FITPED | AI

Translation Model Practice

Ľubomír Benko
Kornel Chromiński

www.fitped.eu

2024

Funded by the European Union

Erasmus+ FITPED-AI
Future IT Professionals Education in Artificial Intelligence
(Project 2021-1-SK01-KA220-HED-000032095)

Translation Model Practice

Published on

November 2024

Authors

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

Kornel Chromiński | University of Silesia in Katowice, Poland

Reviewers

Piet Kommers | Helix5, Netherland

Vaida Masiulionytė-Dagienė | Vilnius University, Lithuania

Małgorzata Przybyła-Kasperek | University of Silesia in Katowice, Poland

Ivo Písařovic | Mendel University in Brno, Czech Republic

Erasmus+ FITPED-AI

Future IT Professionals Education in Artificial Intelligence

Project 2021-1-SK01-KA220-HED-000032095

Funded by
the European Union

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or Slovak Academic Association for International Cooperation. Neither the European Union nor the granting authority can be held responsible for them.

ISBN 978-80-558-2231-0

TABLE OF CONTENTS

Basic Information

Chapter 1

1.1 Theoretical basics

1.1.1

A **regular expression** (regex or regexp) is a string that defines a pattern for searching text using special characters. In general, regular expressions come from the field of theoretical computer science and the theory of formal languages. Regular expressions are most commonly used in text manipulation in the following cases:

- determining whether a text matches a regular expression (input validation),
- text search - finding out where in the text the substring being searched for is located,
- replacing a substring in a string (find and replace),
- extracting all occurrences of a substring.

Regular expressions in Python are written in the so-called raw string, which ensures that special sequences of characters do not have their special meaning activated (e.g. \n creates a new line):

```
r""
```

A raw string is an ordinary string but with a different form of notation. We can also write regular expressions as plain text strings, but then we have to watch out for special character sequences, which we have to treat with a double slash. This reduces the clarity of regular expressions.

```
print("hello\nworld")
print(r"hello\nworld")
```

1.1.2

Why are raw strings (denoted as r"") commonly used for regular expressions in Python?

- They ensure that special character sequences do not activate their usual meanings.
- They automatically create new lines in regular expressions.
- They convert special sequences into plain text.
- They make regular expressions case-insensitive.

1.1.3

Some characters have special meanings in the case of regular expressions. This means that if we want to use them, we have to prefix them with a backslash character, which ensures that the program does not take the special meaning into account. For example, **\n** indicates a new line. However, if we write it as a **raw string**, its special meaning will not be used.

```
print("hello\nworld")
print(r"hello\nworld")
```

1.1.4

Thus, a regular expression defines a so-called search pattern for strings. In programming languages, there are functions to check whether a given string satisfies the regular expression condition. In Python, we will use the **re** library, which contains the **match()** function. This function verifies that the string matches the regular expression and if it finds a match, it returns a special object of type *Match*. If it does not find a match it returns an empty value of the form **None**. The Match object contains a number of useful pieces of information that we can work with later, such as information about where in the string the matched regular expression is located.

There are a number of symbols and characters that have a specific function in the case of regular expressions:

- . (dot) - this is a wildcard that represents an arbitrary character,
- ^ - identifies the beginning of a string or line,
- $ - identifies the end of a string or line.

```
import re

print(re.match(r'^I','Informatics')) # does the word start
with a capital i?
print(re.match(r'$a','Informatics')) # end the word with a
small a?
```

1.1.5

What does the **match()** function from Python's **re** library return if a string does not satisfy the regular expression condition?

- The value None.
- A Match object with no data.

- An empty string.
- The index of the mismatch.

1.1.6

Regular expressions provide powerful symbols for defining search patterns in text. Among these are **.** and **$**, which help us match specific characteristics in strings. The **.** symbol acts as a wildcard for any character, while the **$** symbol ensures the pattern matches at the end of a string. Understanding these symbols is crucial for tasks like searching and validating text.

Using . (Dot) as a wildcard

The dot (**.**) matches **any single character**, except for a newline. This makes it versatile for general patterns:

- The pattern **A.** matches "Al" in "Alan" or "Ab" in "Albert" because **.** matches any character following "A."
- However, it will not match "A" on its own, as there must be one character after "A" to satisfy the pattern.

```
import re

pattern = r"A."
text = "Alan"

result = re.match(pattern, text)
if result:
    print("Match found:", result.group())  # Outputs 'Al'
else:
    print("No match.")
```

1.1.7

Which regular expression matches any string starting with "A" followed by any single character?

- r"A."
- r".A"
- r"A$"
- r"A*"

1.1.8

Using $ to match the end of a string

The dollar sign (**$**) asserts that the pattern must occur at the **end of the string**. For instance:

- The pattern **t$** matches any string ending in "t," like "Alphabet" or "cat."
- It will not match "train" or "Alphabetical" because these strings do not end with "t."

```
import re

pattern = r"t$"
text = "Alphabet"

result = re.match(pattern, text)
if result:
    print("Match found:", result.group())  # Outputs
'Alphabet'
else:
    print("No match.")
```

Combining . and $

We can combine these symbols for more complex patterns. For example, **..$** matches any string ending with exactly two characters. In "test," it will match "st."

1.1.9

Which regular expression checks if a string ends with the word "End"?

- r"End$"
- r".End"
- r"^End$"
- r"End."

1.1.10

Which regular expression matches a string ending with any single character?

- r".$"
- r"^.$"
- r"$.+"
- r".*.$"

1.1.11

Regular expressions allow us to define patterns for matching text, including checking if a string starts with a specific character. This is particularly useful in programming tasks like validating input or searching text efficiently. To accomplish this in Python, we use special symbols like **^**, which asserts that the pattern must match at the beginning of a string.

How the ^ symbol works

The **^** symbol is used in regular expressions to assert that the match must occur at the start of the string. For instance:

- The pattern **^A** matches any string that starts with the letter "A."
- It will match "Apple" and "Alphabet" but not "Banana" or "apple."

This symbol does not check for any other conditions beyond the beginning of the string, so combining it with specific characters or words makes it very precise.

Using ^ in Python with re.match()

In Python, you can use the **re.match()** function from the **re** library to check if a string matches a regular expression at the beginning of the text. Here’s how you can check if the word "Alphabet" starts with a capital "A":

```
import re

pattern = r"^A"  # Regular expression
text = "Alphabet"

# Check if the text starts with 'A'
result = re.match(pattern, text)

if result:
    print("Match found!")
```

```
else:
    print("No match.")
```

In this case, the **^A** pattern asserts that the string must start with "A." If the text satisfies this condition, **re.match()** returns a **Match** object. Otherwise, it returns **None**.

- Always use raw strings (**r""**) for regular expressions in Python to avoid unintended interpretations of escape sequences.
- The **^** symbol is case-sensitive, so **^A** will not match "alphabet" since it starts with a lowercase "a."

1.1.12

Which regular expression checks if the word "Alphabet" starts with a capital "A"?

- r"^A"
- r"A$"
- r".A"
- r"^a"

1.2 Simple examples

1.2.1

Which regular expression checks if the word "Alphabet" starts with a capital "A"?

```
# you can solve / check it by code
import re
```

1.2.2

Which regular expression checks if the string ends with the word "com"?

```
# you can solve / check it by code
import re
```

1.2.3

Quantifiers are used in regular expressions to specify how many times a character or a group of characters can appear in a string. These are important for defining flexible patterns. Here are the most commonly used quantifiers:

- **?** - means no or just 1 occurrence. The **?** quantifier specifies that the preceding character or group can appear zero or one time. It makes the preceding character optional. Example: **r"colou?r"** matches both "color" and "colour".
- * - means no or more occurrences, specifies that the preceding character or group can appear zero or more times. Example: **r"ab*c"** matches "ac", "abc", "abbc", "abbbc", etc.
- **+** - means just 1 or more occurrences, specifies that the preceding character or group must appear one or more times. Example: **r"ab+c"** matches "abc", "abbc", "abbbc", but not "ac".
- **{number}** - means the exact number of occurrences defined in brackets, specifies that the preceding character or group must appear exactly the given number of times. Example: **r"ab{2}"** matches "abb", but not "ab" or "abbb".
- **{min, max}** - means the number of occurrences between the defined boundaries, including the boundaries, specifies that the preceding character or group should appear between the minimum and maximum number of times. Example: **r"ab{2, 4}"** matches "abb", "abbb", and "abbbb", but not "ab" or "abbbbb".

```
import re
print(re.match(r'.*','hello world'))
print(re.match(r'hel{2}o','hello'))
print(re.match(r'ab*a','abbbbba'))
```

1.2.4

What does the quantifier **?** mean in regular expressions?

- No or one occurrence
- Exactly one occurrence
- One or more occurrences
- Any number of occurrences

1.2.5

What does the quantifier ***** mean in regular expressions?

- No or more occurrences
- Exactly one occurrence
- One or more occurrences
- A specific number of occurrences

1.2.6

What does the quantifier **+** mean in regular expressions?

- One or more occurrences
- No or one occurrence
- No or more occurrences
- A specific number of occurrences

1.2.7

What does **{2}** mean in a regular expression?

- Exactly 2 occurrences
- At least 2 occurrences
- 2 or more occurrences
- No or 2 occurrences

1.2.8

What does **{2,4}** mean in a regular expression?

- Between 2 and 4 occurrences
- Exactly 2 occurrences
- 2 or more occurrences
- No or 4 occurrences

1.2.9

Which regular expression correctly verifies and evaluates the word "pool"?

```
# you can solve / check it by code
import re
```

1.2.10

Which regular expression correctly verifies and evaluates all the words "abbc", "abbbc", and "abbbbc"?

```
# you can solve / check it by code
import re
```

1.3 Abbreviated notation

1.3.1

To avoid having to manually print all the characters of a given word or to cover a range of characters, such as when checking the quality of a password, we can use *character groups* in regular expressions. Character groups allow us to specify a set of characters that can match a single position in the string.

Here are ways to define character groups:

- By listing individual characters - example, [cde] will match any one of the characters 'c', 'd', or 'e'.
- By using a character range - example, [a-z] will match any lowercase letter from 'a' to 'z'.
- By combining different characters and ranges - example, [cdex-z1-3] will match either 'c', 'd', 'e', any letter from 'x' to 'z', or any digit from 1 to 3.

1.3.2

Which of the following regular expressions matches any one of the characters 'c', 'd', or 'e'?

- [cde]
- [aeiou]
- [a-z]
- [1-9]

1.3.3

What does the regular expression **[a-z]** match?

- All lowercase letters from 'a' to 'z'
- Any digit between 1 and 9
- All characters between 'a' and 'z', including all digits
- Any uppercase letter from 'A' to 'Z'

1.3.4

Which regular expression matches 'c', 'd', 'e', any letter from 'x' to 'z', or any digit between 1 and 3?

- [cdex-z1-3]
- [cde1-3]
- [x-zcd1-3]
- [cd1-3]

1.3.5

In regular expressions, we can also use sequences of wildcard characters that simplify the expression of certain patterns:

- **\d** - matches any digit, equivalent to the range **[0-9]**.
- **\D** - matches anything except digits.
- **\s** - matches any white space character, such as spaces, tabs, etc.
- **\S** - matches anything except white space characters.
- **\w** - matches all alphanumeric characters and underscores, similar to the range **[a-zA-Z0-9_]**.
- **\W** - matches all characters except those matched by \w.
- | (vertical bar) - is often used in combination with groups of characters. It separates different parts of a regular expression, allowing us to match any one of them. For example, the regular expression **aaa|bbb** matches either the string 'aaa' or 'bbb'.

More complex examples, like **([ab]{2}|z)k**, match either the two characters 'a' or 'b' followed by 'k', or the letter 'z' followed by 'k'. Using this notation, we can match words such as "aak", "abk", "bak", "bbk", and "zk".

1.3.6

What does the regular expression **\d** match?

- Any digit
- Any whitespace character
- Any letter
- Any alphanumeric character

1.3.7

What does **\D** denote in a regular expression?

- All characters except digits
- All digits
- Any whitespace character
- All letters

1.3.8

What does the regular expression **\s** match?

- Any whitespace character
- Any letter
- Any digit
- Any punctuation mark

1.3.9

What does the regular expression **\w** match?

- All alphanumeric characters and underscores
- Only uppercase letters
- Only digits
- Only lowercase letters

1.3.10

Which regular expression would match either the string 'aaa' or 'bbb'?

- aaa|bbb
- a{3}|b{3}
- a|b
- (aaa|bbb)

1.3.11

Which regular expression matches words such as 'aak', 'abk', 'bak', 'bbk', and 'zk'?

- ([ab]{2}|z)k
- [ab]{2}k
- a|b{k}
- a{2}|z{k}

1.4 Functions

1.4.1

With regular expressions, we have several functions that allow us to search for patterns in a string:

- **search()** - looks for the first part of a string that matches the regular expression. It returns only the first occurrence found.
- **findall()** - returns all matching substrings in the string. It finds all occurrences that match the pattern without overlapping.
- **finditer()** is similar to **findall()**, this function returns an iterator that yields match objects for each match. It can be used to obtain detailed information about each match, such as its position in the string.

It’s important to note that when using **findall()** and **finditer()**, the functions traverse the string from left to right and do not recheck characters already evaluated as part of a previous match. This means overlapping matches are not included in the results. On the other hand, search() is only concerned with finding the first match and does not consider overlaps.

```
import re
print(re.search(r'.n', 'Good evening!'))
print(re.findall(r'.n', 'Good evening!'))
print(list(re.finditer(r'.n', 'Good evening!')))
```

1.4.2

Which function returns only the first occurrence of a match in a string?

- search()
- match()
- finditer()
- findall()

1.4.3

What does the **findall()** function return?

- All matching substrings
- The first match
- An iterator of match objects
- Only the last match

1.4.4

What does the **finditer()** function return?

- An iterator of match objects
- A list of strings
- Only the first match
- A dictionary of matches

1.4.5

Which function does not allow for overlapping matches in the string?

- findall()
- search()
- finditer()

1.4.6

What happens when you use **findall()** or **finditer()** in terms of overlapping matches?

- They ignore previously evaluated characters
- They allow overlapping matches
- They return the same match multiple times
- They only return the last match

1.4.7

Another possibility of applying regular expressions is **substitution**. The **sub()** function in Python is used to replace parts of a string that match a regular expression. It works very similarly to the string method **replace()**, with the key difference being that the replaced part must match the condition of the regular expression.

The **count** parameter in **sub()** specifies how many substrings to replace:

- If **count** is set to 1, only the first occurrence is replaced.
- If **count** is set to a higher number, that many occurrences will be replaced.
- If **count** is not provided, all occurrences of the pattern will be replaced.

```
import re
print(re.sub(r'\s+and\s+', ' & ', 'Black  and white and  red
and  green  and  blue'))
print(re.sub(r'\s+and\s+', ' & ', 'Black  and white and  red
and  green  and  blue', count=1))
```

1.4.8

How can you replace only the first occurrence of a pattern using sub()?

- By setting the count parameter to 1
- By setting the count parameter to 0
- By not using the count parameter
- By using the replace() function

1.4.9

What is the function **sub()** used for in Python?

- To replace parts of a string that match a regular expression
- To search for patterns in a string
- To split a string into substrings
- To find the first match in a string

1.4.10

Write a regular expression that verifies that the given number is a three-digit number. Remember that a three-digit number must not start with zero.

```
import re
```

1.4.11

Write a regular expression that verifies that the given string is the beginning of a phone area code. A phone area code always starts with a **+** sign.

```
import re
```

1.4.12

Let's take a closer look at the **findall()** function, which can capture a group of characters/words that satisfy a regular expression. The **findall()** function captures all substrings in a given text that match a regular expression.

It returns a list of all the matches found in the input string.

For example, if we want to identify all the words that start with a capital letter, we can use a regular expression pattern that matches such words. For **r'\b[A-Z][a-z]*\b'**:

- **[A-Z]** matches any uppercase letter at the beginning of a word.
- **[a-z]*** matches any lowercase letters following the uppercase letter.
- **\b** - word boundary, ensuring that we match whole words.

```
import re
text = 'Sarah has lived at the house already 4 years. Peter
came to visit Sarah and John during summer.'
res = re.findall(r'[A-Z][a-z]+',text)
print(res)
```

Program output:

```
['Sarah', 'Peter', 'Sarah', 'John']
```

1.4.13

What does the **findall()** function return?

- A list of all matching substrings
- A dictionary of matches
- The first matching substring

1.4.14

Which regular expression pattern can be used to match words starting with a capital letter?

- r'\b[A-Z][a-z]*\b'
- r'[A-Z][A-Z]*'
- r'[a-z][A-Z]*'
- r'\b[A-Z]\b'

Examples

Chapter 2

2.1 Programs

2.1.1

Create a regular expression that successfully recognizes tagged users on social networks. The user's name is usually prefixed with @.

2.1.2 Tagging

Create a program that successfully recognizes tagged users on social networks. The user's name is usually prefixed with @.

```
Input : Correct me if I’m wrong, but I think that after
today’s release of #willow Chapter 7, Julian Glover is now the
first actor to hit all three @Lucasfilm franchises
Output: ['@Lucasfilm']
```

2.1.3

Create a regular expression that successfully recognizes decimal numbers in the text. Decimal numbers will be written with a decimal point.

