involves counting positive and negative words, but it ignores word order and context. This approach regards sentences with the same words but different sentiment, such as "white blood cells destroying an infection" and "infection destroying white blood cells," as having the same sentiment. The bag-of-n-grams extension considers context in short-word sequences but can lead to data sparsity and high dimensionality problems. Thus, Bag-of-words does not capture semantics effectively, resulting in equal distances between words that should have different relationships, such as "smart," "clever," and "book."

Single-sentence sentiment analysis necessitates a heightened sensitivity to word order, negation, and its scope, as well as other semantic effects, which explains why bag-of-words based sentiment analysis is not particularly effective for single sentences. To accurately conduct sentiment analysis on shorter texts, more sophisticated methods that take these factors into account need to be employed. A more suitable representation would be word embeddings, which encode semantic relationships between words and provide better insights for sentiment analysis.

11.2.2 Aspect-based Sentiment Analysis

Aspect-based sentiment analysis (ABSA), also known as opinion mining (aka feature-based or aspect sentiment analysis) on different features or aspects of entities, involves predicting the sentiment expressed on the attributes or components of an entity. Examples of entities could include a cell phone, digital camera, or bank, while features or aspects might comprise the screen of a cell phone, service of a restaurant, or picture quality of a camera.

Different features can elicit varying sentiments; for instance, a hotel could have a convenient location but mediocre food. In this case, the entity is the hotel,

and the aspects are location and food, with a positive sentiment expressed about the location and a negative or neutral sentiment about the food. By conducting aspect-based sentiment analysis, it is possible to gain more granular insights into public opinion regarding specific features of an entity.

11.2.2.1 ABSA Usage

ABSA enables businesses to conduct a detailed examination of customer feedback for various aspects or features of a complex entity. For instance, consider a sentence that states "Staff are not that friendly, but the taste covers all." The aspect-based analysis could reveal that the sentiment is negative when considering the aspect "service" (staff friendliness) and positive when evaluating the aspect "food" (taste). This level of granularity allows businesses to identify the specific aspects that customers like or dislike, ultimately informing their decision-making processes and improving customer satisfaction.

11.2.2.2 Examples of Camera Reviews

- "Nice & **compact** to carry!"
- "Since the camera is **small & light**, I won't need to carry around those heavy, bulky professional cameras either!"
- "The camera feels **flimsy**, is **plastic and very light** in weight you have to be very delicate in the handling of this camera"

In the above camera review example, we have an entity (camera) with different features, such as size, feel, material, and weight. The aspect-based sentiment analysis reveals the following sentiments:

1. The size is a positive feature, as the camera is described as "nice and compact to carry."
2. The aspect of size is mentioned again with the sentence, "since the camera is small and light, I won't need to carry around those heavy, bulky professional cameras," which conveys another positive sentiment about the size.
3. The "feel" of the camera is a negative feature, as it is described as "flimsy," implying that the reviewer feels it's not sturdy.
4. The material (plastic) and weight (very light) are also given negative sentiments, as they contribute to the overall flimsy feel of the camera and require extra care when handling.

By analyzing these aspects, we can gain a better understanding of the reviewer's overall opinion on the camera, highlighting both its positive and negative features.

11.2.3 Conclusion

ABSA involves several sub-problems, as demonstrated in the provided examples. The main challenges include:

1. Identifying relevant entities: In this example, the entity is the camera.
2. Extracting the features or aspects of the entity: The aspects mentioned in the example are size, weight, feel, and material.
3. Performing sentiment analysis on the opinions expressed about each feature or aspect: Analyzing the sentiment for each aspect allows for a more comprehensive understanding of the reviewer's opinion on the camera.

By addressing these sub-problems, ABSA enables a more granular understanding of customer feedback, helping businesses identify the specific aspects or features that customers appreciate or dislike.

11.2.4 Sentence-level Sentiment Analysis

Sentence-level sentiment analysis involves generating a sentiment score for each sentence, which can be challenging due to the complexity of natural language. An example sentence like "this film does not care about intelligence with or any other kind of intelligent humor" may contain positive words like "care," "cleverness," and "intelligent humor," but the overall sentiment of the sentence is negative.

11.2.4.1 Challenges

Sentence-level sentiment analysis encounters various challenges, such as handling contrastive conjunctions, negating positive sentences, and understanding the context of words. In one example, "slow and repetitive" are contrasted with "just enough spice to keep it interesting," resulting in a somewhat positive sentiment. Another example of a negating positive sentence is, "I liked every single minute of this film," which has a positive sentiment, but when switched to "I didn't like a single minute of this film," the sentiment becomes negative. By recognizing and addressing these challenges, accurate sentence-level sentiment analysis can be achieved. In the context of sentence-level sentiment analysis, it is crucial to employ a learning algorithm that is sensitive to both semantics and syntax. To address the issue of semantics, we require a sentence representation which effectively preserves semantic relations. Sub-word or sub-word embeddings can serve as efficient means of representing the meaning of individual words. Following this, it is important to combine these embeddings into full sentence embeddings that are capable of maintaining word order, as the meaning of certain words can heavily depend on their position within the sentence.

