N-Gram Language Models for Turkish

Introduction

N-gram language models are statistical models used in natural language processing (NLP) to predict the likelihood of a sequence of words or characters. In these models, an "n-gram" is a contiguous sequence of 'n' items (such as words, syllables, or characters) in a given text. N-gram models analyze text sequences to predict the probability of each item, based on the frequency of previous items in the sequence. These models are valuable in various NLP tasks, including speech recognition, text generation, and language modeling.

Syllable-Based vs. Character-Based Models

This study explores two distinct approaches to modeling Turkish text using N-gram models: syllable-based and character-based. Each method has unique strengths and limitations, particularly relevant to Turkish's linguistic structure:

- **Syllable-Based Model**: Turkish, being an agglutinative language, heavily relies on suffixation to form words. A syllable-based model aligns well with this structure by capturing meaningful sub-word components, making it likely to capture semantic and syntactic relationships in a more coherent way. For example, a syllable-based model could recognize recurring syllables that denote specific grammatical functions, improving its ability to model Turkish's morphologically rich structure.
- **Character-Based Model**: Unlike the syllable-based approach, character-based models treat individual characters as the basic unit of analysis. While this might overlook the semantic context of syllables, it can capture unique morphological patterns across word boundaries, offering benefits in low-resource scenarios or for modeling highly inflected forms. This approach is particularly useful in languages with complex inflectional patterns or orthographic variations.

Objectives of the Homework

The main objectives of this homework assignment include:

1. **Implementation of N-Gram Models**: Develop both syllable-based and character-based N-gram models for Turkish, with configurations for 1-gram, 2-gram, and 3-gram variations.
2. **Smoothing Techniques**: Apply appropriate smoothing techniques to adjust probabilities for unseen n-grams in each model.
3. **Evaluation Metrics**: Evaluate the models using perplexity—a common metric in NLP that measures how well a language model predicts a given sequence.
4. **Sentence Generation**: Generate random sentences using both models to observe the output quality across different configurations and assess the models' fluency and coherence.

Expected Outcomes

Given Turkish's linguistic characteristics, it is anticipated that:

- **Syllable-based models** will outperform character-based models in terms of perplexity, as syllables capture more meaningful linguistic units in Turkish. Higher-order syllable-based models (e.g., 2-gram or 3-gram) are expected to provide more coherent sentence structures due to their ability to capture longer sequences and contextual dependencies.

- **Character-based models** may produce lower perplexity values at the 1-gram level due to the prevalence of high-frequency individual letters. However, for higher-order n-grams, this approach may suffer from increased perplexity due to the lack of semantic coherence across sequences of letters.

  Through this approach, the assignment aims to determine which model and n-gram configuration is most suitable for accurately representing Turkish text data. The results will shed light on whether syllable-based modeling is superior for an agglutinative language like Turkish, or if character-based modeling provides comparable performance across certain tasks. By comparing expected and actual outcomes, the study seeks to evaluate the strengths and limitations of each model in the context of Turkish language modeling.

Design and Implementation

In this section, we will detail the project structure, Makefile usage, and the main functionality of the `main.py` file. This will provide a foundational understanding before diving into the specifics of each module in the following steps.

Project Structure and General Workflow

The project directory is organized to handle various stages, from data processing to model generation and evaluation:

- **data/**: This directory is split into two subdirectories:
  - **raw/**: Contains the unprocessed `wiki_00.txt` file, which is the primary data source.
  - **processed/**: Stores the cleaned and prepared data files, including syllable and character representations in both training and testing formats. Files such as `wiki_00_syllables_train.txt` and `wiki_00_characters_test.txt` are saved here for streamlined access during model training and evaluation.
- **modules/**: Contains modular Python scripts, each responsible for distinct processes:
  - `data_preparation.py`: Responsible for data cleaning and preparation.
  - `split_data.py`: Splits processed data into training and testing sets.
  - `ngram_calculation.py`: Builds and saves n-gram tables with frequency counts.
  - `perplexity.py`: Calculates perplexity values for model evaluation.

    - `text_gen.py`: Generates random sentences based on n-gram tables.
    - `turkish_syllable.py`: Handles Turkish-specific syllable processing tasks.
    - `tables.py`: Generate tables for sample sentences and perplexity data.
- **results/**: Stores outputs generated by the models, including n-gram frequency tables for syllable- and character-based models at different levels, such as `character_ngram_1-gram.txt` and `syllable_ngram_2-gram.txt`.

Makefile

The `Makefile` is a key component for automating various steps in the workflow. Each target in the `Makefile` is designed to simplify complex or repetitive tasks, ensuring a consistent setup and execution:

- **install**: Installs required libraries listed in `requirements.txt`.
- **clean**: Initiates data cleaning and processing, creating syllable- and character-based files for training and testing.
- **ngram**: Generates n-gram models and saves them in the `results` directory. It leverages the `--ngram` argument with `main.py`.
- **perplexity**: Calculates perplexity for each model and each n-gram level, aiding in model evaluation.
- **textgen**: Produces random sentences using trained n-gram models, executed by calling `main.py` with the `--textgen` argument.
- **textgen MAX_LENGTH=100:** Generate random sentences using n-gram models for determined size.
- **table:** Generating table for results. (sample sentences and perplexity results)
- **clear**: Deletes all files from `data/processed` and `results`, resetting the data for fresh processing.