2.1.4 Decimal numbers

Create a program that successfully recognizes decimal numbers in the text. Decimal numbers will be written with a decimal point.

```
Input : How do you guys read this as decimal value 0.015? I
read it as 15/100 (fifteen hundredth) and my co-worker read it
as 15/1000 (fifteen thousandth).
Output: ['0.015']
```

2.1.5

Create a regular expression that successfully recognizes fractions in text. Assume that the fraction will consist of positive integers. E.g. ¾

2.1.6 Fractions

Create a program that successfully recognizes fractions in text. Assume that the fraction will consist of positive integers. E.g. 3/4

```
Input : How do you guys read this as decimal value 0.015? I
read it as 15/100 (fifteen hundredth) and my co-worker read it
as 15/1000 (fifteen thousandth).
Output: ['15/100','15/1000']
```

2.1.7

Create a regular expression that successfully recognizes an email address in the text. Assume that the email address will be in the standard format. For example, john.smith@gmail.com

2.1.8 Email

Create a program that successfully recognizes an email address in the text. Assume that the email address will be in a standard format. For example, john.smith@gmail.com.

```
Input : john.smith@gmail.com
Output: ['john.smith@gmail.com']
```

2.1.9

Create a regular expression that successfully evaluates the password strength. The password could contain only upper and lower case letters and numbers. Password length should be at least 8 characters.

2.1.10 Password

Create a program that successfully evaluates the strength of a password. The password could contain only upper and lower case letters and numbers. Password length should be at least 8 characters.

```
Input : ab18aaPL
Output: ['ab18aaPL']
```

2.1.11

Create a regular expression that successfully evaluates the suitability of the username. The username can consist of alphanumeric characters as well as _ and - characters. The username must contain only lowercase letters. The length of the username should be at least 3 characters and no more than 16 characters.

2.1.12 Username

Create a program that successfully evaluates the suitability of a username. The username can consist of alphanumeric characters as well as _ and - characters. The username must contain only lowercase letters. The length of the username should be at least 3 characters and no more than 16 characters.

```
Input : beever10
Output: ['beever10']
```

2.2 Advanced Programs

2.2.1 URL

Create a program that successfully detects the URL in the text and informs whether it is written correctly or not. Assume that https is mandatory.

```
Input : https://www.ukf.sk
Output: True
```

2.2.2 IP address

Create a program that successfully detects the IP address in the text and informs whether it is entered correctly or not.

```
Input : 192.168.1.255
Output: True
```

2.2.3 Date I.

Create a program that successfully detects the date in text in the format YYYY-MM-dd and informs whether it is written correctly or not.

```
Input : 2022-02-02
Output: True
```

⌨ 2.2.4 Date II.

Create a program that successfully detects the date in text in the format dd/MM/YYYYY and informs whether it is entered correctly or not.

```
Input : 31/01/2022
Output: True
```

⌨ 2.2.5 Date III.

Create a program that successfully detects the date in the text in the format dd.mmm.YYYY and informs whether it is entered correctly or not. Write the months in English (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec). Ensure that the months have the correct number of days.

```
Input : 31.Jan.2022
Output: True
```

⌨ 2.2.6 Time I.

Create a program that successfully recognizes the time in text in the 12-hour format HH:MM and informs whether it is written correctly or not.

```
Input : 09:59
Output: True
```

⌨ 2.2.7 Time II.

Create a program that successfully recognizes the time in text in the 12-hour format HH:MM am/pm and reports whether it is written correctly or not. Be sure to check if the time contains am/pm information, which can also be written in upper case.

```
Input : 09:59 PM
Output: True
```

⌨ 2.2.8 Time III.

Create a program that successfully recognizes the time in text in 24-hour HH:MM format and informs whether it is written correctly or not. For hours and minutes 0-9, it is necessary to write a 0 in front of them.

```
Input : 23:59
Output: True
```

⌨ 2.2.9 Time IV.

Create a program that successfully recognizes the time in text in 24-hour HH:MM format and informs whether it is written correctly or not. For hours and minutes 0-9, it is not necessary to write a 0 in front of them.

```
Input : 5:59
Output: True
```

⌨ 2.2.10 Time V.

Create a program that successfully recognizes the time in text in the 24-hour format HH:MM:SS and informs whether it is written correctly or not. For hours and minutes 0-9, it is necessary to write a 0 in front of them.

```
Input : 05:59:42
Output: True
```

⌨ 2.2.11 URL slug

Create a program that successfully recognizes the URL slug in the text. This is the last part of the URL that serves as a unique identifier for the page. These are alphanumeric characters that can be separated by a hyphen. E.g. https://www.abc.com/this-is-a-slug/

```
Input : this-is-a-slug
Output: True
```

⌨ 2.2.12 Phone number

Create a program that successfully recognizes an international phone number in the text. Make sure the area code begins with a + or 00. Also, ensure that it recognizes a number with spaces, but also without spaces.

```
Input : 00421 123 459 21
Output: True
```

2.2.13 File name

Create a program that successfully recognizes a file name in text. Make sure that the file extension contains exactly 3 characters. The file name can contain alphanumeric characters.

```
Input : filename.txt
Output: True
```

2.2.14 Duplicity

Create a program that successfully detects if there are duplicate words in the text (can also be a number).

```
Input : hello world hello
Output: True
```

Natural Language Processing

Chapter 3

3.1 Introduction

3.1.1

Natural Language Processing (NLP) is an interdisciplinary field combining computer science, linguistics, and artificial intelligence to enable machines to process and understand human language. With advancements in AI, NLP has become integral to many modern applications. From virtual assistants like Alexa and Siri to sophisticated chatbots and sentiment analysis tools, NLP technologies are reshaping how we interact with machines.

The key goal of NLP is to bridge the gap between human communication and computational understanding. This involves tasks like parsing language syntax, identifying the meaning of words in context, and even understanding emotions and intents conveyed in text. NLP can process structured and unstructured data, making it a powerful tool in areas ranging from social media analysis to healthcare.

Throughout this course, students will explore foundational concepts of NLP, learn about preprocessing techniques to prepare textual data, and gain insights into machine translation technologies. The focus will be on understanding the mechanics of NLP and applying it effectively in real-world scenarios.

3.1.2

What is the primary goal of Natural Language Processing

- To enable machines to process and understand human languages.
- To create hardware for faster computations.
- To convert text into binary code for storage.
- To analyze numerical datasets.

3.1.3

NLP has found its way into numerous practical applications that touch everyday life. Chatbots, for instance, are widely used by businesses to handle customer inquiries efficiently. These AI-based systems simulate human conversations, providing users with instant responses to common questions. Beyond customer support, NLP powers virtual assistants like Google Assistant, which helps users manage their schedules, search for information, and even control smart home devices.

Another critical application is sentiment analysis, which identifies the emotional tone behind text. For example, businesses use sentiment analysis to gauge public opinions on social media or track customer satisfaction. This application highlights the ability of NLP to derive actionable insights from large volumes of unstructured data.

Machine translation, one of the oldest NLP applications, has evolved significantly with advancements in neural networks. Tools like Google Translate now provide more accurate translations by learning contextual relationships between words and phrases. These applications showcase the versatility and importance of NLP in today’s digital landscape.

3.1.4

Which of the following are common NLP applications?

- Virtual assistants like Alexa
- Sentiment analysis
- Financial forecasting tools
- Computer-aided design software

3.1.5

Text preprocessing is a crucial step in NLP workflows. It involves cleaning and preparing textual data to ensure compatibility with machine learning models. Raw text often contains noise, such as punctuation, special characters, and inconsistencies in case formatting. Removing this noise helps improve the performance and accuracy of NLP applications.

Common preprocessing techniques include tokenization, which splits text into smaller units like words or phrases, and lemmatization, which reduces words to their root forms. Another essential step is removing stop words, such as "and," "the," and "is," which do not add significant meaning to the analysis. Text preprocessing lays the foundation for effective NLP models by converting human language into a format that machines can process.

By mastering preprocessing techniques, students can ensure their NLP models focus on meaningful patterns in the data rather than irrelevant noise.

3.1.6

What is the purpose of text preprocessing in NLP?

- To improve the model's performance by cleaning and preparing text data.

- To generate training datasets automatically.
- To store text data in compressed formats.
- To calculate numerical features from images.

3.1.7

Machine translation is one of the most transformative NLP applications. The objective is to enable automatic translation of text from one language to another, bridging linguistic barriers and facilitating global communication. Early machine translation systems relied on rule-based methods, requiring extensive linguistic knowledge to create translation rules.

Modern approaches use neural networks and machine learning to achieve higher accuracy. Neural machine translation (NMT) models, such as Google's Transformer-based architecture, have revolutionized this field. These models learn patterns from vast multilingual datasets, capturing context and meaning more effectively than traditional methods.

Machine translation plays a critical role in various industries, including education, business, and healthcare. As students delve into this topic, they will understand the evolution of translation models and their underlying principles.

3.1.8

What are the primary goal of machine translation in NLP?

- To enable automatic translation between languages.
- To summarize large volumes of text.
- To create new languages for digital communication.
- To convert audio speech into text.

3.2 Vectorisation

3.2.1

Vectorization in NLP is the process of converting textual data into numerical representations that machines can understand and process. Human language, with its nuances and complexities, is difficult for computers to interpret directly. Vectorization bridges this gap by transforming text into mathematical constructs like vectors or matrices.

Imagine you want to teach a virtual assistant to understand human speech. Instead of manually teaching it grammar, sentence structure, and meaning, you could transform text into vectors that represent words or sentences in an n-dimensional space. Each dimension corresponds to a word, and the value in each dimension reflects the presence or frequency of that word. This numerical representation allows computers to analyze, compare, and manipulate language data mathematically.

Vectorization simplifies complex NLP tasks, making it foundational for applications like chatbots, sentiment analysis, and machine translation. By converting language into numbers, it enables the use of mathematical models to extract patterns, derive insights, and make predictions.

3.2.2

What is the main purpose of vectorization in NLP?

- To convert text into numerical representations for machine understanding.
- To store text data in binary format.
- To compress text data for storage efficiency.
- To create new languages for machines.

3.2.3

In NLP, raw text data is unstructured and cannot be directly used by machine learning models. Vectorization addresses this challenge by structuring the text into a format that models can process. Without vectorization, computers would struggle to interpret the semantic and syntactic aspects of human language.

For instance, a virtual assistant trained without vectorization would require manual input of every possible sentence structure, word, and rule, which is impractical. Instead, vectorization automates this process by creating a mathematical space where sentences are represented as vectors. These vectors encapsulate the relationships between words, enabling models to understand patterns such as word co-occurrence, context, and even sentiment.

Moreover, vectorization helps standardize text data, ensuring consistency in how machines interpret language. This step is essential for building reliable NLP models that can generalize across different datasets and applications.

3.2.4

Why is vectorization crucial in NLP?

- It enables models to interpret and process raw text data.
- It ensures the text data is uncompressed for analysis.
- It eliminates the need for training datasets.
- It simplifies manual grammar annotation.

3.2.5

The vectorization process converts text into numerical representations using techniques like bag-of-words, term frequency-inverse document frequency (TF-IDF), or word embeddings. Each method has its unique approach to capturing information from text.

The **bag-of-words (BoW)** model focuses on word frequency, creating a matrix where each row represents a document and each column corresponds to a word. Values in the matrix indicate the word count in each document. However, BoW ignores word order and context, which limits its ability to capture deeper meanings.

To address this, **TF-IDF** builds on BoW by weighting words based on their importance. Words that are frequent in a document but rare in the corpus receive higher scores, improving the model’s ability to focus on relevant terms. For advanced applications, **word embeddings** like Word2Vec or GloVe map words to dense vectors in a continuous vector space, capturing semantic relationships between words.

3.2.6

Which vectorization technique considers the importance of words in a document relative to the entire corpus?

- TF-IDF
- Bag-of-Words

- Word Embeddings
- Sentence Parsing

📖 3.2.7

Vectorization in NLP can be done manually by representing text data numerically without relying on libraries or pre-built functions. The goal remains the same: converting words or sentences into numbers to enable machines to understand and process human language.

Imagine we have a simple vocabulary consisting of the words **["cat," "dog," "mouse"]**. These words form the dimensions of a mathematical space. To manually vectorize a sentence like **"cat and dog"**, we count the occurrences of each word in our vocabulary:

- "cat" appears once.
- "dog" appears once.
- "mouse" does not appear.

The resulting vector for this sentence would be **[1, 1, 0]**. This is known as a **bag-of-words (BoW)** representation, where each value corresponds to the frequency of a word in the sentence.

This process can be expanded to larger vocabularies and longer texts, allowing us to create a structured numerical representation that computers can analyze mathematically.

📝 3.2.8

What does the vector **[1, 1, 0]** represent in a bag-of-words model based on previous example?

- The presence of "cat" and "dog" and the absence of "mouse."
- The frequency of characters in a sentence.
- The semantic meaning of the sentence.
- A numerical representation of grammar rules.

📖 3.2.9

To vectorize any text, the first step is to create a vocabulary, a list of all unique words across the dataset. For instance, consider three sentences:

1. "The cat sat on the mat."
2. "The dog barked at the cat."
3. "The mouse ran away."

From these sentences, we identify all unique words:

["the," "cat," "sat," "on," "mat," "dog," "barked," "at," "mouse," "ran," "away"].

Each word in the vocabulary is assigned a unique position in the vector. When vectorizing a sentence, we count how many times each word from the vocabulary appears in that sentence and place the counts in the corresponding positions.

For the sentence **"The cat sat on the mat,"** the vector becomes **[2, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0]**, where "the" appears twice, and other words like "dog" or "barked" do not appear.

3.2.10

What is the first step in vectorization?

- Building a vocabulary of unique words.
- Counting the frequency of words in each sentence.
- Creating a matrix of all word counts.
- Assigning random numbers to words.

3.2.11

For example, if we wanted to create a virtual assistant we would have to train it in a human way, we would have to load a language dictionary (the easy part) into its memory, find a way to teach it grammar (speech, clause, sentence structure, etc.) and logical interpretation. This is a time-consuming task. However, what if we could transform the sentence into mathematical objects so that the computer could use mathematical or logical operations to make some sense of it? This mathematical construct could be a **vector**, a matrix, and so on. Suppose we had an n-dimensional space where each axis corresponded to a word of our language. This allows us to represent a given sentence as a vector in this space with its coordinate along each axis as the number of words representing that axis.

However, to avoid having to do the **vectorization** process manually, we can use the Python programming language and the *scikit-learn* library. Using the **CountVectorizer()** function, we can vectorize a sentence and get a matrix of vectors representing that sentence.

```
from sklearn.feature_extraction.text import CountVectorizer

document = ["I like computer science","There are many computer softwares","I have an computer with various softwares"]
vectorizer = CountVectorizer()
vectorizer.fit(document)
print("Vocabulary: ", vectorizer.vocabulary_)
vector = vectorizer.transform(document)
print("Vectorized document:")
print(vector.toarray())
```

Program output:

```
Vocabulary:  {'like': 4, 'computer': 2, 'science': 6, 'there': 8, 'are': 1, 'many': 5, 'softwares': 7, 'have': 3, 'an': 0, 'with': 10, 'various': 9}
Vectorized document:
[[0 0 1 0 1 0 1 0 0 0 0]
 [0 1 1 0 0 1 0 1 1 0 0]
 [1 0 1 1 0 0 0 1 0 1 1]]
```

Preparation of Texts

Chapter 4

4.1 Basic tokenization

4.1.1

The dictionary is an important part of several tasks in NLP. A **lexicon** can be defined as the vocabulary of a person, language, or discipline. Roughly speaking, a lexicon can be thought of as a dictionary of terms called **lexemes**. For example, the terms used by doctors can be thought of as the lexicon of their profession. As an example, in an attempt to create an algorithm to convert a physical prescription provided by doctors into an electronic form, lexicons would consist primarily of medical terms. Lexicons are used for a variety of NLP tasks where they are provided as a word list or dictionary.

Before discussing procedures on how to create a lexicon we need to understand phonemes, graphemes and morphemes:

- Phonemes can be thought of as the sound units that can distinguish one word from another in a given language.
- Graphemes are groups of letters of length one or more that can represent these individual sounds or phonemes.
- A morpheme is the smallest unit of meaning in a language.

4.1.2

Which of the following statements about lexicons and language components is true?

- A lexicon is the vocabulary specific to a person, language, or profession.
- Morphemes represent the smallest unit of meaning in a language.
- Graphemes are the smallest units of meaning in a language.
- Phonemes are the written symbols that represent sound units.

4.1.3

When creating a dictionary you must first divide documents or sentences into parts called **tokens**. Each token carries a semantic meaning associated with it. **Tokenization** is one of the basic stages to be performed in any text processing activity. Tokenization can be thought of as a segmentation technique in which we try to divide larger portions of text into smaller meaningful parts. Tokens generally contain words and numbers but can also be extended to include punctuation marks, symbols, and sometimes comprehensible emoticons.

The simplest approach to tokenization is certainly a simple division based on spaces. We can use the **split()** function to do this, which splits a text variable based on a specified separator. By default, the function splits based on space. However, this is a trivial function that may not work properly.

```
sent = "I like computer science"
print(sent.split())
```

Program output:

```
['I', 'like', 'computer', 'science']
```

4.1.4

Which statements about tokenization are correct?

- Tokenization involves dividing larger portions of text into smaller meaningful parts.
- Tokenization is a segmentation technique used in text processing.
- Tokens are limited to words and numbers and cannot include symbols or punctuation marks.
- Tokens always represent complete sentences in a document.

4.1.5

The **split()** function is a basic string operation that divides a sentence into words based on a specified delimiter, such as spaces. However, it may not always produce accurate results in all cases. For example:

- Punctuation handling - in a sentence like "Hello, world!", the split() function will separate "Hello," as a single token, leaving the comma attached to the word. This requires additional steps for proper tokenization, such as removing or handling punctuation separately.
- Compound words - words like "e-mail" or "mother-in-law" might not be treated as single tokens because the hyphen would cause the split function to separate them into parts.
- Whitespace sensitivity - multiple spaces or tabs in a sentence may cause unexpected behavior when splitting into tokens, leading to empty tokens in the result.
- Special characters - symbols, emojis, or non-standard characters may be incorrectly treated, either as part of a token or split into separate elements.