In order to handle the challenges posed by syntax, we must utilize an algorithm that is proficient in learning sequential patterns. This can be accomplished by implementing recurrent neural networks like RNNs (Recurrent Neural Networks) or LSTM (Long Short-Term Memory), as well as GRU (Gated Recurrent Units) and Transformers. These models allow the learning algorithm to effectively identify and analyze the intricacies and patterns within the sentence structure, ultimately enhancing its overall performance in sentiment analysis.

11.3 Understanding an old SOTA model

This model, which was state-of-the-art in 2017, offers a valuable foundation for comprehending the key principles of sentence embedding-based NLP. It is based on a version of Long Short-Term Memory (LSTM) and developed by OpenAI. This model is trained on two distinct datasets: the first one focuses on self-supervised representation learning, while the second is dedicated to supervised sentiment classification. By analyzing this particular architecture, students can gain deeper insights into the essential techniques and methods behind advanced NLP models.

11.3.1 Datasets

11.3.1.1 Supervised learning

The supervised dataset utilized in this model is the Stanford Sentiment Treebank (SST), which consists of movie review excerpts from the Rotten Tomatoes website. Compiled by Pang and Lee (2005), the dataset contains 11,855 sentiment sentences, with half being positive and the other half negative. In this dataset, it is observed that the sentiment expressed in the sentences becomes apparent with an increasing number of words, emphasizing the need for a model that can learn sequences of words to effectively classify sentiment.

11.3.1.2 Self-supervised learning

For unsupervised or self-supervised learning, the second dataset consists of 82 million reviews from Amazon, spanning from May 1996 to July 2014. This dataset, totalling around 10 gigabytes of data, is not labelled and contains reviews with both positive and negative sentiments. To address this challenge, a multiplicative Long Short-Term Memory (LSTM) model with 4,096 neurons is employed.

First, the LSTM is pre-trained on the Amazon review dataset through a self-supervised learning task that involves predicting the next character in the sequence. This process continues for approximately a month, during which a

representation similar to word embedding is learned. This representation is found in the activity of the 4,096 hidden neurons within the LSTM. The next goal is to learn sentiment based on this representation, which can be thought of as a 4,096-element vector.

11.3.2 The Model

The complete model consists of two parts: a sentiment classifier and a language model.

The language model takes input one character at a time and predicts the next character, utilizing a recurrent model that "remembers" the characters as the sentence is processed. The hidden state of the language model, comprising 4,096 hidden units or neurons, serves as a feature vector for the sentiment classifier.

The sentiment classifier is trained in a supervised manner with positive or negative sentiment labels, allowing it to learn the meaning of sentences from the language model's hidden state representations. During training, each labeled sentence is encoded using the language model, resulting in an embedding vector per sentence, which is then used to train the sentiment classifier.

In use, a sentence is first encoded by passing it through the language model. The resulting vector representation is then fed into the sentiment classifier, which predicts whether the sentence expresses positive or negative sentiment.

11.3.3 Performance

The performance of this 2017 model on the Stanford Sentiment Treebank was 91.8%, which was considered the state of the art at the time. However, in 2019, the T5-11B model achieved a state-of-the-art performance of 97.5%. Since then, no better performing model has been introduced, making it the current state of the art.

A key advance of the 2017 model was its ability to be trained with 30 to 100 times fewer label examples compared to previous models, thanks to self-supervised pre-training. This allowed the model to perform well, such as 88% accuracy with just 11 labelled examples. This was a significant improvement from the state of the art in 2013, which was around 85% and used all 11,855 labelled examples. The combination of self-supervised pre-training and a classification head enabled the 2017 model to outperform earlier models with significantly fewer labelled examples.

11.3.4 Few-shot learning

Few-shot learning was popularized by the article "Language Models are Few-Shot Learners," which is the GPT-3 paper. In this context, a language model

is given a task description and a small number of examples, but no gradient updates or training is performed on the given task. An example of this could be the prompt:

```
Translate English to French:

Sea otter => loutre de mer

Peppermint => menthe poivrée

Plush girafe => peluche

Cheese =>
```

This approach was first demonstrated in the GPT-3 paper, which also analyzed how the number of parameters in the model impacted few-shot learning performance. They found that as the model size increased, few-shot learning improved, potentially in a logarithmic manner with the number of parameters. This indicates that larger language models are better suited for few-shot learning tasks without additional fine-tuning.

11.3.4.1 Few-Shot Learning With Sentence Transformers

Few-shot learning doesn't have to rely on prompt engineering; it can also involve weight updates. In this context, "n-shot learning" refers to the number of examples provided in each class. For instance, n=1, or one-shot learning means that one example is provided for each class, two-shot learning means n is 2, and so on.

