NLP Assignment 01 Report

Hammad Sajid (hs07606)

13/09/2024

Contents

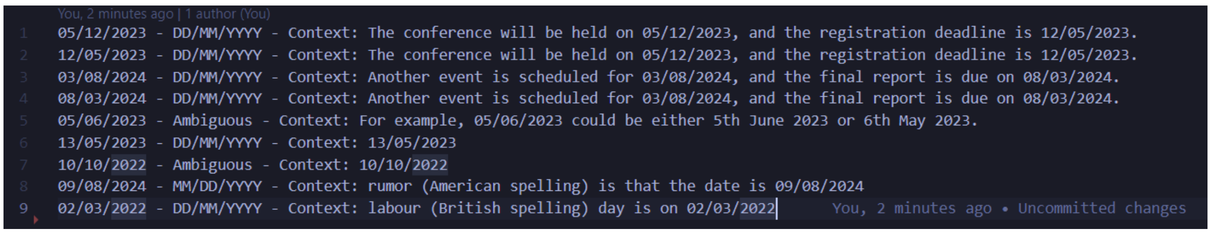
1 Task 01: Resolving Ambiguities Between Date Formats

1.1 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)—̲ 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`.
  - **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` since
  - **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.
  - **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.

1.2 Output



```
You, 2 minutes ago | 1 author (You)
05/12/2023 - DD/MM/YYYY - Context: The conference will be held on 05/12/2023, and the registration deadline is 12/05/2023.
12/05/2023 - DD/MM/YYYY - Context: The conference will be held on 05/12/2023, and the registration deadline is 12/05/2023.
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.
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.
05/06/2023 - Ambiguous - Context: For example, 05/06/2023 could be either 5th June 2023 or 6th May 2023.
13/05/2023 - DD/MM/YYYY - Context: 13/05/2023
10/10/2022 - Ambiguous - Context: 10/10/2022
09/08/2024 - MM/DD/YYYY - Context: rumor (American spelling) is that the date is 09/08/2024
02/03/2022 - DD/MM/YYYY - Context: labour (British spelling) day is on 02/03/2022      You, 2 minutes ago • Uncommitted changes
```

Figure 1: Output of task 01.

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.

1.3 Challenges and Difficulties

This was a very ambiguous question with infinite possibilities which, in my opinion, cannot be done completely using Python coding alone but would require a model to be trained to identify the format with higher accuracy. However, I tried to consider as many cases as possible and generalize my code. Implementing the code was easy, but handling the immense variety of cases and arranging the conditions carefully was crucial.

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

2 Task 02: First 10 Merged Pairs in WordPiece Algorithm

2.1 First 10 Pairs

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

```
First 10 pairs merged by the algorithm with their frequency scores:
Pair: ('1', '##0'), Score: 1.000000
Pair: ('o', '##f'), Score: 0.250000
Pair: ('##f', '##y'), Score: 0.500000
Pair: ('e', '##x'), Score: 0.250000
Pair: ('##m', '##p'), Score: 0.166667
Pair: ('##q', '##u'), Score: 0.142857
Pair: ('##b', '##u'), Score: 0.166667
Pair: ('##mp', '##l'), Score: 0.083333
Pair: ('##bu', '##l'), Score: 0.090909
Pair: ('ex', '##a'), Score: 0.071429
```

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.

2.2 Challenges and Difficulties

There were no significant challenges in this task. The main challenge was identifying which data structure to use for tracking pairs' counts. A dictionary was most suitable for its ability to manipulate key-value pairs. Ensuring correct list indexing was another minor issue, but printing debug information during development helped ensure the code worked as intended. Other than that, I faced some problems in assigning ## before characters during tokenization. Initially, I did not add any ## in my code which was giving a very different output, but when I ran the HuggingFace's implementation seperately on Google Collab and ran the test script on their code, I realized that I had to add ## for correct outputs. Luckily as soon as I added ## prefix, my code started working as desired.

I implemented the WordPiece algorithm as explained in this video:
HuggingFace Video on WordPiece Algorithm.

I also referenced the WordPiece algorithm presented here:
HuggingFace Course - WordPiece Algorithm.

3 Task 03: Urdu Text Tokenization

3.1 Using Regular Expressions

```python
import re


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_re.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)
```

3.2 Challenges/Difficulties of Regex approach

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 **urdu_text_re.py** file in the assignment and output in **urdu_tokens_output_re.txt**

3.3 Using NLTK

```
    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)
```

3.4 Challenges/Difficulties of NLTK approach

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 **urdu_text_nltk.py** file in the assignment and output in **urdu_tokens_output_nltk.txt**.

3.5 Using BERT Tokenizer

```python
from transformers import AutoTokenizer

# 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)
```

3.6 Challenges/Difficulties of BERT approach

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 **urdu_text_transformer.py** file in the assignment and output in **urdu_tokens_output_tokenizer.txt**.

3.7 Comparison of Approaches

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 much 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.