The `Makefile` allows for efficient control over the entire workflow, providing users with quick access to various functions and enabling them to replicate results effortlessly.

Explanation of `main.py`

The `main.py` file is the central script, coordinating data processing, model training, perplexity calculations, and text generation. Using the `argparse` module, `main.py` offers four main options:

1. **Data Cleaning (`--clean`)**:
    - When invoked with `--clean`, this command preprocesses the raw data file and produces syllable- and character-based outputs in the `data/processed` directory.
    - It calls the `process_text_file()` function for cleaning, then splits the cleaned data into training and test sets using `split_data()`.
2. **N-gram Model Generation (`--ngram`)**:
    - This command, triggered with `--ngram`, first checks for processed data files with `check_processed_data_exists()`.

- It then generates n-gram tables (1-gram, 2-gram, and 3-gram) for both syllable and character data. The `calculate_and_save_ngrams()` function builds n-gram frequency tables and applies Good-Turing smoothing before saving the tables in the `results` directory.

3. **Perplexity Calculation (`--perplexity`)**:
   - When `--perplexity` is specified, this command calculates the perplexity of syllable and character models on the test data.
   - Using the `calculate_perplexity()` function, it computes the perplexity score for each n-gram level, helping to assess model effectiveness.
4. **Text Generation (`--textgen`)**:
   - This command generates random sentences based on syllable and character models. The `generate_random_sentence()` function uses the n-gram tables to create sentences of specified lengths, providing an insight into the model's language generation capabilities.

The modular and well-defined structure of `main.py` orchestrates the entire project workflow, ensuring that each part functions cohesively to achieve the overall project goals.

Main Function:

```python
def main():
    parser = argparse.ArgumentParser(description="NLP Pipeline")
    parser.add_argument('--clean', action='store_true', help="Run data cleaning and preprocessing.")
    parser.add_argument('--ngram', action='store_true', help="Run n-gram model generation.")
    parser.add_argument('--perplexity', action='store_true', help="Calculate and display perplexity")
    parser.add_argument('--textgen', action='store_true', help="Generate random sentences using n-gram models.")

    args = parser.parse_args()

    if args.clean: ...


    if args.ngram: ...


    if args.perplexity: ...


    if args.textgen: ...
```

Cleaning and Preprocessing (`--clean` Flag)

The `--clean` flag initiates the data cleaning and preprocessing pipeline, which is crucial for transforming raw text data into syllable-based and character-based formats suitable for n-gram model training. The operations are as follows:

1. **File Paths**:
   - The `raw_file` variable points to the main data source (`./data/raw/wiki_00.txt`).
   - The `syllable_output` and `character_output` variables define output files for syllable-based and character-based processed text (`wiki_00_syllables.txt` and `wiki_00_characters.txt`).
2. **Data Processing**:

- The `process_text_file()` function is called twice, first with `model_type="syllable"` and then with `model_type="character"`. This function, found in `data_preparation.py`, handles all text cleaning, normalization, and format transformations needed for each model type.
- For syllable-based processing, it syllabifies words and preserves punctuation marks. For character-based processing, it splits text into individual characters, separated by spaces.

3. **Data Splitting**:
   - After processing, the `split_data()` function is called to partition the data into training (95%) and testing (5%) subsets. This helps ensure that n-gram models can be trained on one subset and evaluated on another.
   - The function outputs `wiki_00_syllables_train.txt` and `wiki_00_syllables_test.txt` for syllable-based data, and `wiki_00_characters_train.txt` and `wiki_00_characters_test.txt` for character-based data.

Detailed Function Descriptions

`data_preparation.py`

The `data_preparation.py` script includes various text preprocessing steps designed to clean and normalize Turkish language data, particularly for Wikipedia text, which may contain HTML tags, abbreviations, and non-Turkish characters.