While **split()** provides a quick and simple method for dividing text, more robust approaches, such as regular expressions or NLP tokenization libraries, handle these challenges more effectively.

```
sentence = "Slovakia's capital is Bratislava"
print(sentence.split())
```

Program output:

```
["Slovakia's", 'capital', 'is', 'Bratislava']
```

We can observe that the function does not address the apostrophe and simply takes it as part of the word. This can be a problem, especially in the case of English phrases like *I'm* or *we'll* where it is a contraction of the following word.

```
sentence = "I'm happy to visit Bratislava"
print(sentence.split())
```

Program output:

```
["I'm", 'happy', 'to', 'visit', 'Bratislava']
```

Thus, there are a number of issues that can arise if we only use basic functions. The dots indicating abbreviations or different characters can be a problem. Therefore, in the next section, we will show the different tools that can be used in tokenization.

4.1.6

What is a potential limitation of using the split() function for tokenization?

- It may incorrectly handle punctuation.
- It cannot split words at all.
- It requires a predefined list of tokens.
- It only works for numerical data.

4.1.7

Why might the split() function fail to accurately tokenize a sentence?

- It does not handle punctuation correctly.
- It may produce empty tokens if extra spaces exist.
- It automatically converts all words to lowercase.
- It can only process one sentence at a time.

4.1.8

Given the sentence "It's raining cats and dogs!", what will the split() function output if used without any preprocessing?

- ["It's", "raining", "cats", "and", "dogs!"]
- ["It's", "raining", "cats", "and", "dogs"]
- ["It", "is", "raining", "cats", "and", "dogs"]
- ["It", "is", "raining", "cats", "dogs"]

4.2 Tokenization

4.2.1

Among the popular techniques for tokenization, the use of **regular expressions** stands out as a simple yet powerful method. Regular expressions, often referred to as regex, are sequences of characters that define a search pattern, enabling the identification of specific structures within text. They are versatile and can be used for tasks like splitting sentences into words, extracting specific elements, or filtering data.

For example, if you are searching for email addresses within a block of text, you can leverage regular expressions to define the common pattern of email addresses: a sequence of alphanumeric characters followed by an "@" symbol and a domain name. Instead of relying on advanced machine learning techniques, regular expressions offer a straightforward and rule-based approach to solving such problems.

1. Splitting sentences into words

We can use a regular expression to split a sentence into words while ignoring punctuation marks. This is useful for cleaning up text data before further processing.

Tokenize the sentence: "The quick brown fox, jumps over the lazy dog!"

```
import re
sentence = "The quick brown fox, jumps over the lazy dog!"
tokens = re.findall(r'\b\w+\b', sentence)
print(tokens)
```

Program output:

```
['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the',
'lazy', 'dog']
```

- \b\w+\b identifies word boundaries and extracts words.

2. Extracting email addresses

Consider extracting email addresses from a block of text. Email addresses have a standard pattern that can be matched using a regular expression.

Extract email addresses from: "Contact us at support@example.com or sales@business.org for assistance."

```
text = "Contact us at support@example.com or
sales@business.org for assistance."
emails = re.findall(r'[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-
Z]{2,}', text)
print(emails)
```

Program output:

```
['support@example.com', 'sales@business.org']
```

- This regex identifies the general structure of an email address.

Extracting dates in specific formats

We often encounter dates in different formats, such as dd/mm/yyyy or mm-dd-yyyy. Regular expressions can help us extract these dates accurately.

Extract dates from: "Important events are scheduled on 12/05/2023 and 07-11-2022."

```
text = "Important events are scheduled on 12/05/2023 and 07-
11-2022."
dates = re.findall(r'\b\d{2}[/-]\d{2}[/-]\d{4}\b', text)
print(dates)
```

Program output:

```
['12/05/2023', '07-11-2022']
```

4.2.2

Which of the following regular expressions would correctly extract all words from the sentence "Regular expressions are powerful tools!"?

- \b\w+\b
- \d+
- .*
- \s+

4.2.3

Select the regular expressions that can correctly match a 3-digit number:

- \d{3}
- \b\d{3}\b
- [0-9]+
- [A-Za-z]{3}

4.2.4

Which regular expression is best suited to match email addresses in a given text?

- [a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}
- [a-zA-Z]+@[a-z]+\.[a-z]+
- \w+\s@\s\w+\.\w+
- [0-9]{4}

4.2.5

The Python library **nltk** provides a regular expression-based tokenization function (**RegexpTokenizer**). We can use it to tokenize or split a sentence based on a given regular expression. Consider the following sentence:

```
The average price of computers in the US is between $300 -
$500.
```

We will need expressions denoting money and alphabetic sequences. For this purpose, we can define a regular expression and give the expression to the corresponding tokenizer object.

```
from nltk.tokenize import RegexpTokenizer
s = "The average price of computers in the US is between $300
- $500."
tokenizer = RegexpTokenizer('\w+|\$[\d]+|\S+')
print(tokenizer.tokenize(s))
```

4.2.6

Use the given regular expression to tokenize the given sentences into tokens. As a result, print the output from the tokenize() function.

Given sentences:

```
The wooden spoon couldn’t cut but left emotional scars. She
finally understood that grief was her love with no place for
it to go.
Weather is not trivial - it's especially important when you're
standing in it.
```

For tokenization, use the following regular expression:

```
\w+(?:'\w+)?|[^\w\s]
```

```
from nltk.tokenize import RegexpTokenizer
s = "The wooden spoon couldn’t cut but left emotional scars.
She finally understood that grief was her love with no place
for it to go. Weather is not trivial - it\'s especially
important when you\'re standing in it."
tokenizer = RegexpTokenizer(r"\w+(?:'\w+)?|[^\w\s]")
print(tokenizer.tokenize(s))
```

4.2.7

Use the given regular expression to tokenize the given sentences into tokens. As a result, print the output from the **tokenize()** function.

Given sentences:

```
The wooden spoon couldn’t cut but left emotional scars. She
finally understood that grief was her love with no place for
it to go.
Weather is not trivial - it's especially important when you're
standing in it.
```

For tokenization, use the following regular expression:

```
(\w+|#\d|\?|!)
```

```
from nltk.tokenize import RegexpTokenizer
s = "The wooden spoon couldn’t cut but left emotional scars.
She finally understood that grief was her love with no place
for it to go. Weather is not trivial - it\'s especially
important when you\'re standing in it."
tokenizer = RegexpTokenizer(r"(\w+|#\d|\?|!)")
print(tokenizer.tokenize(s))
```

4.2.8

Another option is to use the **Treebank** tokenizer, which also uses regular expressions to tokenize text according to the Penn Treebank database (https://catalog.ldc.upenn.edu/docs/LDC95T7/cl93.html). Words are mostly split based on punctuation. The Treebank tokenizer does an excellent job of splitting abbreviations such as *doesn't* into *does* and *n't*. Further, it identifies periods at the ends of lines and removes them. Punctuation marks, such as commas, are split if they are followed by spaces. Let's look at the previous sentence and tokenise it using the Treebank tokeniser.

```
from nltk.tokenize import TreebankWordTokenizer
s = "The average price of computers in the US is between $300
- $500."
tokenizer = TreebankWordTokenizer()
print(tokenizer.tokenize(s))
```

Program output:
```
['The', 'average', 'price', 'of', 'computers', 'in', 'the',
'US', 'is', 'between', '$', '300', '-', '$', '500', '.']
```

4.2.9

The development of social media has led to the emergence of an informal language in which people tag each other using their social media accounts and use a variety of emoticons, hashtags and shortened texts to express themselves. Hence, there is a need for tokenization tools that can parse such text and make it more comprehensible. **TweetTokenizer** strongly suits this use case. So let's try to analyze the following tweet:

```
@PassedToMessi Whatever happens on Sunday. Whatever God
decides. Whoever wins. Thank you for everything Leo Messi. You
are truly the greatest ever. With or without a Worldcup. The
Worldcup may just be a reward for your hardwork. <3
```

```
from nltk.tokenize import TweetTokenizer
s = "@PassedToMessi Whatever happens on Sunday. Whatever God
decides. Whoever wins. Thank you for everything Leo Messi. You
are truly the greatest ever. With or without a Worldcup. The
Worldcup may just be a reward for your hardwork. <3"
tokenizer = TweetTokenizer()
print(tokenizer.tokenize(s))
```

Program output:
```
['@PassedToMessi', 'Whatever', 'happens', 'on', 'Sunday', '.',
'Whatever', 'God', 'decides', '.', 'Whoever', 'wins', '.',
'Thank', 'you', 'for', 'everything', 'Leo', 'Messi', '.',
'You', 'are', 'truly', 'the', 'greatest', 'ever', '.', 'With',
'or', 'without', 'a', 'Worldcup', '.', 'The', 'Worldcup',
'may', 'just', 'be', 'a', 'reward', 'for', 'your', 'hardwork',
'.', '<3']
```

4.3 Stemming

4.3.1

Stemming is a technique used in NLP to reduce words to their base or root form. This process attempts to strip affixes (prefixes or suffixes) from words to derive a simplified, common root. For example, when we take the words "computer," "computerization," and "computerize," the stemming process reduces them to the base word "comput." Stemming is used in many NLP tasks, especially when we want to treat different forms of a word as a single unit. It is a crude and straightforward approach where the goal is not to retain the actual meaning of the word, but rather to standardize different forms of a word into a common base.

However, it is important to note that the stem resulting from this process may not always be a valid word in itself. For example, stemming the word "running" could give us the stem "run," which is a valid word, but stemming the word "happiness" might yield the non-existent stem "happi." This can sometimes cause issues, especially when the context or meaning of the word is essential.

Original words: "going", "goes", "went"

- After stemming: "go", "go", "go"
- In this case, stemming reduces all the different forms of the verb "go" to its root "go." While this is useful for certain NLP tasks, such as document clustering or search engines, the context of each word (e.g., tense or aspect) is lost.

Original words: "played", "plays", "playing"

- After stemming: "play", "play", "play"
- The stemming algorithm has correctly reduced all the verb forms to "play," again ignoring tense or other grammatical distinctions.

4.3.2

What is the main purpose of stemming in NLP?

- To reduce different forms of a word to a common base form.
- To convert a word to its most meaningful form.
- To keep all word forms separate.
- To identify synonyms of a word.

4.3.3

Which of the following words would likely be reduced to the same stem during stemming?

- "happiness" and "happy"
- "play" and "played"
- "quickly" and "quicker"
- "running" and "ran"

4.3.4

The two most common algorithms used for stemming are the **Porter stemmer** and the **Snowball stemmer**. The Porter stemmer supports English, while the Snowball stemmer which is an enhancement of the Porter stemmer, supports multiple languages. Porter stemmer works only with strings while Snowball works with strings and also with Unicode data. Snowball stemmer allows you to use a built-in function to ignore stop words.

```
plurals = ['caresses', 'flies', 'dies', 'mules', 'died',
'agreed', 'owned', 'humbled', 'sized', 'meeting', 'stating',
'generously']
from nltk.stem.porter import PorterStemmer
stemmer = PorterStemmer()
singles = [stemmer.stem(plural) for plural in plurals]
print(' '.join(singles))
```

Program output:

```
caress fli die mule die agre own humbl size meet state gener
```

4.3.5

Use the Porter stemmer tool to create a word stem for the input sentence. Remember that sentences must first be tokenized into tokens and thus sent for stemming. As a result, output word stems are separated by spaces.

Given sentences:

```
The wooden spoon couldn’t cut but left emotional scars. She
finally understood that grief was her love with no place for
it to go.
Weather is not trivial - it's especially important when you're
standing in it.
```

```
from nltk.stem.porter import PorterStemmer
from nltk.tokenize import RegexpTokenizer
stemmer = PorterStemmer()
sent = "The wooden spoon couldn’t cut but left emotional
scars. She finally understood that grief was her love with no
place for it to go. Weather is not trivial - it\'s especially
important when you\'re standing in it."
tokenizer = RegexpTokenizer(r"(\w+|#\d|\?|!)")
tokens = tokenizer.tokenize(sent)
output = [stemmer.stem(s) for s in tokens]
print(' '.join(output))
```

4.3.6

Snowball stemmer, unlike the previous one, requires language as a parameter. In most cases, its output is similar to that of the Porter stemmer, except for the word generously, where the Porter stemmer outputs gener and the Snowball stemmer outputs generous. The example shows how Snowball stemmer makes minor changes to Porter's algorithm, achieving improvements in some cases.

```
plurals = ['caresses', 'flies', 'dies', 'mules', 'died',
'agreed', 'owned', 'humbled', 'sized', 'meeting', 'stating',
'generously']
from nltk.stem.snowball import SnowballStemmer
stemmer = SnowballStemmer(language='english')
singles = [stemmer.stem(plural) for plural in plurals]
print(' '.join(singles))
```

Program output:

```
caress fli die mule die agre own humbl size meet state
generous
```

4.3.7

Use the Snowball stemmer tool to create a word stem for the input sentence. Remember that sentences must first be tokenized into tokens and thus sent for stemming. As a result, output word stems are separated by spaces.

Given sentences:

```
The wooden spoon couldn’t cut but left emotional scars. She
finally understood that grief was her love with no place for
it to go.
Weather is not trivial - it's especially important when you're
standing in it.
```

```
from nltk.stem.snowball import SnowballStemmer
from nltk.tokenize import RegexpTokenizer
stemmer = SnowballStemmer(language='english')
sent = "The wooden spoon couldn’t cut but left emotional
scars. She finally understood that grief was her love with no
place for it to go. Weather is not trivial - it\'s especially
important when you\'re standing in it."
tokenizer = RegexpTokenizer(r"(\w+|#\d|\?|!)")
tokens = tokenizer.tokenize(sent)
output = [stemmer.stem(s) for s in tokens]
print(' '.join(output))
```

4.3.8

While the **Porter Stemmer** and **Snowball Stemmer** are two of the most commonly used stemming algorithms, they are not the only ones available. Both stemmers aim to remove affixes from words, but they do so in slightly different ways.

Why do we need more stemmers? The answer lies in the fact that different languages, texts, and NLP tasks require different approaches to stemming. The **Porter Stemmer**, one of the earliest and most widely used algorithms, applies a set of rules for stemming English words. It’s simple and efficient, but it can sometimes produce stems that are not linguistically valid, especially for words with complex affixes.

The **Snowball Stemmer** is a more recent improvement upon the Porter Stemmer. It is designed to be more accurate and flexible. Snowball supports stemming for multiple languages, including English, German, French, and others. It generally produces better results in terms of stemming accuracy because it uses a larger set of rules and more sophisticated algorithms.

However, both stemmers still have limitations. They can be too aggressive, removing parts of words that may still hold meaning (e.g., removing the "s" from "cares" would lead to "care," which might lose the plural nuance). This is where other stemmers like **Lancaster Stemmer** or **Lovins Stemmer** come into play. These stemmers have different approaches that might suit specific tasks better, depending on the language and text.

For example, the **Lancaster Stemmer** is more aggressive than the Porter Stemmer, while the **Lovins Stemmer** is based on a larger dictionary and tries to maintain more meaning in the words it stems. Each of these stemmers is designed to handle specific types of words and language structures more effectively.

Therefore, the need for multiple stemming algorithms arises from the fact that no single stemmer can handle all linguistic intricacies perfectly. Different tasks, like search engine optimization, document classification, or sentiment analysis, may benefit from one stemmer over another based on their specific requirements.

4.3.9

Why do we need different stemming algorithms in NLP?

- Because different languages and NLP tasks require different approaches.
- To make stemming faster.
- To reduce the size of the dataset without losing meaning.
- To apply the same rules to every word in a text.

4.4 Lematization

4.4.1

Lemmatization is a process in NLP that converts words into their meaningful base forms, known as lemmas. Unlike stemming, where words are truncated to a root form, lemmatization ensures that the result is a valid word. This process helps group words with similar meanings into a single item, making it easier for machines to understand. For example, the words "running" and "ran" would both be reduced to their base form "run" in lemmatization.

The key difference between stemming and lemmatization lies in how the words are treated. While stemming uses heuristic rules to remove affixes and often produces non-words, lemmatization takes into account the word's context, meaning, and grammatical role in the sentence. This is why lemmatization is generally preferred over stemming when it comes to maintaining the integrity and meaning of the text.

Lemmatization algorithms, such as the **WordNet Lemmatizer**, are designed to identify the correct lemma form by considering the word's surrounding context. This involves using part-of-speech (POS) tags to determine whether a word is a noun, verb, adjective, or adverb, which helps in selecting the appropriate lemma. For instance, "better" might be reduced to "good" when it functions as an adjective, but it would remain "better" if used as a comparative adjective. Other libraries, like

Spacy, **TextBlob**, and **Gensim**, also integrate lemmatization features into their toolsets.

4.4.2

Which of the following are true about lemmatization?

- Lemmatization uses context, part-of-speech (POS) tags, and meaning to find the base form of a word.
- Lemmatization can lead to different lemmas for the same word depending on the context.
- Lemmatization removes affixes from words to produce non-words.
- Lemmatization always produces a non-dictionary word.

4.4.3

WordNet is an English lexical database that is freely and publicly available. WordNet includes nouns, verbs, adjectives and adverbs grouped into sets of cognitive synonyms (synsets), each expressing different concepts. These synsets are linked to each other by lexical and conceptual semantic relations. It can be easily downloaded and the nltk library offers an interface to it that allows to perform lemmatization.

```
import nltk
nltk.download('wordnet')
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
s = "She was putting efforts to heal her emotionally scarred soul"
token_list = s.split()
print("The tokens are: ", token_list)
lemmatized_output = ' '.join([lemmatizer.lemmatize(token) for token in token_list])
print("The lemmatized output is: ", lemmatized_output)
```

Program output:

```
The tokens are:  ['She', 'was', 'putting', 'efforts', 'to',
'heal', 'her', 'emotionally', 'scarred', 'soul']
The lemmatized output is:  She wa putting effort to heal her
emotionally scarred soul
[nltk_data] Downloading package wordnet to
/home/johny/nltk_data...
```

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

We can observe that the lemmatization was not very successful and most of the words were not converted to lemma.

