Kushal Chandani (kc07535)

Natural Language Processing

Assignment 1 Report

TABLE OF CONTENTS

Question 1: Resolving Ambiguities Between DD/MM/YYYY and MM/DD/YYYY Date Formats [40 points].

1. Proposed Algorithm for Resolving the Date Format Ambiguity

My objective was to resolve ambiguities between DD/MM/YYYY and MM/DD/YYYY date formats by using contextual clues and logical assumptions.

2. Steps in the Algorithm

2.1. Reading the Text File: We read the input text from the file date_format_dd_mm_yyyy.txt to extract dates formatted as DD/MM/YYYY or MM/DD/YYYY.

2.2. Extracting the Dates Using Regular Expressions: Dates in the text are identified using the regex pattern *\b\d{2}/\d{2}/\d{4}\b*, which matches dates in the form of dd/mm/yyyy or mm/dd/yyyy.

2.3. Contextual Clues Identification: Contextual phrases such as "registration deadline," "due on," and "final report" suggest that the date is likely in DD/MM/YYYY format (common in UK English contexts). While on the other hand, Phrases like "scheduled for" and "held on" suggest that the date might be in MM/DD/YYYY format (common in US English contexts). The script i wrote also captures context around the date by considering 50 characters before and after each identified date.

2.4. Logical Assumptions:

1. If the day is greater than 12, it must be in DD/MM/YYYY format since no month exceeds 12.
2. If the month is greater than 12, it is in MM/DD/YYYY format since the day cannot exceed 12 in this format.

2.5. Contextual Analysis: If the date cannot be determined through logical assumptions, the script checks contextual clues to decide on the format. If contextual analysis fails to clarify the format, the date is flagged as "Ambiguous."

3. Output and Results

The results are written to a file named KushalChandani_kc07535.txt, which contains the list of extracted dates along with their interpreted formats.

3.1. Final Output:

05/12/2023: DD/MM/YYYY
12/05/2023: DD/MM/YYYY

03/08/2024: DD/MM/YYYY
08/03/2024: MM/DD/YYYY
05/06/2023: Ambiguous

3.2. Interpretation:

1. **05/12/2023:** Interpreted as DD/MM/YYYY - Based on the assumption that the day is greater than 12, indicating the date is in DD/MM/YYYY format.

2. **12/05/2023:** Interpreted as DD/MM/YYYY - Contextual clues have influenced this interpretation, suggesting a US English context.

3. **03/08/2024:** Interpreted as DD/MM/YYYY - Based on contextual terms that fit the DD/MM/YYYY format.

4. **08/03/2024:** Interpreted as MM/DD/YYYY - Likely due to contextual clues suggesting a UK English context.

5. **05/06/2023:** Ambiguous - Both the day and month are ≤ 12, and no clear contextual clues were available to determine the correct format.

4. Challenges and Difficulties Encountered

4.1. Ambiguity in Contextual Interpretation:

Some dates were inherently ambiguous because both day and month values were less than or equal to 12, and the context was not sufficient to determine the correct format definitively. For example, the date 02/02/2023 can be either DD/MM/YYYY or MM/DD/YYYY without any context specifying the intended format.

Solution: Flagging such dates as "Ambiguous" provides a clear indication that the format cannot be determined confidently.

4.2. Handling Contexts with Mixed Clues:

In cases where conflicting contextual clues were present near the date, it became challenging to select a clear format.

Solution: By prioritizing terms associated with one specific format but ultimately flagging any no-context cases as ambiguous.

4.3. Limited Context Around Dates:

The contextual analysis relied on a limited number of characters (50 characters before and after), which sometimes did not provide enough information.

Solution: I approached this problem by adjusting the character range could improve context interpretation, but the current range was chosen to balance performance and accuracy.

Question 2: Identifying the First 10 Merges in a Wordpiece Algorithm [40 points].

1. The first 10 Pairs of Tokens Merged by the Algorithm

1. *Merged: ('1', '##0') -> 10*
2. *Merged: ('o', '##f') -> of*
3. *Merged: ('##f', '##y') -> ##fy*
4. *Merged: ('e', '##x') -> ex*
5. *Merged: ('##m', '##p') -> ##mp*
6. *Merged: ('##q', '##u') -> ##qu*
7. *Merged: ('##b', '##u') -> ##bu*
8. *Merged: ('##mp', '##l') -> ##mpl*
9. *Merged: ('##bu', '##l') -> ##bul*
10. *Merged: ('ex', '##a') -> exa*

2. Challenges Encountered and How They Were Addressed

1. **Handling Special Characters and Punctuation:** Special characters and punctuation were sprinkled in abundance throughout the text, which clearly called for careful tokenization. A custom tokenizer was implemented using regex, where words and punctuation could be effectively separated; this way, all the tokens would be properly treated during the merging process.

2. **Pair Scoring and Merging Logic:** Implementing the scoring mechanism to prioritize merges based on frequency presented a challenge, as it required careful calculation of pair scores using the formula **score = (freq_of_pair) / (freq_of_first_element × freq_of_second_element)**. Debugging steps were crucial to ensure the accuracy of these computations.

3. **Updating Token Splits After Each Merge:** Merging had to be such that token splits could be maintained and after each merge was carried out, the token splits were updated so that merging produced no errors in forming new tokens. A function was designed for merging pairs that updated the token split list, taking into account the most recent merge, to maintain the integrity of the algorithm at each stage.

4. **Frequency Calculation and Tokenization Consistency:** The frequency data of the vocabulary and used splits were consistent. I looped through each token and adjusted frequency counts correctly. Thorough tests on the tokenization and splitting processes allowed for any inconsistencies to be sorted out.

Question 3: Tokenizing Urdu Text [20 points]

1. Python Code

```
import nltk
import re

#Reading the Urdu text from the file
with open('urdu_text_input.txt', 'r', encoding='utf-8') as file:
    urdu_text = file.read()

#Tokenization using NLTK
def tokenize_nltk(text):
    nltk.download('punkt', quiet=True)
    return nltk.word_tokenize(text)

#Tokenization using regex
def tokenize_regex(text):
    return re.findall(r'[\u0600-\u06FF]+', text)

#Getting tokens from different methods
tokens_nltk = tokenize_nltk(urdu_text)
tokens_regex = tokenize_regex(urdu_text)

#Writing the tokens to output files
with open('output_nltk.txt', 'w', encoding='utf-8') as file:
    file.write('\n'.join(tokens_nltk))

with open('output_regex.txt', 'w', encoding='utf-8') as file:
    file.write('\n'.join(tokens_regex))
```

2. Challenges Encountered

1. Handling of Urdu Text Encoding: There was a big problem about properly reading Urdu text from the file. It was done using the explicit setting of the encoding to *'utf-8'* during the reading and writing of files, which preserved the Urdu characters as they are.

2. Tokenization strategies

- **Approach with NLTK:** For the tokenization of text, I used the function nltk.word_tokenize. This approach downloaded the punkt models for the tokenizer, trained on multiple languages; since it didn't have it specifically for Urdu, this might have had an effect on the tokenization accuracy for Urdu text.

- **Regex Approach:** A custom regular expression has been used for tokenizing the text. It focuses on matches within the Unicode range for Urdu characters. In this way, such Urdu-specific tokens are catered to. However, this method needed the regex pattern to be built manually.

3. Outputs and Results

The output and results of both approaches were exactly the same, which shows both libraries can tokenize the Urdu text into individual words or tokens.