- **`process_text_file(file_path, output_file_path, model_type="syllable")`**:
  - This main function orchestrates the entire cleaning and processing workflow, operating line by line on the input file.
  - It uses the `tqdm` library to provide progress feedback, especially useful for large datasets.
  - It invokes `process_text()` to handle specific cleaning steps, such as removing links, converting numbers to words, and syllabifying or character-segmenting based on `model_type`.
  - **Output**: The processed content is saved to `output_file_path`.

```python
# Dosya işleme fonksiyonu (heceleme ve karakter bazlı ayrıştırma için ayrı dosyalar)
def process_text_file(file_path, output_file_path, model_type="syllable"):
    os.makedirs(os.path.dirname(output_file_path), exist_ok=True)

    # Dosya satır sayısını öğrenmek için
    with open(file_path, 'r', encoding='utf-8') as infile:
        total_lines = sum(1 for _ in infile)  # Toplam satır sayısı

    title = "Processing syllable  " if model_type == "syllable" else "Processing character "

    with open(file_path, 'r', encoding='utf-8') as infile, open(output_file_path, 'w', encoding='utf-8') as outfile:
        # tqdm ile ilerleme çubuğu
        for line in tqdm(infile, total=total_lines, desc=title, unit="line", colour="blue"):
            processed_line = process_text(line, model_type=model_type)
            outfile.write(processed_line + "\n")

    return output_file_path
```

- **`process_text(content, model_type="syllable")`**:
  - Executes multiple cleaning steps in sequence:
    1. **Link Removal**: Uses a regex pattern to strip out URLs.
    2. **Special Space Replacement**: Converts non-breaking spaces to regular spaces.

3. **HTML Tag Removal**: Filters out HTML and Wikipedia tags based on an extensive `html_tags` list and a regex pattern.
4. **Non-Turkish Character Removal**: Ensures only Turkish characters and punctuation are retained.
5. **Abbreviation Expansion**: Expands common abbreviations (e.g., `örn.` to `örneğin`).
6. **Number Conversion**: Transforms numeric values into their Turkish word equivalents (e.g., `3` to `üç`).
7. **Lowercasing**: Converts uppercase letters to lowercase, respecting Turkish character mappings.
8. **Segmentation**: Based on `model_type`, calls either `syllabify_text_with_punctuation()` or `char_based_text()` for syllable or character segmentation.

   Output: Returns the cleaned and segmented text for line-by-line processing.

```python
def process_text(content, model_type="syllable"):
    content = remove_links(content)
    content = remove_special_spaces(content)
    content = clean_html_tags(content)
    content = remove_non_turkish_characters(content)
    content = expand_abbreviations(content)
    content = convert_num_to_word(content)
    # content = replace_turkish_characters(content)
    content = turkish_lower(content)

    if model_type == "syllable":
        content = syllabify_text_with_punctuation(content)

    elif model_type == "character":
        content = char_based_text(content)

    return content
```

- **Helper Functions**:
  - **`remove_non_turkish_characters(content)`**: Filters out characters that are not commonly used in Turkish text.
  - **`replace_turkish_characters(content)`**: Normalizes Turkish characters by mapping them to basic Latin equivalents (optional).
  - **`clean_html_tags(content)`**: Removes HTML tags while preserving text.
  - **`expand_abbreviations(content)`**: Uses a dictionary to replace common abbreviations with their full forms.
  - **`syllabify_text_with_punctuation(content)`**: Applies syllable segmentation while preserving punctuation.
  - **`char_based_text(content)`**: Splits text into individual characters separated by spaces.

`split_data.py`

The `split_data.py` script uses Scikit-Learn's `train_test_split` to partition data files into training and testing subsets.

- **`split_data(file_path, train_file_path, test_file_path, test_size=0.05)`**:
  - **Input**: Reads a text file, then splits it based on the specified `test_size` (5% for testing by default).
  - **Output**: Writes the resulting training and test data to `train_file_path` and `test_file_path`.
- Helper functions `read_file()` and `write_file()` handle file I/O, including line reading and writing, to prevent any character encoding issues.

```python
from sklearn.model_selection import train_test_split


def read_file(file_path):...


def write_file(data, file_path):...


def split_data(file_path, train_file_path, test_file_path, test_size=0.05):
    data = read_file(file_path)
    # Veriyi %95 eğitim, %5 test olacak şekilde ayır
    train_data, test_data = train_test_split(data, test_size=test_size, random_state=42)

    write_file(train_data, train_file_path)
    write_file(test_data, test_file_path)


if __name__ == "__main__":...
```

`turkish_syllable.py`

The `turkish_syllable.py` script contains functions specific to Turkish syllabification, which is essential for segmenting Turkish words accurately. I couldn't find an adequate library for this part, so I wrote it all by myself.

- **`syllabify(word)`**:
  - **Description**: Segments Turkish words into syllables based on a set of phonological rules.
  - **Logic**: Implements rule-based syllabification by checking for vowels and consonants and handling syllable boundaries (e.g., `sessiz - sessiz - sesli` patterns).
  - **Output**: Returns a list of syllables for each word, maintaining Turkish phonological rules.
- **`syllabify_text_with_punctuation(content)`**:
  - **Description**: Uses `syllabify(word)` to process each word in a sentence while preserving punctuation as separate tokens.