4.4.4

The reason why the lemmatizer didn't generate the correct lemmas in the previous lesson was that WordNet works better when it also has POS tags for the words in the input. The **nltk** library provides a method to generate POS tags for a list of words. We generate POS tags for a sentence in tuple form using the **pos_tag()** function.

```
import nltk
nltk.download('averaged_perceptron_tagger')
s = "She was putting efforts to heal her emotionally scarred
soul"
token_list = s.split()
pos_tags = nltk.pos_tag(token_list)
print(pos_tags)
```

Program output:

```
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /home/johny/nltk_data...
[nltk_data]   Unzipping
taggers/averaged_perceptron_tagger.zip.
[('She', 'PRP'), ('was', 'VBD'), ('putting', 'VBG'),
('efforts', 'NNS'), ('to', 'TO'), ('heal', 'VB'), ('her',
'PRP$'), ('emotionally', 'RB'), ('scarred', 'JJ'), ('soul',
'NN')]
```

4.4.5

However, in order for WordNet to work with the input we need to convert it to a different type of word type notation. Therefore, we can use the frequently used **get_part_of_speech_tags()** function to convert the tags into the form we need. We can then send the output of the function as a parameter to the lemmatizer.

```
import nltk
nltk.download('averaged_perceptron_tagger')
s = "She was putting efforts to heal her emotionally scarred
soul"
token_list = s.split()
pos_tags = nltk.pos_tag(token_list)
print(pos_tags)
```

Program output:

```
[('She', 'PRP'), ('was', 'VBD'), ('putting', 'VBG'),
('efforts', 'NNS'), ('to', 'TO'), ('heal', 'VB'), ('her',
'PRP$'), ('emotionally', 'RB'), ('scarred', 'JJ'), ('soul',
'NN')]
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /home/johny/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already
up-to-
[nltk_data]       date!
```

```
from nltk.corpus import wordnet
def get_part_of_speech_tags(token):
    tag_dict = {"J": wordnet.ADJ,
                "N": wordnet.NOUN,
                "V": wordnet.VERB,
                "R": wordnet.ADV}
    tag = nltk.pos_tag([token])[0][1][0].upper()
    return tag_dict.get(tag, wordnet.NOUN)
```

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
output = [lemmatizer.lemmatize(token,
get_part_of_speech_tags(token)) for token in token_list]
print(' '.join(output))
```

Program output:

```
She be put effort to heal her emotionally scar soul
```

We can see that the output is much better than in the case of not using POS tags.

4.4.6

Generate the lemmatized text for the specified text. Remember that the text needs to be tokenized first, followed by POS tags, and then lemmatized. There is a function to convert the tags for WordNet needs.

You are given the following text:

```
Analogous terms were later introduced for use of computers in
various fields, such as business informatics, forest
informatics, legal informatics etc. However, these fields have
more to do with digital literacy than with real informatics.
Their name is probably the result of a lack of knowledge of
the true meaning of informatics. Later in the United States,
next absurd term such as computational informatics were
developed, while all informatics is computational by its
nature.
```

```
import nltk
nltk.download('averaged_perceptron_tagger')
# create pos_tags
sentence = "Analogous terms were later introduced for use of
computers in various fields, such as business informatics,
forest informatics, legal informatics etc. However, these
fields have more to do with digital literacy than with real
informatics. Their name is probably the result of a lack of
knowledge of the true meaning of informatics. Later in the
United States, next absurd term such as computational
informatics were developed, while all informatics is
computational by its nature."
token_list = sentence.split()
pos_tags = nltk.pos_tag(token_list)
print(pos_tags)
```

```
# just run this method for tagset transformation
from nltk.corpus import wordnet
def get_part_of_speech_tags(token):
    tag_dict = {"J": wordnet.ADJ,
                "N": wordnet.NOUN,
                "V": wordnet.VERB,
                "R": wordnet.ADV}
    tag = nltk.pos_tag([token])[0][1][0].upper()
    return tag_dict.get(tag, wordnet.NOUN)
```

```
# create the output of lemmatization
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
output = [lemmatizer.lemmatize(token,
get_part_of_speech_tags(token)) for token in token_list]
print(' '.join(output))
```

4.4.7

Generate the lemmatized text for the specified text. Remember that the text needs to be tokenized first, followed by POS tags, and then lemmatized. There is a function to convert the tags for WordNet needs.

You are given the following text:

```
Resources include individual files or an item's data, computer
programs, computer devices and functionality provided by
computer applications. Examples of consumers are computer
users, computer Software and other Hardware on the computer.
```

```
import nltk
nltk.download('averaged_perceptron_tagger')
# create pos_tags
sentence = "Resources include individual files or an item's
data, computer programs, computer devices and functionality
provided by computer applications. Examples of consumers are
computer users, computer Software and other Hardware on the
computer."
token_list = sentence.split()
pos_tags = nltk.pos_tag(token_list)
print(pos_tags)
```

```
# just run this method for tagset transformation
from nltk.corpus import wordnet
def get_part_of_speech_tags(token):
    tag_dict = {"J": wordnet.ADJ,
                "N": wordnet.NOUN,
                "V": wordnet.VERB,
                "R": wordnet.ADV}
    tag = nltk.pos_tag([token])[0][1][0].upper()
    return tag_dict.get(tag, wordnet.NOUN)
```

```
# create the output of lemmatization
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
output = [lemmatizer.lemmatize(token,
get_part_of_speech_tags(token)) for token in token_list]
print(' '.join(output))
```

4.4.8

Generate the lemmatized text for the specified text. Remember that the text needs to be tokenized first, followed by POS tags, and then lemmatized. There is a function to convert the tags for WordNet needs.

You are given the following text:

```
An automated online assistant providing customer service on a
web page, an example of an application where natural language
processing is a major component. Natural language processing
(NLP) is a subfield of linguistics, computer science, and
artificial intelligence concerned with the interactions
between computers and human language, in particular how to
program computers to process and analyze large amounts of
natural language data. Challenges in natural language
processing frequently involve speech recognition, natural
language understanding, and natural-language generation.
```

```
import nltk
nltk.download('averaged_perceptron_tagger')
# create pos_tags
sentence = "An automated online assistant providing customer
service on a web page, an example of an application where
natural language processing is a major component. Natural
language processing (NLP) is a subfield of linguistics,
computer science, and artificial intelligence concerned with
the interactions between computers and human language, in
particular how to program computers to process and analyze
large amounts of natural language data. Challenges in natural
language processing frequently involve speech recognition,
natural language understanding, and natural-language
generation."
token_list = sentence.split()
pos_tags = nltk.pos_tag(token_list)
print(pos_tags)
```

```
# just run this method for tagset transformation
from nltk.corpus import wordnet
def get_part_of_speech_tags(token):
    tag_dict = {"J": wordnet.ADJ,
                "N": wordnet.NOUN,
                "V": wordnet.VERB,
                "R": wordnet.ADV}
    tag = nltk.pos_tag([token])[0][1][0].upper()
    return tag_dict.get(tag, wordnet.NOUN)
```

```
# create the output of lemmatization
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
output = [lemmatizer.lemmatize(token,
get_part_of_speech_tags(token)) for token in token_list]
print(' '.join(output))
```

4.5 Additional features

4.5.1

Stop words in English are words like *a, an, the, in, at,* and so on, which occur frequently in text corpora and don't carry much information in most contexts. These words are generally needed to complete sentences and make them grammatically correct. They are often the most common words in the language and can be filtered out in most NLP tasks, consequently, helping in reducing the vocabulary or search space. There is no single list of stop words available universally and they vary mostly based on use cases. However, a certain list of words is maintained for languages that can be considered language-specific stop words but should be modified based on the problem being solved.

For the use of stop words, there is a *stopwords* module in the **nltk** library that provides a list of English stop words that can be filtered out of the text under study.

```
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
stop = set(stopwords.words('english'))
s = "She was putting efforts to heal her emotionally scarred 
soul"
token_list = s.split()
output = [token for token in token_list if token not in stop]
print(" ".join(output))
```

Program output:

```
She putting efforts heal emotionally scarred soul
[nltk_data] Downloading package stopwords to
/home/johny/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

4.5.2

For the specified text, remove stop words from it. Then print the text without stop words.

Given text:

```
An automated online assistant providing customer service on a
web page, an example of an application where natural language
processing is a major component. Natural language processing
(NLP) is a subfield of linguistics, computer science, and
```

```
artificial intelligence concerned with the interactions
between computers and human language, in particular how to
program computers to process and analyze large amounts of
natural language data. Challenges in natural language
processing frequently involve speech recognition, natural
language understanding, and natural-language generation.
```

```
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
stop = set(stopwords.words('english'))
s = "An automated online assistant providing customer service
on a web page, an example of an application where natural
language processing is a major component. Natural language
processing (NLP) is a subfield of linguistics, computer
science, and artificial intelligence concerned with the
interactions between computers and human language, in
particular how to program computers to process and analyze
large amounts of natural language data. Challenges in natural
language processing frequently involve speech recognition,
natural language understanding, and natural-language
generation."
token_list = s.split()
output = [token for token in token_list if token not in stop]
print(" ".join(output))
```

4.5.3

For the specified text, remove stop words from it. Then print the text without stop words.

Given text:

```
Members of the public have certain rights of access. These
include the right to access documents about the operation of
government departments and documents that are in the
possession of government Ministers or agencies (Freedom of
Information Act 1982). Certain documents are exempt from this,
including (but not limited to) documents detailing Cabinet
deliberations or decisions; Cabinet documents. documents
disclosing trade secrets; Documents disclosing trade secrets
or commercially valuable information.
```

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

```
from nltk.corpus import stopwords
stop = set(stopwords.words('english'))
s = "Members of the public have certain rights of access.
These include the right to access documents about the
operation of government departments and documents that are in
the possession of government Ministers or agencies (Freedom of
Information Act 1982). Certain documents are exempt from this,
including (but not limited to) documents detailing Cabinet
deliberations or decisions; Cabinet documents. documents
disclosing trade secrets; Documents disclosing trade secrets
or commercially valuable information."
token_list = s.split()
output = [token for token in token_list if token not in stop]
print(" ".join(output))
```

4.5.4

Another strategy that helps in normalizing the text is called **Case folding**. It is the unification of uppercase and lowercase with all the letters in the text corpus made lowercase. In several cases, the size of the letters plays a role and hence it is better to have all the words of the same size. This technique helps systems that deal with information retrieval, such as web search engines.

However, in situations where proper nouns are derived from common nouns, the unification of uppercase and lowercase letters becomes a hindrance because case distinction becomes an important feature here. Another problem is when abbreviations are converted to lowercase. There is a high probability that they will map to generic nouns.

A potential solution to this problem is to create machine learning models that can use features from the sentence to determine which words or tokens in the sentence should be lowercase and which should not. However, this approach is not always helpful when users mostly write in lowercase. As a result, writing everything in lowercase becomes the appropriate solution. Therefore, the strings-to-lowercase conversion function **lower()** can be used.

```
s = "She was putting efforts to heal her emotionally scarred
soul"
print(s.lower())
```

Program output:

```
she was putting efforts to heal her emotionally scarred soul
```

4.5.5

For the text you have entered, change the case of the letters to lowercase. Then print the text.

Given text:

```
Members of the public have certain rights of access. These
include the right to access documents about the operation of
government departments and documents that are in the
possession of government Ministers or agencies (Freedom of
Information Act 1982). Certain documents are exempt from this,
including (but not limited to) documents detailing Cabinet
deliberations or decisions; Cabinet documents. documents
disclosing trade secrets; Documents disclosing trade secrets
or commercially valuable information.
```

```
s = "Members of the public have certain rights of access.
These include the right to access documents about the
operation of government departments and documents that are in
the possession of government Ministers or agencies (Freedom of
Information Act 1982). Certain documents are exempt from this,
including (but not limited to) documents detailing Cabinet
deliberations or decisions; Cabinet documents. documents
disclosing trade secrets; Documents disclosing trade secrets
or commercially valuable information."
print(s.lower())
```

4.5.6

Sentences usually contain names of people and places and other open compound expressions, such as *living room* or *coffee mug*. These expressions convey a specific meaning when two or more words are used together. When used alone they carry a completely different meaning and the meaning of compound expressions is somehow lost. Using multiple tokens to represent such meaning can be very beneficial for NLP tasks performed. Although such occurrences are rare they still yield a lot of information. For this reason, we use techniques to preserve the meaning of compound expressions.

In general, these techniques fall under the term **n-grams**. If **n** is equal to **1**, they are referred to as **unigrams**. **Bigrams** or **2-grams** refer to pairs of words, such as *living room*. Phrases such as *United Arab Emirates*, which consist of three words are referred to as **trigrams** or **3-grams**. This naming system can be extended to larger n-grams but in most NLP tasks only trigrams or lower are used.

4.5.7

Let's try working with n-grams in practice. Let's have a sentence to describe what natural language processing is:

```
Natural language processing is an interdisciplinary subfield
of linguistics, computer science, and artificial intelligence
concerned with the interactions between computers and human
language, in particular how to program computers to process
and analyze large amounts of natural language data.
```

Since we know that natural language processing is a domain and processing these three words individually could cause a loss of meaning, we may prefer to use trigrams and preserve the meaning of the words. We can use the **nltk** library module called **ngrams** to create n-grams. The parameters for the function are the tokens of the sentence and the number of n-grams we want to generate.

```
from nltk.util import ngrams
sent = "Natural language processing is an interdisciplinary
subfield of linguistics, computer science, and artificial
intelligence concerned with the interactions between computers
and human language, in particular how to program computers to
process and analyze large amounts of natural language data."
tokens = sent.split()
trigrams = list(ngrams(tokens, 3))
print([" ".join(token) for token in trigrams])
```

Program output:

```
['Natural language processing', 'language processing is',
'processing is an', 'is an interdisciplinary', 'an
interdisciplinary subfield', 'interdisciplinary subfield of',
'subfield of linguistics,', 'of linguistics, computer',
'linguistics, computer science,', 'computer science, and',
'science, and artificial', 'and artificial intelligence',
'artificial intelligence concerned', 'intelligence concerned
with', 'concerned with the', 'with the interactions', 'the
interactions between', 'interactions between computers',
'between computers and', 'computers and human', 'and human
language,', 'human language, in', 'language, in particular',
'in particular how', 'particular how to', 'how to program',
'to program computers', 'program computers to', 'computers to
process', 'to process and', 'process and analyze', 'and
analyze large', 'analyze large amounts', 'large amounts of',
'amounts of natural', 'of natural language', 'natural language
data.']
```

4.5.8

For the given text, generate its unigrams. Print the unigrams.

Given text:

```
Members of the public have certain rights of access. These
include the right to access documents about the operation of
government departments and documents that are in the
possession of government Ministers or agencies (Freedom of
Information Act 1982). Certain documents are exempt from this,
including (but not limited to) documents detailing Cabinet
deliberations or decisions; Cabinet documents. documents
disclosing trade secrets; Documents disclosing trade secrets
or commercially valuable information.
```

```
from nltk.util import ngrams
sent = "Members of the public have certain rights of access.
These include the right to access documents about the
operation of government departments and documents that are in
the possession of government Ministers or agencies (Freedom of
Information Act 1982). Certain documents are exempt from this,
including (but not limited to) documents detailing Cabinet
deliberations or decisions; Cabinet documents. documents
disclosing trade secrets; Documents disclosing trade secrets
or commercially valuable information."
tokens = sent.split()
unigrams = list(ngrams(tokens, 1))
print([" ".join(token) for token in unigrams])
```

4.5.9

For the given text, generate its bigrams. Print the bigrams.

Given text:

```
Documents are also distinguished from "realia", which are
three-dimensional objects that would otherwise satisfy the
definition of "document" because they memorialize or represent
thought; documents are considered more as 2-dimensional
representations. While documents can have large varieties of
```

```
customization, all documents can be shared freely and have the
right to do so, creativity can be represented by documents,
also. History, events, examples, opinions, etc. all can be
expressed in documents.
```

```
from nltk.util import ngrams
sent = 'Documents are also distinguished from "realia", which
are three-dimensional objects that would otherwise satisfy the
definition of "document" because they memorialize or represent
thought; documents are considered more as 2-dimensional
representations. While documents can have large varieties of
customization, all documents can be shared freely and have the
right to do so, creativity can be represented by documents,
also. History, events, examples, opinions, etc. all can be
expressed in documents.'
tokens = sent.split()
bigrams = list(ngrams(tokens, 2))
print([" ".join(token) for token in bigrams])
```

4.5.10

For the given text, generate its trigrams. Print the trigrams.