The model is fine-tuned on a support set with n examples of each class and then given a query image. The model must determine the class in the query's support set most similar to. If the model, in the example below correctly identifies the animal as a pangolin, it has successfully completed a two-shot learning task based on the provided examples.

This task involves determining if two inputs are similar or different, a task commonly performed using siamese networks, which will be discussed in a later class. To learn this ability, data sets such as the Meta dataset and the Mini-ImageNet are utilized to provide input examples for the model to differentiate between the same or different inputs.

11.3.4.2 SetFit

SetFit is a library for few-shot fine-tuning of text classifiers. It has pre-trained sentence transformers that have been trained to represent sentences efficiently. Fine-tuning in SetFit is accomplished using contrastive laws, which are specific functions applied to a small labeled dataset. This process helps the model determine whether two sentences are similar or different.

After completing the fine-tuning process, we can then add a classification step. To do this, we use the encodings produced by the fine-tuned transformer model to train the classification head, allowing the system to correctly classify text. One advantage of SetFit is its ability to achieve strong performance with as few as eight examples of each class, making it an efficient approach for text classification with minimal labeled data. In some cases, SetFit can even show good results with only four or three shots of learning.

11.3.5 Tokenization for Pre-trained Models

To effectively utilize pre-trained models, such as text encoders like transformers or large language models, using the tokenizer employed during their training is crucial. The specific tokenizer used depends on the corpus it was fitted on. It is recommended to recall how byte-pair encoding or word-piece algorithms function.

Consider the example of using a tokenizer from the HuggingFace Transformers library. First, import the desired tokenizer architecture. For instance, to import the tokenizer for the BERT model, load the tokenizer and specify the exact pre-trained BERT model to use, such as in this case, a BERT model "bert-base-uncased". To obtain the tokenized form, call the `tokenize()` method on the tokenizer object and pass in the text as an argument. This will return a list of strings representing the tokenized output:

```
from transformers import BertTokenizer
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

text = "here is a sentence adapted to our tokenizer"

print(tokenizer.tokenize(text))
```

```
['here', 'is', 'a', 'sentence', 'adapted', 'to', 'our', 'token', '##izer']
```

By using the correct tokenizer for the given pre-trained model, the resulting tokens will be properly formatted for use in downstream tasks, such as classification or generation.

However, when using a pre-trained tokenizer, it is important to note that it may not perform as well with sentences that are dissimilar from the corpus it was trained on. The dissimilarities could arise from a variety of factors, such as a new language, new characters, a new domain, or a new style.

11.3.5.1 Issues With Documents from a New Domain

To illustrate the limitations of using pre-trained tokenizers on dissimilar text, consider an example where the tokenizer faces a medical text. The input text

contains medical terms, such as "paracetamol" and "pharjitis". Using the same BERT tokenizer from the previous example, we observe that the medical terms are incorrectly divided into sub-tokens.

```
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
text = ("the medical terms are often divided into sub-tokens: "
   "paracetamol, pharyngitis")

print(tokenizer.tokenize(text))
['the', 'medical', 'terms', 'are', 'often', 'divided', 'into', 'sub', '-', 'token', '##s', ':', '
```

For instance, the drug "paracetamol" is divided into four sub-tokens: `'para'`, `'##ce'`, `'##tam'`, `'##ol'`. Similarly, "pharyngitis" is divided into four sub-tokens: `'ph'`, `'##ary'`, `'##ng'`, `'##itis'`. These sub-tokens correspond to indices into the word embedding lookup table, but the word embeddings for `'ph'`, `'##ary'`, `'##ng'` and `'##itis'` are likely not informative about the parent word "pharyngitis". The result is that the word embeddings for unusual or domain-specific words can be of low quality, reducing the model's ability to effectively process such words and potentially compromising performance.

11.3.5.2 Issues With Documents from a New Language

We can see a similar issue when using a pre-trained tokenizer, such as BERT, on text in another language, like Hindi. In this example, when we tokenize the Hindi text using the bert-base-uncased tokenizer, each word is broken down into multiple sub-tokens. This is because the tokenizer was trained on a different language and is not well-suited to handling inputs from a different language or domain. It highlights the importance of selecting a tokenizer that is appropriate for the specific input text or adapting the tokenizer to better handle new languages or domains.

11.3.5.3 Conclusions

When dealing with pre-trained models such as text encoders and language models, it is essential to use a tokenizer that was specifically trained for the given task or domain. In cases where the tokenizer produces sub-token embeddings that are less informative about the intended meaning of a word, it would be preferable to have a tokenizer that tokenizes whole words for improved performance.

To achieve better performance, ensure that the tokenizer is trained on a corpus similar to the one used in deployment. If training a new model from scratch,

it is crucial that the tokenizer is also trained on a similar corpus, as this will enhance the compatibility between the tokenizer and the model. To do this, simply obtain a relevant corpus, choose a tokenizer architecture, feed the corpus into the tokenizer, and save the trained tokenizer for use with your model.