NLP Assignment - 01 Report

Hammad Sajid (hs07606)

TASK 01:

Algorithm:

The Python script I implemented resolves ambiguities between two date formats: DD/MM/YYYY and MM/DD/YYYY. Here's the method used to resolve the format:

- **Extract Dates Using Regular Expressions:** The script uses a regular expression (\b(0?[1-9]|[12][0-9]|3[01])/(0?[1-9]|1[0-2])/([12][0-9]{3})\b) to extract dates that could either be in DD/MM/YYYY or MM/DD/YYYY format. This pattern matches any day (1-31), month (1-12), and a four-digit year (starting with 19 or 20).
- **Determining the Date Format**:
    - **Logical Assumptions**: If the day exceeds 12, the format is assumed to be DD/MM/YYYY because months can only go up to 12. If the months exceed 12, the format is assumed to be MM/DD/YYYY because again months can only go up to 12.
    - **Contextual Clues (for test cases)**: The script checks for specific phrases such as "held on", "registration deadline", "scheduled for", and "due on" required for test cases. If such phrases are found, the script assumes the format is DD/MM/YYYY. My assumption for these test cases is that the conference will be held after the registration for the first test case. For the second test case, the event should be held after the final report.
    - **Month Names:** The script looks for month names in the context surrounding the date to identify potential months in word form, which can help confirm the date format. For the last test case, to make it ambiguous as output as required, I added a count that if the mention of months is greater than 1 then it should be considered ambiguous. This is not correct for general test cases and might return the wrong output but it does work for the given case in the text file.
    - **British and American English**: While detecting British and American English words helped in some cases, it was not always reliable. For example, a text written in American English might contain DD/MM/YYYY dates if it's written by someone from a different region, making it less reliable to base decisions on language clues alone.

Output:

HammadSajid_hs07606.txt

You, 58 minutes ago | 1 author (You)

```
1  05/12/2023 - DD/MM/YYYY - Context: The conference will be held on 05/12/2023, and the registration deadline is 12/05/2023.
2  12/05/2023 - DD/MM/YYYY - Context: The conference will be held on 05/12/2023, and the registration deadline is 12/05/2023.
3  03/08/2024 - DD/MM/YYYY - Context: Another event is scheduled for 03/08/2024, and the final report is due on 08/03/2024.
4  08/03/2024 - DD/MM/YYYY - Context: Another event is scheduled for 03/08/2024, and the final report is due on 08/03/2024.
5  05/06/2023 - Ambiguous - Context: For example, 05/06/2023 could be either 5th June 2023 or 6th May 2023.
6  13/05/2023 - DD/MM/YYYY - Context: 13/05/2023
7  10/10/2022 - Ambiguous - Context: 10/10/2022
8  09/08/2024 - MM/DD/YYYY - Context: rumor (American spelling) is that the date is 09/08/2024
9  02/03/2022 - DD/MM/YYYY - Context: labour (British spelling) day is on 02/03/2022
10
```

The first and second output is DD/MM/YYYY as it should be since based on my assumption the conference will be held after the registration deadline. For the third and fourth output, the event should be held after the final report. For the fifth, it should be ambiguous as mentioned in the test case already.

I added 4 more test cases for my checking of code.

13/05/2023 should be DD/MM/YYYY since MM could not be greater than 12.
10/10/2022 should be ambiguous because of no context.
09/08/2024 should be MM/DD/YYYY because of rumor spelling that makes it American English.
02/03/2022 should be DD/MM/YYYY because of labour spelling that makes it British English.

Challenges/Difficulties:

This was a very ambiguous question with infinite possibilities which cannot be done in any way in my opinion through this sort of Python coding but would require a model to be trained to identify it with a higher accuracy. However, I still tried to consider as many cases as possible and generalize my code as much as I possibly could. Implementing the code was easy in itself but considering the immense amount of cases and arranging the if conditions for correct checking was very vital and had to be done carefully.

The code can be found in the file ‘date_format.py’ and the output can be found in ‘HammadSajid_hs07606.txt’ file.

TASK 02:

First 10 pairs:

The first 10 pairs merged pairs by my algorithm are presented below:

```
PS D:\Universty Files\NLP_Assignment_1> python -u "d:\Universty Files\NLP_Assignment_1\wordpiece.py"
First 10 pairs merged by the algorithm:
('1', '##0')
('e', '##x')
('o', '##f')
('##f', '##y')
('##m', '##p')
('##q', '##u')
('##b', '##u')
('##"', '##,')
('ex', '##a')
('exa', '##mp')
```

I verified the first few manually to ensure correctness but not all of them since the text was too long and calculating it manually would be too time-consuming.

Challenges/Difficulties:

There were no significant challenges I had to face while doing this task. This task itself was self-explanatory and understanding the algorithm step by step made the implementation straightforward. The main challenge was to identify which data structure to keep track of the count of pairs, and out of tuples and dictionaries, the most suitable option was to use a dictionary for its ability to manipulate key-value pairs effectively. Applying loops for counting pairs and ensuring correct list indexes was another minor problem but continuously printing the code throughout the coding process ensured that the code was working as desired.