Given text:

```
The Philip R. Lee Institute for Health Policy Studies is a
partner in this consortium. UCSF is home to the Industry
Documents Library (IDL), a digital library of previously
secret internal industry documents, including over 14 million
documents in the internationally known Truth Tobacco Industry
Documents, the Food Industry Documents Archive, Chemical
Industry Documents Archive and the Drug Industry Documents
Archive. The IDL contains millions of documents created by
major companies related to their advertising, manufacturing,
marketing, sales, and scientific research activities.
```

```
from nltk.util import ngrams
sent = 'The Philip R. Lee Institute for Health Policy Studies
is a partner in this consortium. UCSF is home to the Industry
Documents Library (IDL), a digital library of previously
secret internal industry documents, including over 14 million
documents in the internationally known Truth Tobacco Industry
Documents, the Food Industry Documents Archive, Chemical
```

```
Industry Documents Archive and the Drug Industry Documents
Archive. The IDL contains millions of documents created by
major companies related to their advertising, manufacturing,
marketing, sales, and scientific research activities.'
tokens = sent.split()
trigrams = list(ngrams(tokens, 3))
print([" ".join(token) for token in trigrams])
```

Sequence-to-Sequence Model

Chapter 5

5.1 Introduction

5.1.1

In this lesson, we will introduce the **Sequence-to-Sequence (Seq2Seq)** model, which is crucial in tasks such as machine translation, where we need to map one sequence of words to another sequence. The Seq2Seq model uses two primary components: an **encoder** and a **decoder**. The encoder processes the input sequence and transforms it into a fixed-size context vector, which contains the essential information from the input. The decoder then takes this context vector and generates the output sequence.

One key challenge in machine translation is that the input and output sequences often have different lengths. For example, the English sentence "How are you doing?" can be translated into Slovak as "Ako sa máš?" Even though both sentences convey the same meaning, the number of words differs. Similarly, the sentence "Can we do this?" in English translates to "Môžeme ísť na to?" in Slovak, where the English and Slovak sentences contain the same number of words, but the structure and length of the translation differ.

To address this, the **encoder** reads the input sentence word by word and compresses it into a single context vector, which encapsulates the semantic meaning of the entire input sequence. The **decoder** then uses this context vector to generate the output sequence, one word at a time. By combining the encoder and decoder, Seq2Seq models can successfully handle input and output sequences of varying lengths, making them a powerful tool for tasks like machine translation.

5.1.2

What is the primary purpose of an encoder in a Sequence-to-Sequence (Seq2Seq) model?

- To convert an input sequence into a fixed-size context vector.
- To generate the output sequence from a context vector.
- To handle variations in the length of the input sequence.
- To learn word embeddings from the input sequence.

5.1.3

In the Seq2Seq model, the **encoder** is the first crucial component in the architecture. Its role is to process the input data and create a **context vector -** a low-dimensional representation that captures the meaning of the input sequence. This context vector, sometimes referred to as the embedding of the input, is designed to encapsulate all the essential information needed for generating an output sequence.

The encoder is commonly implemented using different types of neural networks, with **Recurrent Neural Networks (RNNs)** being a popular choice. RNNs are well-suited for processing sequences because they maintain a **hidden state** that stores information about the input seen at each time step. As the RNN processes each word in the input sequence, its hidden state is updated. The last hidden state of the RNN, after processing the final word of the sentence, is considered the context vector. This last hidden state is crucial because it has "seen" all the words in the input and preserved their context. The context vector therefore contains a compressed version of the entire sentence’s meaning.

The context vector produced by the encoder has two key components:

1. **The hidden state from the last time step** of the encoder, which represents the processed information from the entire input sequence.
2. **The neural network memory state** that contains additional details about the input sentence.

5.1.4

Which of the following are components of the context vector generated by the encoder?

- The last hidden state of the encoder.
- The neural network memory state of the input sentence.
- The initial hidden state of the decoder.
- The output sequence generated by the decoder.
- The input data itself.

5.1.5

After obtaining the **context vector** from the encoder, the next step in a Seq2Seq model is to feed this vector into the **decoder**, which generates the output sequence—often a translation of the input sentence. In recurrent neural networks (RNNs), the decoder’s job is to predict the words of the output sentence, step by step, based on the context provided by the encoder.

The decoder's behavior differs slightly during the **training** and **inference** phases. For both, the context vector generated by the encoder is used as the **initial hidden state** of the decoder. Unlike standard RNNs, where the hidden state is initialized randomly, the decoder's initial hidden state is specifically the context vector. This vector contains all the necessary information that the decoder needs to start generating the output sequence.

At the first time step of the decoding process, the input to the decoder is a special token, often referred to as **<start>**. This token signals the beginning of the output sentence. The decoder then uses the context vector and this **<start>** token to predict the first word of the target sentence. Once the first word is predicted, it is fed back into the decoder as the input for the next time step, and the process continues until the entire output sentence is generated.

5.1.6

Which of the following statements about the decoder in a Seq2Seq model are true?

- The decoder receives the context vector as its initial hidden state.
- The input to the decoder at the first time step is the <|start> token.
- The decoder's task is to generate the input sequence.
- The context vector is generated by the decoder itself.
- The decoder outputs are used to generate the input sequence for the encoder.

5.1.7

The **training** and **inference phase** involve different behaviors for the decoder, even though both rely on the same general structure of using a context vector and an initial token.

During the **training phase**, the decoder is given both the context vector (from the encoder) and the actual target sequence (the correct output) at each time step. The input at the first time step is the **<start>** token, and the decoder learns to predict the first token of the target sequence. At each subsequent time step, the decoder is provided with the true token from the target sequence (the correct word), not its previous prediction. This allows the model to learn the correct sequence of tokens directly. The decoder's task during training is to learn the mapping from the context vector and the **<start>** token to the entire output sequence.

On the other hand, during the **inference phase**, the true target sequence is unavailable. The model's task is to predict the sequence from scratch. At time step 0, the decoder is given the context vector and the **<start>** token, just like in training. However, instead of being provided with the true tokens for subsequent time steps, the decoder is fed its own previous predictions. The output of each time step becomes the input to the next, which can lead to cumulative errors if the decoder makes an incorrect prediction early in the process. Despite this, the goal of the decoder during inference is to generate the correct sequence, one token at a time, using the context vector and its previous predictions.

In summary, during training, the decoder has access to the true target sequence to learn the mappings accurately, while during inference, it must generate the output sequence based on its own predictions.

5.1.8

Which of the following statements about the decoder during the training and inference phases are correct?

- During training, the decoder is given the true previous word from the target sequence.
- During inference, the decoder uses the output from the previous time step as input for generating the next token.
- During inference, the decoder is given the true word from the previous time step.
- During training, the decoder learns to predict the correct output sequence by being given both the context vector and the true target sequence.

5.1.9

In Seq2Seq models, once the decoder starts generating tokens, we need a clear stopping condition to prevent it from continuing indefinitely. There are two common stopping criteria used to determine when the decoder should halt its sequence generation:

1. **<end> token** is a special token marks the end of a sentence or sequence. Whenever the decoder generates this token as part of the predicted sequence, it signals that the sequence is complete, and no further tokens need to be predicted. This is crucial for generating sequences that vary in length, as the model can produce shorter outputs without requiring a fixed sequence length.
2. **Maximum length** - in some cases, we impose a limit on the number of tokens the decoder can generate, even if the <end> token hasn't appeared. This predefined maximum length ensures that the model doesn't generate excessively long sequences that might not be useful, especially in applications like machine translation where sentences usually have a logical endpoint.

These stopping conditions ensure that the decoder produces meaningful and appropriately sized output sequences, whether it's translating a sentence, generating a chatbot response, or transcribing speech. The use of the **<end> token** and a **maximum output length** is an essential part of controlling the sequence generation process.

5.1.10

Which of the following stopping conditions are commonly used in Seq2Seq models?

- The decoder stops if the model reaches a maximum predefined output length.
- The decoder stops when the <|end> token is generated, marking the end of the sequence.
- The decoder will stop once it generates a token indicating the start of the sequence.
- The decoder will always generate a fixed-length output regardless of the content.

5.2 Machine translation - text preparation

5.2.1

In the following sections, we will describe how to create a machine translation using Seq2Seq modelling. We will focus on translation from the Slovak language to the English language. We will use a dataset from http://www.manythings.org/anki/, which contains about 11 000 sentences and their translations into English. In addition to the **nltk** library, we will also need the **tensorflow** and **keras** libraries to help us train our model. Next, we will need the **pandas** and **re** libraries to help us prepare the dataset. In the following microlessons, we'll go through the step-by-step process of how to build the model using mainly custom functions.

```
# Importing necessary libraries for data processing and
modeling
# pandas is used for data manipulation and analysis,
especially for working with structured data like DataFrames
import pandas as pd
# string provides common string operations such as constants
for punctuation and string manipulation
import string
```

```
# re provides regular expression matching operations, useful
for pattern matching in strings (e.g., cleaning text)
import re
# urlopen is used to fetch content from a URL, helpful for
reading data from online resources
from urllib.request import urlopen
# numpy is a package for numerical computing, providing
support for large, multi-dimensional arrays and matrices
import numpy as np
# unicodedata is used to work with Unicode characters,
particularly for normalization (e.g., removing accents)
from unicodedata import normalize
# keras is a high-level neural networks API built on top of
TensorFlow, used for building deep learning models
import keras, tensorflow
# Model class from Keras is used to define and manage the
architecture of neural networks
from keras.models import Model
# LSTM (Long Short-Term Memory) layer from Keras is a type of
Recurrent Neural Network (RNN) useful for sequence prediction
tasks
from keras.layers import Input, LSTM, Dense
```

5.2.2

The first step will be to load an input file to help us train our model. We load the input file line by line into a DataFrame structure using the **pandas** library that acts like a table and will help us to access the individual data in the file. Once the file is loaded, we can examine the created DataFrame to have a better idea of what it contains.

```
# Import the pandas library, typically used for data
manipulation and analysis.
import pandas as pd
# Import the urlopen function from urllib.request to open and
retrieve data from a URL.
from urllib.request import urlopen
# Import the entire urllib module, which provides a set of
functions for working with URLs.
import urllib
```

```
# Define a function to read and process a file from a URL
def input_file(file_name):
    # Initialize an empty list to store the file data
    data = []
    # Open the file from the URL using urllib's urlopen
function
    file = urllib.request.urlopen(file_name)
    # Iterate through each line in the file
    for row in file:
        # Decode each row from bytes to string using UTF-8
encoding
        row = row.decode("utf-8")
        # Remove any leading or trailing whitespace characters
(like newline)
        row = row.strip()
        # Append the cleaned row (line) to the data list
        data.append(row)
    # Return the list of processed data (all lines in the
file)
    return data

# Call the function input_file with the URL of the file and
store the result in 'data'
data =
input_file('https://priscilla.fitped.eu/data/nlp/slk.txt')
# Print a small slice (lines 1500 to 1510) of the data for
inspection
print(data[1500:1510])
# Print the total number of lines in the data
print(len(data))
# Limit the data to the first 10,000 lines for further
processing (if needed)
data = data[:10000]
```

Program output:

```
["I've heard that.\tPočul som to.\tCC-BY 2.0 (France)
Attribution: tatoeba.org #2248400 (CK) & #9846917
(Dominika7)", "I've heard that.\tPočula som to.\tCC-BY 2.0
(France) Attribution: tatoeba.org #2248400 (CK) & #9846918
(Dominika7)", 'Is he breathing?\tDýcha?\tCC-BY 2.0 (France)
Attribution: tatoeba.org #239892 (CK) & #8957974 (Dominika7)',
'Is it poisonous?\tJe to jedovaté?\tCC-BY 2.0 (France)
Attribution: tatoeba.org #2248466 (CK) & #10033911
(Dominika7)', 'Is it poisonous?\tJe jedovatý?\tCC-BY 2.0
(France) Attribution: tatoeba.org #2248466 (CK) & #10033914
(Dominika7)', 'Is it poisonous?\tJe jedovatá?\tCC-BY 2.0
(France) Attribution: tatoeba.org #2248466 (CK) & #10033916
(Dominika7)', 'Is she a doctor?\tJe lekárka?\tCC-BY 2.0
(France) Attribution: tatoeba.org #312527 (CK) & #9734240
(Dominika7)', 'Is this a river?\tJe toto rieka?\tCC-BY 2.0
(France) Attribution: tatoeba.org #56259 (CK) & #4642167
(Sim)', 'Is this my wine?\tTo je moje víno?\tCC-BY 2.0
(France) Attribution: tatoeba.org #1764491 (CK) & #10086221
(Dominika7)', 'Is today Monday?\tDnes je pondelok?\tCC-BY 2.0
(France) Attribution: tatoeba.org #2248648 (CK) & #9948296
(Dominika7)']
11550
```

We can see that the dataset contained more than 11 thousand sentences. So let's take the first 10 000 sentences from the dataset, which will be used for training, and keep the rest as a test set on which we will then test our model.

5.2.3

With the input file, we saw that the sentences are separated by a tab (\t), so we can very easily use the **split()** function to split the sentences into Slovak and English.

```
def create_english_slovak_sentences(data):
  # Initialize empty lists to store English and Slovak
sentences
  EN_sentences = []
  SK_sentences = []
  # Iterate over each data point in the 'data' list
  for data_point in data:
    # Split each data point by tab ('\t'), where the first
part is English and the second part is Slovak
    EN_sentences.append(data_point.split("\t")[0])  # Add the
English sentence to list
    SK_sentences.append(data_point.split("\t")[1])  # Add the
Slovak sentence to list
  # Return both lists containing English and Slovak sentences
  return EN_sentences, SK_sentences

# Calling the function with 'data' and storing the result in
the variables EN_sentences and SK_sentences
```

```
EN_sentences, SK_sentences =
create_english_slovak_sentences(data)
```

5.2.4

Once we have the sentences divided, we can proceed to the essential part namely the preprocessing of the texts. The goal of this feature is to remove unnecessary characters from sentences, such as punctuation or special characters. The next step is to unify the case of the letters with all words starting with a lowercase letter. The result will be a preprocessed sentence cleaned of unnecessary characters.

```
def preprocess_sentences(sentence):
  # Create a regular expression to match non-printable
characters
  re_print = re.compile('[^%s]' % re.escape(string.printable))
  # Create a translation table to remove punctuation
  table = str.maketrans('', '', string.punctuation)
  # Normalize the sentence to remove accents (NFD
normalization) and ignore characters that cannot be
represented in ASCII
  cleaned_sent = normalize('NFD', sentence).encode('ascii',
'ignore')
  # Decode the cleaned sentence back into UTF-8
  cleaned_sent = cleaned_sent.decode('UTF-8')
  # Split the sentence into words (tokens)
  cleaned_sent = cleaned_sent.split()
  # Convert all words to lowercase
  cleaned_sent = [word.lower() for word in cleaned_sent]
  # Remove punctuation from each word using the translation
table
  cleaned_sent = [word.translate(table) for word in
cleaned_sent]
  # Remove non-printable characters from the words using the
regular expression
  cleaned_sent = [re_print.sub('', w) for w in cleaned_sent]
  # Keep only alphabetic words (words containing only letters)
  cleaned_sent = [word for word in cleaned_sent if
word.isalpha()]
  # Join the words back into a single sentence
  return ' '.join(cleaned_sent)
```

5.2.5

Once we have prepared the sentence preprocessing function, we can apply it to our English and Slovak sentences.

```
def preprocess_EN_SK_sentences(EN_sentences, SK_sentences):
  # Initialize empty lists to store the cleaned sentences
  SK_sentences_cleaned = []
  EN_sentences_cleaned = []

  # Iterate through each Slovak sentence in SK_sentences
  for sent in SK_sentences:
    # Preprocess each Slovak sentence and append to the
cleaned list
    SK_sentences_cleaned.append(preprocess_sentences(sent))
  # Iterate through each English sentence in EN_sentences
  for sent in EN_sentences:
    # Preprocess each English sentence and append to the
cleaned list
    EN_sentences_cleaned.append(preprocess_sentences(sent))
  # Return the cleaned English and Slovak sentences
  return EN_sentences_cleaned, SK_sentences_cleaned

# Calling the function to preprocess both the English and
Slovak sentences
EN_sentences_cleaned, SK_sentences_cleaned =
preprocess_EN_SK_sentences(EN_sentences, SK_sentences)
```

5.2.6

The next phase is important because it's where we'll be creating our own dictionary. The goal will also be to obtain tokens that mark the beginning and end of the sequence, as required by the decoder. When we covered dictionary creation in previous lessons we dealt with it at the word level. In this case, we'll go one level lower and work at the character level. We'll place a tab at the beginning of our sequence and a newline label at the end. We'll also prepare a list of unique input and output characters. Our model will then attempt to predict at the character level.

```
def build_data(EN_sentences_cleaned, SK_sentences_cleaned):
  # Initialize empty lists for storing the input and target
datasets
  input_dataset = []
  target_dataset = []
  # Initialize sets to store unique characters from the input
(Slovak) and target (English) sentences
  input_characters = set()
  target_characters = set()

  # Iterate over the cleaned Slovak sentences to create the
input dataset
  for SK_sentence in SK_sentences_cleaned:
    input_datapoint = SK_sentence
    input_dataset.append(input_datapoint)  # Add the Slovak
sentence to the input dataset
    for char in input_datapoint:
      input_characters.add(char)  # Add each character of the
Slovak sentence to the set of input characters

  # Iterate over the cleaned English sentences to create the
target dataset
  for EN_sentence in EN_sentences_cleaned:
    target_datapoint = "\t" + EN_sentence + "\n"  # Add start-
of-sequence and end-of-sequence tokens
    target_dataset.append(target_datapoint)  # Add the English
sentence to the target dataset
    for char in target_datapoint:
      target_characters.add(char)  # Add each character of the
English sentence to the set of target characters

  # Return the input and target datasets, and sorted lists of
unique characters from both datasets
  return input_dataset, target_dataset,
sorted(list(input_characters)),
sorted(list(target_characters))

# Calling the function to generate the datasets and characters
input_dataset, target_dataset, input_characters,
target_characters = build_data(EN_sentences_cleaned,
SK_sentences_cleaned)
```