- **Output**: Returns a string where words are replaced by their syllable-separated forms and punctuation is retained.

```python
def syllabify(word):
    syllables = [] # -> heceler
    current_syllable = "" # -> anlık hece

    i = 0

    while i < len(word):

        char = word[i]
        current_syllable += char

        if is_vowel(char): # sesli - ...

            if (i + 1) < len(word) and not is_vowel(word[i + 1]): # sesli - sessiz - ...

            else: # sesli - sesli - ...

        else: # sessiz - ...

            if (i + 1) < len(word) and is_vowel(word[i + 1]): # sessiz - sesli - ...

            else:
            i += 1

    # Son bir hece kalmışsa onu da ekle
    if current_syllable and len(word) != 3: # 4 harfli özel durumlar için eklenmiştir (sa-at)

    return syllables
```

N-gram Model Generation (`--ngram` Flag)

The `--ngram` flag is responsible for generating n-gram models for both syllable-based and character-based data using the processed training data. The n-grams produced help in calculating the probabilities required for the language model, and Good-Turing smoothing is applied to account for unseen n-grams, improving the model's accuracy.

1. **Checking Processed Data**:
   - The function `check_processed_data_exists()` ensures that the necessary preprocessed data files exist in the `processed` directory (`wiki_00_syllables_train.txt` and `wiki_00_characters_train.txt`). If they are absent, a message prompts the user to run `--clean` first.
2. **Defining File Paths**:
   - Two main files serve as inputs for this step:
     - `syllable_train_file` for syllable-based n-gram calculations.
     - `character_train_file` for character-based n-gram calculations.
   - Output file prefixes are set for each n-gram type, such as `syllable_ngram_` and `character_ngram_`, which allow the n-gram files to be stored in a structured way.
3. **N-gram Table Generation**:
   - The code iterates over values of `n` (from 1 to 3) for unigram, bigram, and trigram models:

- **Syllable-based n-grams**: The `calculate_and_save_ngrams()` function is called with the syllable-based training data and output prefix for each `n`.
- **Character-based n-grams**: The same function is called with character-based data and prefix.

4. **Completion Confirmation**:
   - Upon completion of the loop, a message is printed indicating successful generation of n-gram models.

```
if args.ngram:

    if not check_processed_data_exists():

        print("\033[33m<-> Processed data not found. Please run with --clean first.\033[0m")
        return

    print("\033[33m<-> Running n-gram model generation...\033[0m")

    # File paths
    syllable_train_file = "./data/processed/wiki_00_syllables_train.txt"
    character_train_file = "./data/processed/wiki_00_characters_train.txt"
    # Output paths
    syllable_ngram_output = "./results/syllable_ngram_"
    character_ngram_output = "./results/character_ngram_"

    # Store n-gram tables in dictionaries
    syllable_ngram_tables = {}
    character_ngram_tables = {}

     # Generate and save n-grams for syllable-based data
    for n in range(1, 4):
        syllable_ngram_tables[n] = calculate_and_save_ngrams(syllable_train_file, syllable_ngram_output, n)

    # Generate and save n-grams for character-based data
    for n in range(1, 4):
        character_ngram_tables[n] = calculate_and_save_ngrams(character_train_file, character_ngram_output, n)

    print("\033[33m<-> N-gram model generation completed.\033[0m")
```

Detailed Function Descriptions

`calculate_and_save_ngrams(file_path, output_prefix, n)`:

This function is central to the `--ngram` flag's functionality and performs the following steps:

- **Loading Data**:
  - It first loads data from `file_path` using `load_data()` to retrieve lines of preprocessed text.
- **Building N-gram Table**:
  - It then invokes `build_ngram_table()` with the data and n-value to generate the n-gram frequency table. This table is stored in a dictionary, where keys are n-grams (tuples of `n` tokens), and values are their occurrence counts in the text.
- **Applying Smoothing**:
  - The Good-Turing smoothing technique is applied via `apply_good_turing_smoothing()` to adjust the frequency of observed n-grams. Smoothing handles cases of zero-frequency n-grams (unseen n-grams) and produces more reliable probability estimates by adjusting observed frequencies.
- **Saving N-gram Table**:
  - The `save_ngram_table()` function stores the resulting smoothed n-grams and their probabilities in a specified output file (`output_path`).

`ngram_calculation.py`

This module contains various functions for building, smoothing, and saving/loading n-gram models.

- `generate_ngrams(text, n)`:

- **Description**: Converts the input text into a list of n-grams by iterating over tokens in the text.
- **Logic**: The function splits the text into tokens (words or characters) and forms tuples of `n` tokens, ensuring that all overlapping n-grams are captured.
- **Output**: Returns a list of n-gram tuples.

- `build_ngram_table(data, n)`:

- **Description**: Builds an n-gram frequency table and a frequency count dictionary.
- **Logic**: For each line in `data`, the function splits tokens and generates n-grams. Each n-gram's frequency is recorded in `ngram_table`, while `freq_count` counts the number of n-grams at each frequency level.
- **Output**: Returns `ngram_table` with n-gram frequencies and `freq_count` for frequency distribution, both essential for Good-Turing smoothing.