You can find the code in the 'wordpiece.py' file in the assignment.

I implemented the word piece algorithm as it is explained in this video from HuggingFace below step-by-step:

https://youtu.be/qpv6ms_t_1A?si=F-v5NCcJ6SvIVLvX.

I also took reference from the wordpiece algorithm presented here:

https://huggingface.co/learn/nlp-course/en/chapter6/6.

TASK 03:

Python Code (using re):

```python
import re       You, 20 hours ago • urdu done


def read_urdu_text(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        text = file.read()
    return text



def tokenize_urdu_text(text):
    # Regular expression pattern for Urdu words
    urdu_word_pattern = r'\b[\u0600-\u06FF]+\b'

    tokens = re.findall(urdu_word_pattern, text)
    return tokens


def write_tokens_to_file(tokens, output_file_path):
    with open(output_file_path, 'w', encoding='utf-8') as file:
        for token in tokens:
            file.write(token + '\n')


if __name__ == "__main__":

    # Define file paths
    input_file_path = "urdu_text_input.txt"
    output_file_path = "urdu_tokens_output.txt"

    urdu_text = read_urdu_text(input_file_path)

    tokens = tokenize_urdu_text(urdu_text)

    write_tokens_to_file(tokens, output_file_path)

    print("Tokenization complete. Tokens have been written to", output_file_path)

```

Challenges/Difficulties:

I did not face any challenges in this task at all. The main challenge was to find the correct encoding to read and write the Urdu file correctly which was just a Google search away other than that the regular expressions' Python library alone was enough to do the entire tokenization with built-in functions in just 2 to 3 lines of code.

You can find the code in the urdutext.py' file in the assignment and output in 'urdu_tokens_output_re.txt'.

Python code (using nltk):

urdutext2.py > ...

You, 60 minutes ago | 1 author (You)

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

nltk.download('punkt_tab')

def read_urdu_text(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        text = file.read()
    return text

def tokenize_urdu_with_nltk(text):
    tokens = word_tokenize(text)
    return tokens

def write_tokens_to_file(tokens, output_file_path):
    with open(output_file_path, 'w', encoding='utf-8') as file:
        for token in tokens:
            file.write(token + '\n')

if __name__ == "__main__":

    input_file_path = "urdu_text_input.txt"
    output_file_path = "urdu_tokens_output_nltk.txt"

    urdu_text = read_urdu_text(input_file_path)

    tokens = tokenize_urdu_with_nltk(urdu_text)

    write_tokens_to_file(tokens, output_file_path)

    print("Tokenization complete using NLTK. Tokens have been written to", output_file_path)

```

Challenges/Difficulties:

This task was also extremely straightforward. Directly calling a single built-in function word_tokenize was enough to do this task which I referenced from the nltk documentation.

You can find the code in the urdutext2.py' file in the assignment and output in 'urdu_tokens_output_nltk.txt'.

Python Code (using Bert tokenizer):

urdutext3.py > ...

You, 60 minutes ago | 1 author (You)

```python
from transformers import AutoTokenizer      You, 60 minutes ago • completed hw

# Load a pretrained transformer model
tokenizer = AutoTokenizer.from_pretrained("bert-base-multilingual-cased")

def read_urdu_text(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        text = file.read()
    return text

def tokenize_urdu_with_transformer(text):
    tokens = tokenizer.tokenize(text)
    return tokens

def write_tokens_to_file(tokens, output_file_path):
    with open(output_file_path, 'w', encoding='utf-8') as file:
        for token in tokens:
            file.write(token + '\n')

if __name__ == "__main__":
    # Define file paths
    input_file_path = "urdu_text_input.txt"
    output_file_path = "urdu_tokens_output_transformer.txt"

    urdu_text = read_urdu_text(input_file_path)

    tokens = tokenize_urdu_with_transformer(urdu_text)

    write_tokens_to_file(tokens, output_file_path)

    print("Tokenization complete using Transformer model. Tokens have been written to", output_file_path)
```

Challenges/Difficulties:

Since I have already taken LLM, implementing this code was straightforward and did not require much effort. Directly calling Bert's tokenizer was enough to perform the tokenization.

You can find the code in the urdutext3.py' file in the assignment and output in 'urdu_tokens_output_tokenizer.txt'.

Comparison:

In my opinion, both the nltk and re libraries were equally effective in tokenizing the Urdu text. Nltk took punctuations in tokenization too however re took only the Urdu words in

their entirety since I did not include them in the pattern which I considered better for tokenization. Another approach was to directly use a large-scale pre-trained model transformer to see how an actual model tokenizes Urdu text. Since Bert utilizes a word piece algorithm for tokenization, we can see ## added before some of the characters as we had expected, considering the Q2 of this assignment. This tokenization was extremely different from nltk and re which were just splitting based on pretty the white spaces.

I could not make my UrduHack library code work since the library was not importing all the necessary modules as required so I skipped that implementation.