5.2.7

The following code will serve as a revision of the functions already created. So let's take a look at what the dictionary we generated looks like. Run the individual code blocks in order. Your task is to print what the **input character** list looks like.

```
import pandas as pd
import string
import re
from urllib.request import urlopen
import numpy as np
from unicodedata import normalize
import urllib
```

```
def input_file(file_name):
  data = []
  file = urllib.request.urlopen(file_name)
  for row in file:
    row = row.decode("utf-8")
    row = row.strip()
    data.append(row)
  return data

data =
input_file('https://priscilla.fitped.eu/data/nlp/slk.txt')
print(data[1500])
print(len(data))
data = data[:10000]
```

```
def create_english_slovak_sentences(data):
  EN_sentences = []
  SK_sentences = []
  for data_point in data:
    EN_sentences.append(data_point.split("\t")[0])
    SK_sentences.append(data_point.split("\t")[1])
  return EN_sentences, SK_sentences

EN_sentences, SK_sentences =
create_english_slovak_sentences(data)
```

```
def preprocess_sentences(sentence):
  re_print = re.compile('[^%s]' % re.escape(string.printable))
  table = str.maketrans('', '', string.punctuation)
  cleaned_sent = normalize('NFD', sentence).encode('ascii', 
'ignore')
  cleaned_sent = cleaned_sent.decode('UTF-8')
  cleaned_sent = cleaned_sent.split()
  cleaned_sent = [word.lower() for word in cleaned_sent]
  cleaned_sent = [word.translate(table) for word in 
cleaned_sent]
  cleaned_sent = [re_print.sub('', w) for w in cleaned_sent]
  cleaned_sent = [word for word in cleaned_sent if 
word.isalpha()]
  return ' '.join(cleaned_sent)
```

```
def preprocess_EN_SK_sentences(EN_sentences, SK_sentences):
  SK_sentences_cleaned = []
  EN_sentences_cleaned = []
  for sent in SK_sentences:
    SK_sentences_cleaned.append(preprocess_sentences(sent))
  for sent in EN_sentences:
    EN_sentences_cleaned.append(preprocess_sentences(sent))
  return EN_sentences_cleaned, SK_sentences_cleaned

EN_sentences_cleaned, SK_sentences_cleaned = 
preprocess_EN_SK_sentences(EN_sentences, SK_sentences)
```

```
def build_data(EN_sentences_cleaned, SK_sentences_cleaned):
  input_dataset = []
  target_dataset = []
  input_characters = set()
  target_characters = set()

  for SK_sentence in SK_sentences_cleaned:
    input_datapoint = SK_sentence
    input_dataset.append(input_datapoint)
    for char in input_datapoint:
      input_characters.add(char)

  for EN_sentence in EN_sentences_cleaned:
    target_datapoint = "\t" + EN_sentence + "\n"
    target_dataset.append(target_datapoint)
    for char in target_datapoint:
      target_characters.add(char)

  return input_dataset, target_dataset, 
sorted(list(input_characters)), 
sorted(list(target_characters))
```

```
input_dataset, target_dataset, input_characters,
target_characters = build_data(EN_sentences_cleaned,
SK_sentences_cleaned)
```

```
# write your code here
print(input_characters)
```

5.2.8

The following code will serve as a revision of the functions already created. So let's take a look at what the dictionary we generated looks like. Run the individual code blocks in order. Your task is to write out what the list of **output characters** looks like.

```
import pandas as pd
import string
import re
from urllib.request import urlopen
import numpy as np
from unicodedata import normalize
import urllib
```

```
def input_file(file_name):
  data = []
  file = urllib.request.urlopen(file_name)
  for row in file:
    row = row.decode("utf-8")
    row = row.strip()
    data.append(row)
  return data

data =
input_file('https://priscilla.fitped.eu/data/nlp/slk.txt')
print(data[1500])
print(len(data))
data = data[:10000]
```

```
def create_english_slovak_sentences(data):
  EN_sentences = []
  SK_sentences = []
  for data_point in data:
    EN_sentences.append(data_point.split("\t")[0])
    SK_sentences.append(data_point.split("\t")[1])
  return EN_sentences, SK_sentences

EN_sentences, SK_sentences =
create_english_slovak_sentences(data)
```

```
def preprocess_sentences(sentence):
  re_print = re.compile('[^%s]' % re.escape(string.printable))
  table = str.maketrans('', '', string.punctuation)
  cleaned_sent = normalize('NFD', sentence).encode('ascii',
'ignore')
  cleaned_sent = cleaned_sent.decode('UTF-8')
  cleaned_sent = cleaned_sent.split()
  cleaned_sent = [word.lower() for word in cleaned_sent]
  cleaned_sent = [word.translate(table) for word in
cleaned_sent]
  cleaned_sent = [re_print.sub('', w) for w in cleaned_sent]
  cleaned_sent = [word for word in cleaned_sent if
word.isalpha()]
  return ' '.join(cleaned_sent)
```

```
def preprocess_EN_SK_sentences(EN_sentences, SK_sentences):
  SK_sentences_cleaned = []
  EN_sentences_cleaned = []
  for sent in SK_sentences:
    SK_sentences_cleaned.append(preprocess_sentences(sent))
  for sent in EN_sentences:
    EN_sentences_cleaned.append(preprocess_sentences(sent))
  return EN_sentences_cleaned, SK_sentences_cleaned

EN_sentences_cleaned, SK_sentences_cleaned =
preprocess_EN_SK_sentences(EN_sentences, SK_sentences)
```

```
def build_data(EN_sentences_cleaned, SK_sentences_cleaned):
  input_dataset = []
  target_dataset = []
  input_characters = set()
  target_characters = set()

  for SK_sentence in SK_sentences_cleaned:
    input_datapoint = SK_sentence
    input_dataset.append(input_datapoint)
    for char in input_datapoint:
      input_characters.add(char)
  for EN_sentence in EN_sentences_cleaned:
    target_datapoint = "\t" + EN_sentence + "\n"
    target_dataset.append(target_datapoint)
    for char in target_datapoint:
      target_characters.add(char)

  return input_dataset, target_dataset, 
sorted(list(input_characters)),
sorted(list(target_characters))

input_dataset, target_dataset, input_characters, 
target_characters = build_data(EN_sentences_cleaned, 
SK_sentences_cleaned)
```

```
# write your code here
print(target_characters)
```

5.3 Machine translation - model creation

5.3.1

The results of the previous assignments show us the difference in that we have added escape sequences to the output characters indicating the beginning and end of our sequence, so the **\t** and **\n** tokens are also there. These are used for the decoder to better understand the beginning and end of the sequence. Our input and output dictionaries need not be the same for tasks such as natural language translation. In fact, sometimes even our character set may not be the same. For example, we may be trying to translate between English and Arabic, which have completely different character sets. In addition to the differences in vocabulary, we should also be aware that our input sequence and the target sequence may not be the same size. Not only the number of words in two parallel sentences may be different but also the number of characters in each word. Therefore, we need to get information about the metadata of our sentences:

- the size of the input and output vocabulary,
- the maximum length of the input and output character set.

```
def get_metadata(input_dataset, target_dataset, 
input_characters, target_characters):
  num_Encoder_tokens = len(input_characters)
  num_Decoder_tokens = len(target_characters)
  max_Encoder_seq_length = max([len(data_point) for data_point 
in input_dataset])
```

```
  max_Decoder_seq_length = max([len(data_point) for data_point
in target_dataset])
  print('Number of data points:', len(input_dataset))
  print('Number of unique input tokens:', num_Encoder_tokens)
  print('Number of unique output tokens:', num_Decoder_tokens)
  print('Maximum sequence length for inputs:',
max_Encoder_seq_length)
  print('Maximum sequence length for outputs:',
max_Decoder_seq_length)
  return num_Encoder_tokens, num_Decoder_tokens,
max_Encoder_seq_length, max_Decoder_seq_length

num_Encoder_tokens, num_Decoder_tokens,
max_Encoder_seq_length, max_Decoder_seq_length =
get_metadata(input_dataset, target_dataset, input_characters,
target_characters)
```

Number of data points: 10000

Number of unique input tokens: 26

Number of unique output tokens: 29

Maximum sequence length for inputs: 50

Maximum sequence length for outputs: 38

5.3.2

Number of data points: 10000

Number of unique input tokens: 26

Number of unique output tokens: 29

Maximum sequence length for inputs: 50

Maximum sequence length for outputs: 38

In the previous microlesson, we got information about our metadata. We discovered the following:

- there are 10 000 unique English-Slovak sentence pairs in our dataset,
- the number of unique input tokens (characters) is 26,

- the number of unique output tokens (characters) that we try to extract and predict is 29,
- our longest input sequence is 50 characters long,
- our longest output sequence is 38 characters long.

5.3.3

A very important step is to create a mapping from characters to indexes and vice versa. This will help us in the following activities:

- represent our input characters using the appropriate indices,
- convert our predicted indices to their corresponding characters when predicting.

```
def create_indices(input_characters, target_characters):
  # Initialize dictionaries to store mappings between
characters and indices
  input_char_to_idx = {}  # Maps characters from the input
language (Slovak) to indices
  input_idx_to_char = {}  # Maps indices to characters for the
input language
  target_char_to_idx = {}  # Maps characters from the target
language (English) to indices
  target_idx_to_char = {}  # Maps indices to characters for
the target language

  # Create the mappings for the input characters (Slovak)
  for i, char in enumerate(input_characters):
    input_char_to_idx[char] = i  # Assign an index to each
character
    input_idx_to_char[i] = char  # Create a reverse mapping
from index to character

  # Create the mappings for the target characters (English)
  for i, char in enumerate(target_characters):
    target_char_to_idx[char] = i  # Assign an index to each
character
    target_idx_to_char[i] = char  # Create a reverse mapping
from index to character

  # Return all four dictionaries
  return input_char_to_idx, input_idx_to_char,
target_char_to_idx, target_idx_to_char
```

```
# Create the indices for the input and target languages
input_char_to_idx, input_idx_to_char, target_char_to_idx,
target_idx_to_char = create_indices(input_characters,
target_characters)
```

5.3.4

We can then build our data structure based on the extracted metadata from the previous microlessons.

```
def build_data_structures(length_input_dataset,
max_Encoder_seq_length, max_Decoder_seq_length,
num_Encoder_tokens, num_Decoder_tokens):
  Encoder_input_data = np.zeros((length_input_dataset,
max_Encoder_seq_length, num_Encoder_tokens), dtype='float32')
  Decoder_input_data = np.zeros((length_input_dataset,
max_Decoder_seq_length, num_Decoder_tokens), dtype='float32')
  Decoder_target_data = np.zeros((length_input_dataset,
max_Decoder_seq_length, num_Decoder_tokens), dtype='float32')
  print("Dimensionality of Encoder input data is : ",
Encoder_input_data.shape)
  print("Dimensionality of Decoder input data is : ",
Decoder_input_data.shape)
  print("Dimensionality of Decoder target data is : ",
Decoder_target_data.shape)
  return Encoder_input_data, Decoder_input_data,
Decoder_target_data

Encoder_input_data, Decoder_input_data, Decoder_target_data =
build_data_structures(len(input_dataset),
max_Encoder_seq_length, max_Decoder_seq_length,
num_Encoder_tokens, num_Decoder_tokens)
```

```
Dimensionality of Encoder input data is : (10000, 50, 26)

Dimensionality of Decoder input data is : (10000, 38, 29)

Dimensionality of Decoder target data is : (10000, 38, 29)
```

5.3.5

Dimensionality of Encoder input data is : (10000, 50, 26)

Dimensionality of Decoder input data is : (10000, 38, 29)

Dimensionality of Decoder target data is : (10000, 38, 29)

Let's look at the properties of the data structure we have created:

- the dimension of the input data is (10000, 50, 26),
- the first dimension represents the number of data points: 10 000,
- the second dimension represents the maximum length of our input sequence: 50,
- the third dimension represents the size of our input character set: 26,
- the dimension of the decoder input and output data is (10000, 38, 29),
- the first dimension represents the number of data points: 10 000,
- the second dimension represents the maximum length of our output sequence: 38,
- the third dimension represents the size of our output character set: 29,

Once we have created the data structure we add data to it.

```
def add_data_to_data_structures(input_dataset, target_dataset,
Encoder_input_data, Decoder_input_data, Decoder_target_data):
  # Loop over the input and target datasets
  for i, (input_data_point, target_data_point) in
enumerate(zip(input_dataset, target_dataset)):

    # Fill in the Encoder input data
    for t, char in enumerate(input_data_point):
      # Set the appropriate index in the Encoder input data to
1. This represents the one-hot encoding for the input
character.
      Encoder_input_data[i, t, input_char_to_idx[char]] = 1.

    # Fill in the Decoder input and target data
    for t, char in enumerate(target_data_point):
      # Set the appropriate index in the Decoder input data to
1. This represents the one-hot encoding for the target
character.
      Decoder_input_data[i, t, target_char_to_idx[char]] = 1.
      # For the Decoder target data, set the appropriate index
for the previous character (to align with the teacher forcing
method).
      if t > 0:
        Decoder_target_data[i, t - 1,
target_char_to_idx[char]] = 1.

  return Encoder_input_data, Decoder_input_data,
Decoder_target_data

# Calling the function to fill the data structures
Encoder_input_data, Decoder_input_data, Decoder_target_data =
add_data_to_data_structures(input_dataset, target_dataset,
Encoder_input_data, Decoder_input_data, Decoder_target_data)
```

5.3.6

We used a character-to-index mapping and converted some of the entries in our data structure to 1, indicating the presence of a particular character at a particular position in each of the sentences. As a result of the mapping, the last dimension (26 in the encoder input data structure and 29 in the decoder input or target data structure) is a vector of 1's indicating which item is present at a given position in our data. We do not insert anything for the *<start>* token when building the decoder target data and it is also prefixed by a one-time step for the same reasons we mentioned in the section on decoders. Our decoder target data is the same as the decoder input data, it's just shifted by one-time step. We are now ready to set the hyperparameters of our model.

```
batch_size = 256
epochs = 100
latent_dim = 256
```

5.3.7

The next step is to create our coder. We set the *return_state* property to *True* so that the decoder will return the last hidden state and memory that will create the context vector. The states *state_h* and *state_c* represent our last hidden state and memory information. The role of the encoder is to provide a context vector that captures the context of the input sentence. However, we have no explicit target context vector defined against which we can compare the performance of the encoder. The encoder learns from the performance of the decoder, which we will describe later. The decoder error is fed back and this is how backpropagation works in the encoder and the encoder learns based on this.

```
# Step 1: Define the input layer for the encoder.
# The shape of the input is (None, num_Encoder_tokens).
# 'None' indicates that the sequence length can vary, and
'num_Encoder_tokens'
# is the number of unique tokens in the input dataset.
Encoder_inputs = Input(shape=(None, num_Encoder_tokens))

# Step 2: Define the LSTM layer for the encoder.
# 'latent_dim' is the number of units in the LSTM, which
defines the dimensionality of the hidden state.
# 'return_state=True' means we want the LSTM to return the
hidden and cell states in addition to its outputs.
Encoder = LSTM(latent_dim, return_state=True)

# Step 3: Pass the encoder inputs through the LSTM layer.
# This will return the encoder outputs and the final hidden
and cell states.
# 'Encoder_outputs' will contain the hidden states for each
time step (not used directly for Seq2Seq).
# 'state_h' and 'state_c' are the final hidden and cell
states, which we'll use as the context vector for the decoder.
Encoder_outputs, state_h, state_c = Encoder(Encoder_inputs)

# Step 4: Collect the hidden and cell states into a list
called 'Encoder_states'.
# These states will be passed to the decoder to initialize the
decoder's hidden state and cell state.
Encoder_states = [state_h, state_c]
```

5.3.8

Let's focus on the second part, the decoder. During training, both input and output data are provided to the decoder and the decoder is asked to predict the input data with an offset of 1. This helps the decoder understand what it should predict if it receives a context vector from the encoder. This learning method is referred to as teacher forcing. The initial state of the decoder is in the *Encoder_states* variable, which is our context vector obtained from the encoder. The neural network layer is part of the decoder, where the number of neurons is equal to the number of tokens (in our case, characters) present in the output character set of the decoder. This layer is associated with the output of the *softmax* function, which helps us obtain normalized probabilities for each output character. Also, this function predicts the target character with the highest probability.