- `apply_good_turing_smoothing(ngram_table, freq_count)`:

- **Description**: Adjusts n-gram frequencies using Good-Turing smoothing to handle unseen n-grams.
- **Logic**: For each n-gram frequency `f`: - If a higher frequency (f+1) exists in `freq_count`, the smoothed frequency is calculated as `(f+1) * (count of f+1) / (count of f)`. - Otherwise, the original frequency is used.
- **Normalization**: The smoothed frequencies are divided by the total n-gram count to calculate relative probabilities.
- **Output**: Returns a dictionary of n-grams with their smoothed probabilities.

- `save_ngram_table(ngram_table, file_path)`:

- **Description**: Saves the n-gram table to a specified file.
- **Logic**: Iterates over `ngram_table` and writes each n-gram and its smoothed frequency to a file, with each n-gram token separated by spaces.
- **Output**: Creates an output file that stores n-grams with their smoothed frequencies for later use.

- `load_data(file_path)`:

- **Description**: Reads lines from a specified file path.

- **Logic**: Opens the file in read mode and returns a list of lines (processed text).
- **Output**: Returns the list of processed data lines, used as input for building n-grams.

- `load_ngram_table(file_path)`:

- **Description**: Loads an existing n-gram table from a file.
- **Logic**: Reads each line in the file, splits the n-gram tokens, and parses the smoothed frequency value.
- **Output**: Returns a dictionary representing the n-gram table, useful for generating text or calculating perplexity.

Perplexity Calculation (`--perplexity` Flag)

The `--perplexity` flag is designed to measure the effectiveness of the generated n-gram models in predicting test data. By calculating perplexity, we assess the uncertainty of the language model regarding the test data. Lower perplexity scores indicate a better model fit, making this metric an important indicator of model performance.

1. **Loading N-gram Tables**:
    - For each n-gram level (unigram, bigram, trigram), the code loads precomputed n-gram probabilities from previously saved files (`syllable_ngram_{n}-gram.txt` and `character_ngram_{n}-gram.txt`). These files are loaded into dictionaries (`syllable_ngram_tables` and `character_ngram_tables`) for syllable-based and character-based models.
    - The `load_ngram_table()` function retrieves these tables, containing n-grams and their smoothed probabilities. These probabilities are necessary for calculating the likelihood of each n-gram in the test data.
2. **Loading Test Data**:
    - The test data for both syllable-based (`wiki_00_syllables_test.txt`) and character-based (`wiki_00_characters_test.txt`) models is loaded from the `processed` directory using `load_data()` to provide sentences for perplexity calculation.
3. **Calculating Perplexity**:
    - For each n-gram level (1 to 3), `calculate_perplexity()` computes perplexity values separately for syllable-based and character-based models, iterating over each sentence in the test data.
    - Perplexity is calculated by accumulating log-probabilities for all n-grams in the test set and using these probabilities to compute the model's uncertainty regarding unseen data.
4. **Output of Perplexity Scores**:
    - Perplexity values are printed for each n-gram model, enabling an easy comparison of the effectiveness of each model in predicting new data.

```python
if args.perplexity:

    # Load n-gram tables from files if needed
    syllable_ngram_tables = {}
    character_ngram_tables = {}

    for n in range(1, 4):
        syllable_ngram_tables[n] = load_ngram_table(f"./results/syllable_ngram {n}-gram.txt")
        character_ngram_tables[n] = load_ngram_table(f"./results/character_ngram_{n}-gram.txt")

    # Calculate perplexity with data and n-gram tables used in education
    syllable_test_file  = "./data/processed/wiki_00_syllables_test.txt"
    character_test_file = "./data/processed/wiki_00_characters_test.txt"

    syllable_test_data  = load_data(syllable_test_file)
    character_test_data = load_data(character_test_file)

    print("\033[33m<-> Calculating perplexity for syllable-based model...\033[0m")
    for n in range(1, 4):
        perplexity = calculate_perplexity(syllable_test_data, syllable_ngram_tables[n], n)
        print(f"\033[33mSyllable-based {n}-gram perplexity:\033[0m {perplexity}")

    print("\033[33m<-> Calculating perplexity for character-based model...\033[0m")
    for n in range(1, 4):
        perplexity = calculate_perplexity(character_test_data, character_ngram_tables[n], n)
        print(f"\033[33mCharacter-based {n}-gram perplexity:\033[0m {perplexity}")
```

Detailed Function Descriptions

`calculate_perplexity(test_data, ngram_probs, n)`
The primary function for calculating perplexity operates as follows:

- **Input Parameters**:
  - `test_data`: List of strings, where each line represents a sentence in the test data.
  - `ngram_probs`: Dictionary of n-grams with their smoothed probabilities.
  - `n`: The n-gram level for which perplexity is being calculated (1 for unigram, 2 for bigram, etc.).
- **Logic**:
  - **Initialize Counters**: `total_log_prob` accumulates the log-probabilities for all n-grams, while `total_ngrams` counts the number of n-grams in the test set.
  - **Iterate Through Sentences**:
    - Each sentence is tokenized, and the function iterates over each possible n-gram in the sentence.
    - For each n-gram, it checks if the n-gram exists in the `ngram_probs` dictionary:
      - **Found**: Adds the log of the n-gram's probability to `total_log_prob`.
      - **Not Found**: Assigns a very small probability (`1e-10`) to avoid zero probability, adding its log to `total_log_prob`.
    - Updates `total_ngrams` by counting each processed n-gram.
  - **Calculate Average Log Probability**: `avg_log_prob` is computed as `total_log_prob / total_ngrams`.
  - **Perplexity Calculation**: Using the formula `perplexity = exp(-avg_log_prob)`, the perplexity score is derived.
- **Output**:
  - Returns the perplexity score, which indicates the model's performance on the test data. Lower values signify a better model.

Supporting Functions in `perplexity.py`

- `load_ngram_probabilities(file_path)`:
  - **Description**: Loads n-gram probabilities from a file, structured as n-grams with their probabilities on each line.
  - **Logic**: Reads each line, splits the n-gram and its probability, and stores it in a dictionary with the n-gram tuple as the key.
  - **Output**: Returns a dictionary of n-grams and their probabilities, used in perplexity calculations.

Random Sentence Generation (`--textgen` Flag)

The `--textgen` flag enables the system to generate random sentences by leveraging the trained n-gram models. This functionality is used to assess how well the models capture the structure and flow of the Turkish language by observing the coherency of generated sentences. For each model (syllable-based and character-based), sentences are generated for 1-gram, 2-gram, and 3-gram models, and displayed as output.

Steps Involved in Sentence Generation

1. **Loading N-gram Models**:
   - The code begins by loading n-gram models for both syllable-based and character-based configurations. Each n-gram model (1, 2, and 3) is loaded from saved files using `load_ngram_table()` and stored in dictionaries (`syllable_ngram_table` and `character_ngram_table`).
   - This setup allows the function `generate_random_sentence()` to access the precomputed probabilities for each n-gram during sentence generation.
2. **Generating Sentences**:
   - For each n-gram model (1 to 3), `generate_random_sentence()` is called, generating sentences with a specified maximum length (default: 15 tokens). These sentences are printed for both syllable-based and character-based models.
   - Each sentence generation leverages the n-gram probabilities to produce a sequence of tokens (syllables or characters), aiming to form coherent phrases while also limiting punctuation frequency and preventing excessive repetition of the same syllables.

Detailed Function Descriptions

`generate_random_sentence(ngram_table, start_context=tuple(), max_length=15)`

The primary function for generating random sentences works as follows:

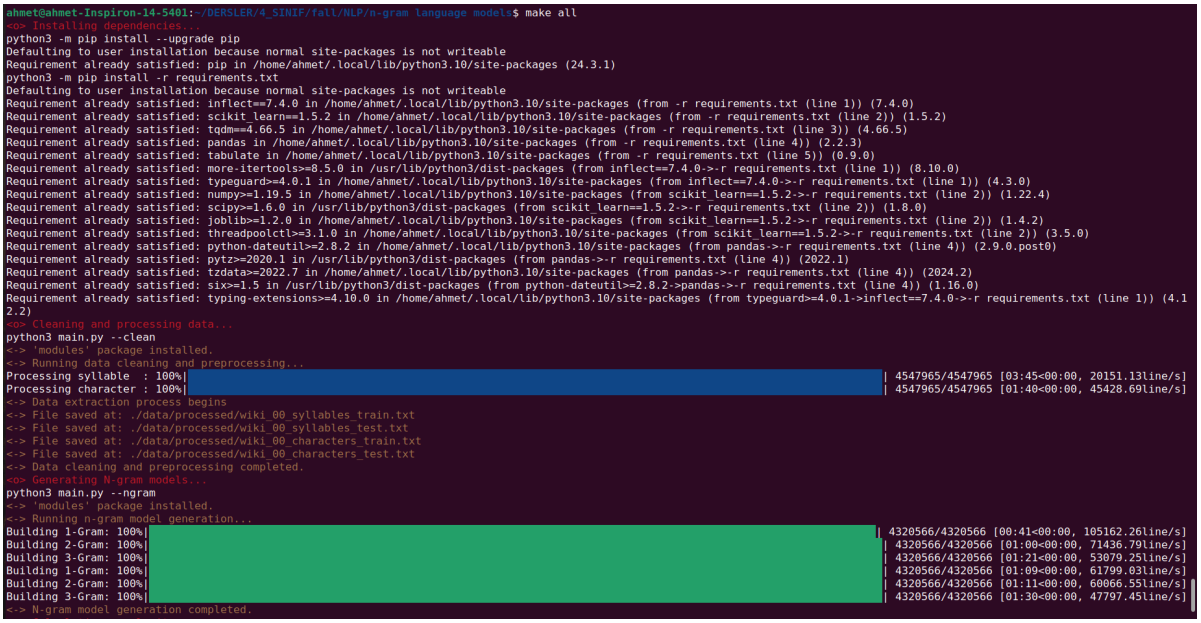
- **Input Parameters**:
  - `ngram_table`: The n-gram model dictionary, where each n-gram has a probability or frequency score.
  - `start_context`: The starting context, which is initially an empty tuple. If empty, the function selects a frequent starting n-gram from the model.

