NLP Assignment Report

Dataset Preprocessing

Both English and Urdu datasets require normalization, tokenization, stop-word-removal, stemming, lemmatization, and n-gram modelling for preprocessing. Due to this each stage was ensured. The normalisation of the Urdu dataset was hard as 2 columns had to be separately extracted and then combined into a string for each column which was a combination of all rows. Once this was done, only then was the Urdu data ready to be normalized, although still with a few setbacks.

Preprocessing Pipeline Steps

1. Normalization Step

- **Goal**: Convert text to lowercase, remove numbers, and punctuation.
- **Implementation**: Removed unwanted characters.

Removing punctuation and numbers of any kind was achieved by using the string python library and for the Urdu text, I had to add a few extra characters to the English punctuation list as well as Urdu number digits.

2. Tokenization Step

- **Goal**: Split text into individual words for further processing.
- **Implementation**: Using nltk.word_tokenize() for English and customised tokenization for Urdu.

One function in one line was needed to achieve this step's success.

3. Stop Word Removal

- **Goal**: Eliminate common stop words that don't add significant meaning.
- **Implementation**: Using predefined stop words from nltk and manually defining Urdu.

The corpus of stopwords inside the nltk library was used for English, while I had to search the internet and find a text file of Urdu stopwords which I have submitted in the submission portal. Along with this, I also discovered a new NLP Library for Urdu and I tried to use it for this step but found no success.

4. Stemming and Lemmatization

- **Goal**: Reduce words to their root form.
- **Implementation**: Used PorterStemmer and WordNetLemmatizer.

```
from nltk.stem import PorterStemmer, WordNetLemmatizer

stemmer = PorterStemmer()

lemmatizer = WordNetLemmatizer()
```

5. N-gram Modeling

- **Goal**: Capture phrase structure with n-grams of 5.
- **Implementation**: Use nltk.ngrams() to generate n-grams for analysis.

Comparison and Challenges

- **Original vs. Processed Text**: A comparison will highlight how normalization, stop word removal, and stemming affect the text structure.
- **Challenges with Urdu Dataset**: Handling the stop words, and tokenization issues due to different script structures. Along with this, difficulty in tokenization and a lot of time was consumed in data processing.

Poetry Generation Using N-grams

Using N-grams, I generated poetry somewhat mimicking Shakespeare's and Frost's writing styles.

Steps for Poetry Generation

1. Load and Tokenize Corpus

- Load a corpus with Shakespeare and Frost texts, then tokenize

Both text files were converted into separate word chunks.

2. Generate N-gram Models

- Build unigram, bigram, and trigram models using Conditional Frequency Distribution

```
def build_ngram_model(tokens, n):
    ngrams = list(ngrams(tokens, n))
    return ConditionalFreqDist((gram[:-1], gram[-1]) for gram in ngrams)
```

3. Generate Poems

- Select starting words randomly, then predict the most probable next word based on n-gram models.

```
def generate_poem(cfd, start_word, n_lines, words_per_line):
    poem = [ ]
    word = start_word
    for _ in range(n_lines):
        line = [ ]
        for _ in range(words_per_line):
            line.append(word)
            word = random.choice(list(cfd[word])) if word in cfd else '.'
        poem.append(' '.join(line))
    return poem
```

4. Example Output and Analysis

- The generated poems would exhibit stylistic elements from the chosen poet’s work. Examples and analysis would compare the structural differences due to word choice and form influenced by the n-gram order.