The *return_sequences* parameter in the neural network decoder helps us to get the entire output sequence from the decoder. We want the output from the decoder at each time step and hence we set this parameter to *True*. Since we have used a layer along with the output of the *softmax* function, we get the probability distribution over our output features for each time step, selecting the feature with the highest probability. We judge the performance of our decoder by comparing its output produced at each time step.

```
# Step 1: Define the input layer for the decoder.
# Similar to the encoder, the shape of the input is (None,
num_Decoder_tokens),
# where 'None' indicates that the sequence length can vary,
and 'num_Decoder_tokens'
# is the number of unique tokens in the target dataset (i.e.,
the output vocabulary).
Decoder_inputs = Input(shape=(None, num_Decoder_tokens))

# Step 2: Define the LSTM layer for the decoder.
# 'latent_dim' is the number of units in the LSTM, which
defines the dimensionality of the hidden state.
# 'return_sequences=True' ensures that the LSTM returns the
hidden state for every time step (not just the final time
step).
# 'return_state=True' is used to get the hidden and cell
states from the decoder, but we don't use them here directly.
Decoder_lstm = LSTM(latent_dim, return_sequences=True,
return_state=True)

# Step 3: Pass the decoder inputs through the LSTM layer.
# 'Encoder_states' are passed as the initial hidden and cell
states for the decoder,
# ensuring that the decoder begins generating the output
sequence based on the encoder's context.
Decoder_outputs, _, _ = Decoder_lstm(Decoder_inputs,
initial_state=Encoder_states)

# Step 4: Define the Dense layer for the decoder's output.
# The Dense layer projects the LSTM outputs into the output
vocabulary space (num_Decoder_tokens).
# 'softmax' activation function ensures that the output is a
probability distribution over the target tokens.
Decoder_dense = Dense(num_Decoder_tokens,
activation='softmax')
```

```
# Step 5: Apply the Dense layer to the decoder outputs to
generate predictions for each time step.
Decoder_outputs = Decoder_dense(Decoder_outputs)
```

5.3.9

We have defined our encoder and decoder and now we will combine them into a model. We'll use the **Keras Model API** to define the different inputs and outputs that we'll use at different stages. The Model API provides *Encoder_input_data*; *Decoder_input_data* is the input to our model that will be used as the encoder and decoder inputs; *Decoder_target_data* is used as the decoder output. The model will try to convert *Encoder_input_data* and *Decoder_input_data* to *Decoder_target_data*.

```
# Step 1: Create the Seq2Seq model using the Keras Model API.
# The model takes two inputs: Encoder_inputs and
Decoder_inputs.
# Encoder_inputs are the input sequences, and Decoder_inputs
are the shifted target sequences.
# The model's output is the Decoder_outputs, which are the
predicted target sequences.
model = Model(inputs=[Encoder_inputs, Decoder_inputs],
outputs=Decoder_outputs)

# Step 2: Compile the model.
# We use the 'rmsprop' optimizer, which is effective for
training sequence models.
# The loss function used is 'categorical_crossentropy', which
is appropriate for multi-class classification tasks
# (where each token in the output sequence is a class).
# This loss function measures the difference between the
predicted probability distribution and the true distribution.
model.compile(optimizer='rmsprop',
loss='categorical_crossentropy')

# Step 3: Display the model summary.
# The model summary will print out the architecture of the
model, including the layers, number of parameters,
# and shapes of the input/output tensors for each layer.
model.summary()
```

5.3.10

The following code will serve as a reiteration of the already created functions and deployment of the encoder and decoder. So let's take a look at what the summary of our model looks like. Run the individual code blocks in order. Your task is to list what is the number of **parameters in the LSTM**.

```
import pandas as pd
import string
import re
from urllib.request import urlopen
import numpy as np
from unicodedata import normalize
import urllib
```

```
def input_file(file_name):
  data = []
  file = urllib.request.urlopen(file_name)
  for row in file:
    row = row.decode("utf-8")
    row = row.strip()
    data.append(row)
  return data

data =
input_file('https://priscilla.fitped.eu/data/nlp/slk.txt')
print(data[1500])
print(len(data))
data = data[:10000]
```

```
def create_english_slovak_sentences(data):
  EN_sentences = []
  SK_sentences = []
  for data_point in data:
    EN_sentences.append(data_point.split("\t")[0])
    SK_sentences.append(data_point.split("\t")[1])
  return EN_sentences, SK_sentences

EN_sentences, SK_sentences =
create_english_slovak_sentences(data)

def preprocess_sentences(sentence):
  re_print = re.compile('[^%s]' % re.escape(string.printable))
  table = str.maketrans('', '', string.punctuation)
  cleaned_sent = normalize('NFD', sentence).encode('ascii',
'ignore')
  cleaned_sent = cleaned_sent.decode('UTF-8')
  cleaned_sent = cleaned_sent.split()
  cleaned_sent = [word.lower() for word in cleaned_sent]
  cleaned_sent = [word.translate(table) for word in
cleaned_sent]
  cleaned_sent = [re_print.sub('', w) for w in cleaned_sent]
  cleaned_sent = [word for word in cleaned_sent if
word.isalpha()]
  return ' '.join(cleaned_sent)

def preprocess_EN_SK_sentences(EN_sentences, SK_sentences):
  SK_sentences_cleaned = []
  EN_sentences_cleaned = []
  for sent in SK_sentences:
    SK_sentences_cleaned.append(preprocess_sentences(sent))
  for sent in EN_sentences:
    EN_sentences_cleaned.append(preprocess_sentences(sent))
  return EN_sentences_cleaned, SK_sentences_cleaned

EN_sentences_cleaned, SK_sentences_cleaned =
preprocess_EN_SK_sentences(EN_sentences, SK_sentences)

def build_data(EN_sentences_cleaned, SK_sentences_cleaned):
  input_dataset = []
  target_dataset = []
  input_characters = set()
  target_characters = set()

  for SK_sentence in SK_sentences_cleaned:
    input_datapoint = SK_sentence
    input_dataset.append(input_datapoint)
    for char in input_datapoint:
      input_characters.add(char)

  for EN_sentence in EN_sentences_cleaned:
    target_datapoint = "\t" + EN_sentence + "\n"
    target_dataset.append(target_datapoint)
    for char in target_datapoint:
      target_characters.add(char)
  return input_dataset, target_dataset,
sorted(list(input_characters)),
sorted(list(target_characters))

input_dataset, target_dataset, input_characters,
target_characters = build_data(EN_sentences_cleaned,
SK_sentences_cleaned)
```

```
def get_metadata(input_dataset, target_dataset,
input_characters, target_characters):
  num_Encoder_tokens = len(input_characters)
  num_Decoder_tokens = len(target_characters)
  max_Encoder_seq_length = max([len(data_point) for data_point
in input_dataset])
  max_Decoder_seq_length = max([len(data_point) for data_point
in target_dataset])
  print('Number of data points:', len(input_dataset))
  print('Number of unique input tokens:', num_Encoder_tokens)
  print('Number of unique output tokens:', num_Decoder_tokens)
  print('Maximum sequence length for inputs:',
max_Encoder_seq_length)
  print('Maximum sequence length for outputs:',
max_Decoder_seq_length)
  return num_Encoder_tokens, num_Decoder_tokens,
max_Encoder_seq_length, max_Decoder_seq_length

num_Encoder_tokens, num_Decoder_tokens,
max_Encoder_seq_length, max_Decoder_seq_length =
get_metadata(input_dataset, target_dataset, input_characters,
target_characters)
```

```
def create_indices(input_characters, target_characters):
  input_char_to_idx = {}
  input_idx_to_char = {}
  target_char_to_idx = {}
  target_idx_to_char = {}
  for i, char in enumerate(input_characters):
    input_char_to_idx[char] = i
    input_idx_to_char[i] = char
  for i, char in enumerate(target_characters):
    target_char_to_idx[char] = i
    target_idx_to_char[i] = char
  return input_char_to_idx, input_idx_to_char,
target_char_to_idx, target_idx_to_char
```

```
input_char_to_idx, input_idx_to_char, target_char_to_idx,
target_idx_to_char = create_indices(input_characters,
target_characters)
```

```
def build_data_structures(length_input_dataset,
max_Encoder_seq_length, max_Decoder_seq_length,
num_Encoder_tokens, num_Decoder_tokens):
  Encoder_input_data = np.zeros((length_input_dataset,
max_Encoder_seq_length, num_Encoder_tokens), dtype='float32')
  Decoder_input_data = np.zeros((length_input_dataset,
max_Decoder_seq_length, num_Decoder_tokens), dtype='float32')
  Decoder_target_data = np.zeros((length_input_dataset,
max_Decoder_seq_length, num_Decoder_tokens), dtype='float32')
  print("Dimensionality of Encoder input data is : ",
Encoder_input_data.shape)
  print("Dimensionality of Decoder input data is : ",
Decoder_input_data.shape)
  print("Dimensionality of Decoder target data is : ",
Decoder_target_data.shape)
  return Encoder_input_data, Decoder_input_data,
Decoder_target_data

Encoder_input_data, Decoder_input_data, Decoder_target_data =
build_data_structures(len(input_dataset),
max_Encoder_seq_length, max_Decoder_seq_length,
num_Encoder_tokens, num_Decoder_tokens)
```

```
def add_data_to_data_structures(input_dataset, target_dataset,
Encoder_input_data, Decoder_input_data, Decoder_target_data):
  for i, (input_data_point, target_data_point) in
enumerate(zip(input_dataset, target_dataset)):
    for t, char in enumerate(input_data_point):
      Encoder_input_data[i, t, input_char_to_idx[char]] = 1.
    for t, char in enumerate(target_data_point):
      Decoder_input_data[i, t, target_char_to_idx[char]] = 1.
      if t > 0:
        Decoder_target_data[i, t - 1,
target_char_to_idx[char]] = 1.
  return Encoder_input_data, Decoder_input_data,
Decoder_target_data

Encoder_input_data, Decoder_input_data, Decoder_target_data =
add_data_to_data_structures(input_dataset, target_dataset,
Encoder_input_data, Decoder_input_data, Decoder_target_data)
```

```
import tensorflow
from tensorflow import keras

#from keras.models import Model
#from keras.layers import Input, LSTM, Dense
print('done')
```

```
batch_size = 256
epochs = 100
latent_dim = 256

Encoder_inputs = keras.layers.Input(shape=(None,
num_Encoder_tokens))
Encoder = keras.layers.LSTM(latent_dim, return_state=True)
Encoder_outputs, state_h, state_c = Encoder(Encoder_inputs)
Encoder_states = [state_h, state_c]

Decoder_inputs = keras.layers.Input(shape=(None,
num_Decoder_tokens))
Decoder_lstm = keras.layers.LSTM(latent_dim,
return_sequences=True, return_state=True)
Decoder_outputs, _, _ = Decoder_lstm(Decoder_inputs,
initial_state=Encoder_states)
Decoder_dense = keras.layers.Dense(num_Decoder_tokens,
activation='softmax')
Decoder_outputs = Decoder_dense(Decoder_outputs)
```

```
model = keras.Model(inputs=[Encoder_inputs, Decoder_inputs],
outputs=Decoder_outputs)
model.compile(optimizer='rmsprop',
loss='categorical_crossentropy')
print(model.summary())
```

5.3.11

The last step in this phase is to train the model. We will train on 80% of the data and the remaining 20% will be used to evaluate the model. We can then save the created model using the **save()** function.

```
# Step 1: Train the Seq2Seq model.
# We use the fit() method to train the model on the provided
data.
# The model will learn to translate the Slovak sentences
(Encoder_input_data) to the English sentences
(Decoder_target_data).

model.fit([Encoder_input_data, Decoder_input_data],
Decoder_target_data,
          batch_size=batch_size,   # The batch size defines
the number of samples processed before the model updates its
parameters.
          epochs=epochs,           # The number of times the
entire dataset will be passed through the model.
          validation_split=0.2)    # This splits 20% of the
data for validation during training to monitor overfitting and
evaluate performance.

# Step 2: Save the trained model to a file.
# This saves the entire model, including the architecture,
optimizer, and learned weights,
# so you can reload it later for inference or further
training.

model.save('translation_slovak_to_english.h5')
```

5.4 Machine translation - model deployment

5.4.1

Once we have created our model we need to test and deploy it. To do this we need to create a few more functions to ensure that we can send the input sequence to the encoder and retrieve the initial state of the decoder. We then send the start token and initial state to the decoder to get the next output character. Then we add the predicted output character to the sequence and repeat this process until we receive the end token or reach the maximum number of predicted characters.

```
# Step 1: Define the Encoder model for inference
# The Encoder model is used to get the hidden states from the
trained encoder.
# This is necessary for passing the context information to the
decoder during inference.

Encoder_model = Model(Encoder_inputs, Encoder_states)

# Step 2: Define the Decoder model for inference
# The Decoder model is used during inference to generate the
predicted output sequence, one token at a time.
# It takes the context vector from the Encoder model and the
previously generated token as input to predict the next token.

# Decoder's input for state (initial hidden and cell states
from the encoder)
Decoder_state_input_c = Input(shape=(latent_dim,))  # Cell
state input for the decoder LSTM
Decoder_state_input_h = Input(shape=(latent_dim,))  # Hidden
state input for the decoder LSTM
Decoder_states_inputs = [Decoder_state_input_h,
Decoder_state_input_c]

# Decoder LSTM layer receives the Decoder inputs (shifted
target sentences) and the initial states from the encoder.
Decoder_outputs, state_h, state_c =
Decoder_lstm(Decoder_inputs,
initial_state=Decoder_states_inputs)

# The decoder outputs are the predicted tokens, and we update
the states for the next prediction.
Decoder_states = [state_h, state_c]
```

```
# Decoder dense layer generates the output token predictions
for the current time step.
Decoder_outputs = Decoder_dense(Decoder_outputs)

# Step 3: Define the Decoder model
# This model takes the decoder inputs and the previous states
as inputs,
# and returns the output token predictions and updated states
for the next time step.
Decoder_model = Model([Decoder_inputs] +
Decoder_states_inputs, [Decoder_outputs] + Decoder_states)
```

5.4.2

In the next step let's create a **decode_sequence()** function that will use the encoder-decoder model we created.

```
def decode_sequence(input_seq):
    # Step 1: Predict the encoder's output states for the
input sequence
    # The Encoder model processes the input sequence and
returns the encoder's final hidden states (context vector).
    states_value = Encoder_model.predict(input_seq)

    # Step 2: Initialize the target sequence with the  token
(represented as 1 in the one-hot encoding)
    # The target sequence for the decoder is initialized with
the  token to begin generating the translated sentence.
    target_seq = np.zeros((1, 1, num_Decoder_tokens))
    target_seq[0, 0, target_char_to_idx['\t']] = 1.  # '\t' is
the  token in the target sentence

    # Step 3: Initialize the stop condition flag and the
decoded sentence
    stop_condition = False
    decoded_sentence = ''

    # Step 4: Start generating the translated sentence one
token at a time
    while not stop_condition:
        # Step 4a: Predict the next token (word) and its
updated states using the decoder
```

```
        # The Decoder model takes the current target sequence
and the encoder's states to predict the next token.
        output_tokens, h, c =
Decoder_model.predict([target_seq] + states_value)

        # Step 4b: Find the token with the highest probability
(argmax)
        # The output tokens contain probabilities for each
possible next token. We take the one with the highest
probability.
        sampled_token_index = np.argmax(output_tokens[0, -1,
:])

        # Step 4c: Get the character corresponding to the
predicted token index
        # We look up the predicted token index to get the
corresponding character.
        sampled_char = target_idx_to_char[sampled_token_index]

        # Step 4d: Add the predicted character to the decoded
sentence
        decoded_sentence += sampled_char

        # Step 4e: Check if we have reached the  token or
exceeded the max length
        # We stop generating tokens if we encounter the  token
or if the sentence length exceeds the limit.
        if (sampled_char == '\n' or len(decoded_sentence) >
max_Decoder_seq_length):
            stop_condition = True

        # Step 4f: Update the target sequence for the next
prediction step
        # The next input for the decoder is the previously
predicted token.
        target_seq = np.zeros((1, 1, num_Decoder_tokens))
        target_seq[0, 0, sampled_token_index] = 1.

        # Step 4g: Update the decoder states for the next time
step
        states_value = [h, c]

    # Step 5: Return the fully decoded sentence
    return decoded_sentence
```

5.4.3

Finally, we can create a **decode()** function whose parameter is the index of a sentence from the data file.

```
def decode(seq_index):
  input_seq = Encoder_input_data[seq_index: seq_index + 1]
  decoded_sentence = decode_sequence(input_seq)
  print('-')
  print('Input sentence:', input_dataset[seq_index])
  print('Decoded sentence:', decoded_sentence)
```

Machine Translation Evaluation

Chapter 6

6.1 Basics of evaluation

6.1.1

Language is the bridge that connects people across cultures, and translation is the key process that enables this connection. A good translation is not merely a word-for-word replacement of terms but an effort to express ideas, emotions, and nuances as closely as possible in the target language. The goal is to convey the meaning of the source text in such a way that it resonates with the target audience, without sounding artificial or forced. Translation is about capturing the essence of the source content while making it accessible and understandable to those who speak the target language.

To understand translation better, we need to familiarize ourselves with some key concepts:

- **source text** is the original text that needs to be translated
- **target text** is the translated version in the desired language
- **hypothesis** refers to the machine-generated translation of the source text
- **reference** is a human-generated translation, typically used for comparison
- **machine translation (MT)** involves using computer software to automatically translate text from one language to another

While machine translation has made significant advances, it is still a challenge to achieve translations that perfectly reflect the subtleties of human language. Machine translation is commonly used in various applications, such as translating websites, documents, and even conversations in real-time. However, there are still limitations. Although MT can quickly process large amounts of text, it often struggles with idiomatic expressions, cultural context, and other nuances that human translators naturally understand. Human translators, on the other hand, bring contextual knowledge, cultural understanding, and emotional intelligence to their work, making their translations more accurate and meaningful.

Despite these challenges, MT has revolutionized how we communicate across languages, making it easier to access information and connect with people globally. However, to achieve high-quality translations, we must understand the strengths and limitations of both machine-generated translations and human translations.

6.1.2

Which of the following best describes a "target text"?

- The text translated into the desired language
- The original text that needs to be translated
- A machine-generated translation
- A human-generated translation

6.1.3

Why do machine translations often struggle with idiomatic expressions and cultural context?

- They lack emotional intelligence and cultural knowledge
- They can process large amounts of text quickly
- They are created by human translators
- They are always more accurate than human translations

6.1.4

When evaluating a translation, whether human or machine-generated, it's important to consider the purpose of the evaluation. For instance, in some cases, the focus might be on the comprehension of the text, ensuring that the translated message makes sense to the reader. This could be done without delving into any errors or mistakes. On the other hand, an in-depth error analysis can be valuable to identify both weaknesses and strengths in machine translation systems or human translators. Such detailed analysis helps pinpoint specific areas that need improvement, such as translating idiomatic expressions, technical terms, or cultural context.

The quality of a translation is influenced by several factors. A translator's subjective view and their experience with the language pair can significantly affect the final output. Additionally, the context in which the source text is written is also a key determinant. For example, a text in the scientific field might require specialized knowledge and a more technical vocabulary than a general conversation. Translators, or machine translation systems, must be able to handle such complexities, ensuring that nuances are preserved while accurately conveying the meaning.