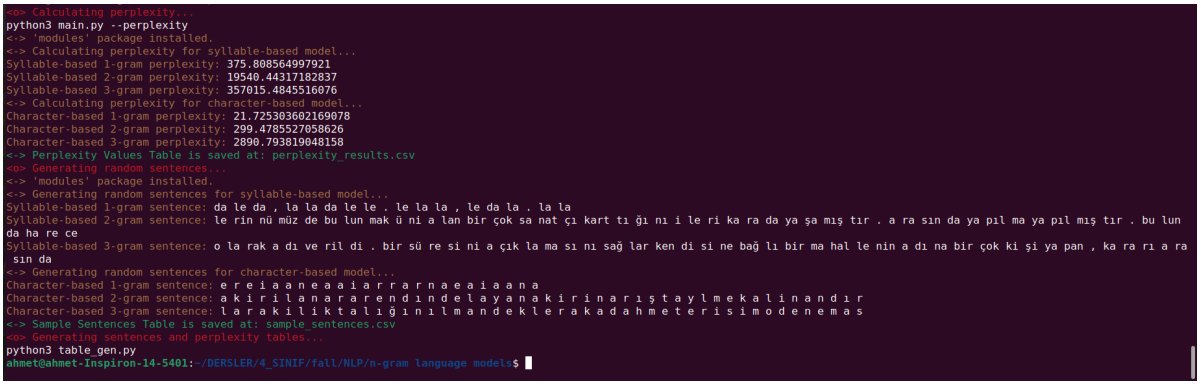
    - `max_length`: The maximum number of tokens in the generated sentence.
    - `n`: The n value of the n-gram model.
- **Logic**:
    - **Selecting Start Context**:
        - If `start_context` is empty, the function defaults to a frequent initial n-gram without punctuation to start the sentence.
    - **Sentence Generation Loop**:
        - Using `current_context`, the function iterates over `max_length - len(start_context)`, appending one token per iteration based on the following:
            - **Top N-grams Selection**: Retrieves the top 5 most probable n-grams using `get_top_n_grams()` based on `current_context`.
            - **Next Token Selection**:
                - If top n-grams are available, the function randomly selects a next n-gram from these, weighted by their probabilities.
                - The chosen n-gram's last token is appended to `sentence`, and `current_context` is updated to the latest tokens.
            - **Punctuation Control:** Limits consecutive punctuation marks by enforcing a cooldown `(punctuation_limit)` before another punctuation mark can be added.
            - **Repetition Prevention:** Avoids adding a syllable or word if it repeats more than twice consecutively in the sentence.
            - **Fallback**: If no suitable n-grams are found, the function selects a random n-gram from the model to continue sentence generation, which prevents the sentence from stalling.
    - **Output**:
        - After reaching `max_length`, the function returns `sentence` as a single string, combining tokens with spaces.

Supporting Functions in `text_gen.py`

- **`get_top_n_grams(ngram_table, current_context)`**:
    - **Description**: Retrieves the top K most probable n-grams from the model given the current context. This function is crucial for generating realistic sentences, as it identifies likely continuations based on n-gram probabilities.
    - **Output**: Returns a list of probable n-grams, allowing `generate_random_sentence()` to construct the next part of the sentence with contextual relevance.

A sample console output for `make all` command:





Results and Tables

Sample Sentences

	Model	N-Gram	Sentence 1	Sentence 2
0	Syllable-Based	1-Gram	da la le la le . da le da . da la la le . da la da da , da da le da . le le da la . le la da . le la la . la la da le , da da le le . la da da . le la le da da le da . la le da la la , le la da , da la la da . le la	le la da la da . da le da la la . le la la . da da la le , la da le la la da la . le le la la . da da la . la le da . le la le da da . la le le . da da le le , la le le . le la da le . da le la le , da la le da , da da le la . le