In machine translation, the relationship between the source text and the target text is especially crucial. The translation process involves converting one language's textual form into another while maintaining meaning, tone, and style. Any loss of these elements can reduce the quality of the translation. The closer the machine-generated target text mirrors these factors in the source, the better the translation will be, although machines still struggle with capturing context and tone compared to human translators.

6.1.5

What is the main goal of an in-depth error analysis in translation?

- To identify the weaknesses and strengths of the translation
- To make the target text sound better
- To improve the source text
- To reduce the length of the translation

6.1.6

Which of the following factors can affect the quality of a translation?

- The experience of the translator
- The context of the source text
- The length of the source text
- The format of the text

6.1.7

Evaluating the quality of a translation is a complex task, as the concept of "quality" varies widely depending on the context. There are different ideas about what makes a translation "neat" or "acceptable," and these opinions can differ significantly between cultures, languages, and even individual preferences. The diversity of translation methods and approaches adds another layer of ambiguity when trying to define clear and universally accepted criteria for evaluation. This complexity makes it difficult to develop a one-size-fits-all approach to translation assessment.

To address these challenges, translation models and evaluation methods often rely on human assessors. These assessors are tasked with identifying errors in the translation and determining whether it aligns with specific linguistic or functional standards. This process usually involves comparing the translated text with the original to evaluate its clarity and fidelity. Fidelity refers to how accurately the meaning of the original text has been conveyed, while clarity focuses on the overall readability and natural flow of the target text. Both attributes are essential not only for human translations but also for evaluating machine translations, which, despite advancements, still face challenges in producing text that mirrors these qualities.

In the case of machine translation, it is crucial to ensure that the output not only maintains the meaning of the original text but also conveys it clearly and naturally in the target language. Machine translation systems, while increasingly sophisticated, often struggle with capturing nuances, idiomatic expressions, and cultural contexts that human translators can more easily navigate. Therefore, the evaluation of machine translation is particularly focused on how well these systems replicate the fidelity and clarity that would be expected from human translations.

6.1.8

Why is evaluating translation quality difficult?

- There are different conceptions of quality and translation approaches
- There is a lack of human assessors
- Machines cannot evaluate translations
- All translations are identical

6.1.9

What is the main focus when evaluating machine translations?

- Fidelity and clarity of the translation
- The length of the translation
- The time it took to complete the translation
- The choice of vocabulary used

6.1.10

The quality of machine translation can be assessed using two primary methods: manual evaluation and automatic evaluation. Manual evaluation involves human assessors, typically professional translators, who judge the translation based on its accuracy and precision. Accuracy refers to how well the translated text reflects the meaning of the source text, while precision evaluates the exactness of the language used. Manual evaluation typically uses a scale to rate these factors, often from 1 to 5 points, allowing the evaluator to assess how well the translation preserves the context and intent of the original text.

While manual evaluation provides in-depth and human-centered feedback, it has its limitations. The process is time-consuming and can be influenced by the subjectivity of the evaluator, as different people may interpret the translation's quality differently. Moreover, relying on experts to assess translations can be resource-intensive, making it less feasible for large-scale evaluations. These challenges have led to the rise of automatic evaluation metrics, which can quickly and consistently assess the quality of a translation. Automatic methods do not require human assessors and are often based on algorithms that compare the machine translation output to a reference translation, measuring various factors like fluency and adequacy.

Automatic evaluation is increasingly used in the industry due to its efficiency and scalability. Common automatic evaluation metrics include BLEU (Bilingual Evaluation Understudy), which measures the overlap between n-grams in the machine translation output and reference translations, and METEOR, which considers synonymy and word order in its evaluation. Although automatic evaluation has made substantial progress, it is still not perfect and often lacks the nuanced understanding that human evaluators provide. However, it remains a valuable tool for quickly assessing large amounts of data and providing a baseline for further refinement of machine translation systems.

6.1.11

What is the main limitation of manual evaluation of machine translations?

- It is time-consuming and subject to evaluator subjectivity
- It is more accurate than automatic evaluation
- It cannot evaluate precision
- It does not use a scale for ratings

6.1.12

What is the advantage of automatic evaluation over manual evaluation?

- It is faster and does not rely on experts
- It provides more nuanced feedback
- It requires fewer human resources and is more scalable
- It is always more accurate

6.1.13

Machine translation automatic evaluation metrics are highly valued for their objectivity and efficiency. Unlike manual evaluation, they eliminate the subjectivity of human assessors and can process large datasets quickly. However, these metrics are not without limitations. One of the key challenges is their reliance on reference translations, which are used to compare the machine-generated translation and determine its quality. The accuracy of the evaluation is directly influenced by the number of reference translations available. More reference translations generally lead to a more accurate evaluation, as they provide a broader basis for comparison. However, in practice, it is common to only have one reference translation, which may limit the reliability of the evaluation.

Despite this limitation, the use of automatic evaluation metrics has grown significantly due to their speed and scalability. These metrics are based on various linguistic and statistical approaches, each with its own strengths and weaknesses. For example, some metrics focus on the overlap of n-grams (sequences of n words) between the machine translation output and the reference translation, while others consider factors such as word order or synonymy. These metrics are particularly useful for quickly assessing the quality of translations in large-scale machine translation systems, especially when human evaluation is not feasible. As machine translation continues to evolve, more advanced evaluation metrics are being developed to address the challenges of reference scarcity and to provide more accurate assessments.

Despite their widespread use, automatic evaluation metrics are not perfect. They may struggle to capture nuances in language such as idiomatic expressions, cultural context, or subtle differences in meaning that a human evaluator would easily detect. Additionally, different metrics may yield different results depending on their underlying methodology. As a result, while automatic metrics are valuable tools for initial evaluations and large-scale assessments, human evaluation remains essential for a deeper understanding of translation quality.

6.1.14

What is the main disadvantage of using automatic evaluation metrics in machine translation?

- They rely on reference translations, which are not always available
- They are subjective
- They are slower than manual evaluation
- They cannot evaluate accuracy

6.2 Automatic evaluation metrics

6.2.1

Automatic evaluation of machine translation is crucial for assessing the performance of machine translation systems. The evaluation process often relies on several key metrics such as:

- precision,
- recall,
- F-measure,

They are fundamental to understanding how well a machine translation system is performing. These metrics are particularly important for evaluating the accuracy and relevance of the translation produced by the system in comparison to a reference translation.

Precision measures the proportion of relevant words or phrases in the output that are also present in the reference, while **recall** assesses the proportion of relevant words or phrases in the reference that appear in the machine-generated translation. **F-measure** is a combined metric that balances precision and recall into a single value, providing a more comprehensive measure of translation quality.

In the context of machine translation, these metrics are applied to the results of binary classification, where each word or phrase in the translation is classified as either a match (1) or a mismatch (0) compared to the reference translation. The focus is on determining how many of the words or phrases in the machine-generated translation are accurate and how well the overall translation reflects the original meaning. This binary approach allows for a straightforward comparison between machine-generated translations and human-generated reference translations.

While precision, recall, and F-measure provide important insights into the accuracy of a translation, they are not the only metrics used in machine translation evaluation. Many other automatic evaluation metrics have been developed, building upon these basic metrics, to address various aspects of translation quality. However, the reliance on reference translations remains a central feature of automatic evaluation, and the availability of multiple reference translations generally leads to more accurate assessments.

6.2.2

Which of the following are basic metrics used in automatic evaluation of machine translation?

- Precision
- Recall
- F-measure
- Perplexity

6.2.3

Precision

Precision is a crucial metric in evaluating machine translation systems as it determines the accuracy of the positive predictions made by the system. In machine translation, the positive predictions refer to the words or phrases that the system translates correctly. Precision is calculated by comparing the number of true positives (TP), which are the correctly translated words, to the sum of true positives and false positives (FP), which are the words incorrectly identified as correct. Mathematically, precision is defined as:

$$\text{Precision} = \frac{TP}{(TP + FP)}$$

Where:

- **TP (True Positives)** are the correct translations, i.e., the words that the machine translation system has translated correctly.
- **FP (False Positives)** are the incorrect translations, i.e., the words that the system has wrongly translated or falsely predicted as a correct translation.

In machine translation, the goal is to maximize precision, meaning the system should produce as many correct translations as possible without introducing false translations. However, precision alone is not sufficient to fully evaluate a translation system, which is why it is often combined with recall and the F-measure to provide a more comprehensive assessment of translation quality.

```
def my_precision(ref, hyp):
  correct = 0
  length0 = len(hyp)
  my_ref = ref.copy()
  for i in range(0,len(hyp)):
    for j in range(0,len(my_ref)):
      if hyp[i]==my_ref[j]:
        correct += 1
        my_ref.remove(my_ref[j])
        break
  return float(correct/length0)
```

```
# try to compare two translations
# Example usage
reference = ['the', 'cat', 'sat', 'on', 'the', 'mat']
hypothesis = ['the', 'cat', 'is', 'on', 'the', 'mat']

# Calling the function to calculate precision
precision_value = my_precision(reference, hypothesis)
print(f"Precision: {precision_value}")
```

Program output:

```
Precision: 0.8333333333333334
```

```
# Example usage with more complicated sentences
reference = ['the', 'quick', 'brown', 'fox', 'jumps', 'over', 
'the', 'lazy', 'dog', 'in', 'the', 'morning']
hypothesis = ['the', 'fast', 'brown', 'fox', 'leaps', 'over', 
'the', 'lazy', 'dog', 'at', 'dawn']

# Calling the function to calculate precision
precision_value = my_precision(reference, hypothesis)
print(f"Precision: {precision_value}")
```

Program output:

```
Precision: 0.6363636363636364
```

6.2.4

Which of the following does precision measure in machine translation?

- Correct translations versus incorrect translations
- Correct translations versus false predictions
- Correct translations versus missing translations
- False predictions versus missing translations

6.2.5

Recall

Recall is another important metric used to evaluate the performance of machine translation systems. While precision focuses on the accuracy of the positive predictions, recall is concerned with the system's ability to correctly identify all relevant cases in the dataset. In machine translation, recall measures how many of the true translations (true positives) the system was able to capture from the total number of relevant translations. It is calculated by comparing the number of true positives (TP) to the sum of true positives and false negatives (FN), which are the cases where the system failed to identify the correct translation.

Mathematically, recall is defined as:We calculate the recall metric as:

$$recall = \frac{TP}{TP + FN}$$

Where:

- **TP (True Positives)** are the translations that the system correctly identified.
- **FN (False Negatives)** are the translations that the system failed to identify as correct, or missed.

Recall is crucial when the goal is to minimize missing important translations, even at the cost of introducing some false positives. Recall and precision are often inversely related; improving one typically leads to a decrease in the other. Thus, striking a balance between these two metrics is important for optimizing machine translation performance. This balance is often assessed using the F-measure, which combines both precision and recall into a single metric.

```
def my_recall(ref, hyp):
  correct = 0
  lengthR = len(ref)
  my_ref = ref.copy()
  for i in range(0,len(hyp)):
    for j in range(0,len(my_ref)):
      if hyp[i]==my_ref[j]:
        correct += 1
        my_ref.remove(my_ref[j])
        break
  return float(correct/lengthR)
```

```
# Example usage
reference = ['the', 'quick', 'brown', 'fox', 'jumps', 'over', 
'the', 'lazy', 'dog', 'in', 'the', 'morning']
hypothesis = ['the', 'fast', 'brown', 'fox', 'leaps', 'over', 
'the', 'lazy', 'dog', 'at', 'dawn']

# Calling the function to calculate
recall_value = my_recall(reference, hypothesis)
print(f"Recall: {recall_value}")
```

Program output:

```
Recall: 0.5833333333333334
```

6.2.6

Which of the following is measured by recall in machine translation?

- The proportion of true translations correctly identified
- The proportion of missed correct translations
- The proportion of incorrect translations
- The proportion of false positives

6.2.7

F-measure

The F-measure (also known as the **F-score**) is a metric that combines both precision and recall into a single number, offering a balanced assessment of a model's performance. It is especially useful when there is an uneven class distribution, or when both precision and recall are important to evaluate together.

The formula for the F-measure is calculated as the harmonic mean of precision and recall:

$$F_\beta = \frac{(1+\beta^2) \times \text{precision} \times \text{recall}}{(\beta^2 \times \text{precision}) + \text{recall}}$$

Where:

- **Precision** - the proportion of true positive predictions out of all positive predictions made.
- **Recall** - the proportion of true positive predictions out of all actual positive cases.
- **β (beta)** - a weighting factor that determines the relative importance of precision and recall. When β=1, the F-measure is the harmonic mean of precision and recall, balancing them equally.

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

Unlike the simple arithmetic mean, the harmonic mean used in the F-measure penalizes extreme values (outliers). This means that if either precision or recall is very low, the F-measure will also be low, even if the other metric is high. Therefore, the F-measure ensures that both precision and recall contribute equally to the evaluation, and it is particularly useful when it is necessary to balance these two metrics, especially in situations where both false positives and false negatives are important to minimize.

For instance, if a machine translation system has a **precision** of 0.9 (90%) and a **recall** of 0.7 (70%), the F-measure would be:

```
F1 = 2 * (0.9 * 0.7) / (0.9 + 0.7) = 1.26 / 1.6 ≈ 0.7875
```

This means that the machine translation system, taking both precision and recall into account, has an F-measure of approximately 0.79.

```
def my_f_measure(ref, hyp):
  prec = float(my_precision(ref, hyp))
  rec = float(my_recall(ref, hyp))
  try:
    res = (prec*rec)/((prec+rec)/2)
  except:
    res = NaN
  return res
# -------------------------------------
def my_recall(ref, hyp):
  correct = 0
  lengthR = len(ref)
  my_ref = ref.copy()
  for i in range(0,len(hyp)):
    for j in range(0,len(my_ref)):
      if hyp[i]==my_ref[j]:
        correct += 1
        my_ref.remove(my_ref[j])
        break
  return float(correct/lengthR)

def my_precision(ref, hyp):
  correct = 0
  lengthO = len(hyp)
  my_ref = ref.copy()
  for i in range(0,len(hyp)):
    for j in range(0,len(my_ref)):
      if hyp[i]==my_ref[j]:
        correct += 1
        my_ref.remove(my_ref[j])
        break
  return float(correct/lengthO)
```

```
# Example usage
reference = ['the', 'quick', 'brown', 'fox', 'jumps', 'over',
'the', 'lazy', 'dog', 'in', 'the', 'morning']
hypothesis = ['the', 'fast', 'brown', 'fox', 'leaps', 'over',
'the', 'lazy', 'dog', 'at', 'dawn']

# Calling the function to calculate
f_score_value = my_f_measure(reference, hypothesis)
print(f"F-score: {f_score_value}")
```

Program output:

```
F-score: 0.6086956521739131
```

6.2.8

Which of the following statements about the F-measure is true?

- The harmonic mean used in the F-measure penalizes outliers more than the arithmetic mean.
- The F-measure is calculated by averaging precision and recall without any weighting.
- The F-measure gives more weight to precision than recall.
- The F-measure only considers recall and ignores precision.

6.2.9

BLEU

The most well-known metric for automatic machine translation evaluation is the **BLEU** (*Bilingual Evaluation Understudy*) metric, which measures *precision*, i.e., how many words (and/or n-grams) in the machine-generated translations appeared in the human reference translations. In other words, it claims that the closer a machine translation is to a professional human translation the better it is.

The issue can be that every translator has a different vocabulary and a different way of composing a sentence, so it is almost impossible to get identical translations. One way of comparing translations is at the level of so-called **n-grams**. This is a sequence of *1, 2, ..., n* words, i.e. unigrams, bigrams,..., n-grams. The correspondence of these n-grams in translations is characterized by the so-called **n-gram precision**, which can be calculated for individual n-grams or even for the whole text.

Ideally, the length of the candidate translation would be equal to the length of the reference translation. Otherwise:

- if a candidate translation is created that is too long, we penalize it using the **modified n-gram precision**,
- if too short a translation is produced, a **brevity penalty** (BP) is applied.

We can use the most popular natural language processing library, **nltk**, to compute the BLEU metric score. However, before we can compute the BLEU score we need to tokenize the reference text and the hypothesis. This means breaking the sentence into tokens, i.e. word units. We will also use a function from the nltk library to do this, namely **word_tokenize()**. We can then call the **sentence_bleu()** function, which will return the score for the BLEU metric. By setting the weights parameters, we can tell the function for which n-gram we want to calculate the

BLEU. The closer the BLEU score is to 1, the better, and more correct the translation. Conversely, a value closer to 0 indicates a poor translation.

```
from nltk.translate.bleu_score import sentence_bleu
from nltk import word_tokenize

ref = "Computer science spans theoretical disciplines (such as
algorithms, theory of computation, information theory, and
automation) to practical disciplines (including the design and
implementation of hardware and software)."
hyp = "Computer science includes theoretical disciplines (such
as algorithms, theory of computing, theoretical computer
science and automation) and practical disciplines (including
hardware and software design and implementation)."

ref = word_tokenize(ref)
hyp = word_tokenize(hyp)

print('BLEU-1:',sentence_bleu([ref], hyp, weights=(1,0,0,0)))
print('BLEU-2:',sentence_bleu([ref], hyp,
weights=(0.5,0.5,0,0)))
print('BLEU-3:',sentence_bleu([ref], hyp,
weights=(0.33,0.33,0.33,0)))
print('BLEU-4:',sentence_bleu([ref], hyp,
weights=(0.25,0.25,0.25,0.25)))
```

Program output:

```
BLEU-1: 0.7700677691930449
BLEU-2: 0.656003030651081
BLEU-3: 0.5422370790508815
BLEU-4: 0.43076614970957955
```

PRISCILLA

priscilla.fitped.eu