	Model	N-Gram	Sentence 1	Sentence 2
1	Syllable-Based	2-Gram	le re ti ği ve ya da i le rin ço cuk ba şa yan lar dan son ra sı nın da ki ye ti mi si ne ti ril di ği ni ver me ye ni ve a lan ma sı na li bir şe kil de , a ra fın dan o la nı sı ra fın da i ki li ne de ya şa rı nı i se zo nu cu dur . yı lın da i le me ye ni den ge le rin de bu ra da ya şa ğı ve ya pı lan mış tır . bu ra sın dan son ra sı na gö re ti ği ve bu lun ma sı na da ha va , a lan dı . bu lu ğu nu sun da , a ra sın dan bi lir . yı i ki şi o yun da , ka ra sın da ya yın lan dı . bu lu şan bir a ra fın dan son ra fın dan bi lir . o lup bu lu nur lar , bu lu var dır . bu a ra sı na ka ra fın dan son yıl dız o la	le ri ni a ra fın dan bi ri ni den bi le ri ne si ni ver me si te le ri ni den o luş tur . a ra fın da ki e lekt ron lar da i ki a ra fın dan o la rı na ka na ka dar t o la ma sı nı i le me si ne bağ lı bir le ri ne bağ lı ğı na da ki ye ti . bu a dı . bu lu ğu o la rak ta dır . bu lu nan ilk o yun da , bu nun da ha son o la rak da i le di . o lan bir lik le ri nin ya pı la rı na ka ra fın dan bi le ri ka ra sın da ki e dil me ri ne de ki ye ni a lan bir çok sa yı lı ğa be le ri nin a i çin de ki şi sel o lan ma dı . bu lu var dır . bu lu şan mak ta ri nin e ği ne km u zak laş ma sı na da ha re ce ği tim o la
2	Syllable-Based	3-Gram	o la rak a nı la ma sı nın ar dın dan i ti ba rı i le ev li ve ü ze rin de yer al mak ta o lup , i ki li se si ve ka ra sal ik lim et ki li se si , bir le şik dev let le ri , sa de ce ği gi bi i sim li bir ye re ya pı sı na kar şı laş ma sı , i kin ci e dad ku lüp le rin ya şa mış , bu ra ya pı lan bir nü fus sa yı mı nı i çe ri sin de ki ya nın e n i yi bir per de si ni is ter se de di ko nu su dur . pro fe sör lü ğü nün i kin ci e şi nin ya nı sı ra la ma sı na ka ra i ki ço cuk la rın da bu lu nan ve ka dın lar , i se a çık a nah tar ke li me sa jı gön de ren , ka ra sal ik lim et ki a ra sın da , ya pım lar la	o la rak a ta kım la rı nın ve ya a tıl dı ğı bir dö nem de ki ba kan lı ğı , e ği li mi , ka ra de niz den yük sel ti le rin de i se e ği tim den ya yım lan dı . da ha a z sa yı mı na ka ra de re ce ye ka rar ver mek te dir . ya rı sın da ki ka da şı nın ya nın e n çok sa yı mı na gö re nek ve ye mek le ri ne ya kın dan il gi len me si a ra sın da ki ka sa sı na rağ men , bu i ki si de o la rak gö rü şü nü len bir stad yum da ya lı dır . kö yün ik li mi et ki le ri ni ge nel lik le ri ni ge nel lik le rin de , il köğ re tim yı lın da , a na do lu ve ya pı sı nı sağ la yan yol as fal t o lup , ka li for ma giy di . an cak

	Model	N-Gram	Sentence 1	Sentence 2
3	Character-Based	1-Gram	r e n i a i e r n r i i a e n i r n a i n n e e r n i e a e n a e r n e a e n i n e a e a a r a i a r n r a a i i e e a a n e a r a a r n i a e n i n e n a n a i e n e i a n r e i a r n r e a i i e e	r n a e e n a a e e r r n i r n a n a r e n i i r r e e r e r r a r i n e i a i a r n r a i e n a n i e e i r a a e e i n i n a a e n n a r r a i e i e i e r a a n r a r e n r e a a i i e e n a r r
4	Character-Based	2-Gram	a r ı l a l ı n e r a n e k l e s i n d a n ı r a k e r e r i r a r e r a k e r i l e k i n d ı r ı n d i n d e n i l e s i l e n d a k a r a r i r i r a r a r a k t ı l ı n ı n ı n i n a r . a l m a y l a k e k a k l a k a r . s i k i n ı l e r a n e n ı k i n e r i r a n e s a r i r . a n ı r a l a k l e r . b a n d a l a n d i l a l e n d a y o l ı n a n ı l a r i l e r i r i l e n e l a n e	a l i l e n i n d a n i r a r a n i k a y o y o n i k e n i r i n d e k t e n i l a r . b i r e s t a k i n ı ş t a k e k a k l e k e r ı r ı k e n a r ı l ı n ı n a r e s e s i r a n d e l a k l a n d u l m ı k i l i n ı n e n e k t a l a r ı r e k t e l i r e k i n e r a l a l ı n ı k l a y e r ı n d ı k l m e r i s e k a k l m e n a r . k a y e r i n i n a k a r ı n e n e r i s t i l m i k a k i n
5	Character-Based	3-Gram	l a n d a n l e r a k ı r . k u l a r d e v i y a r i s t e d i r i l m a s o n r a s t i k l e n d i l k e z i s e l d ı . d ü z d e k t a r ı n d a y r u n a n a k a d i l e r d i r . b i r l a r a s a y a n a k t a r ı y ı l ı ğ ı n d a n a n d a n l i n i k l e n d a b e k t u r d i . a y a y a r a n s a l ı k t e m d e v e b e l i n e k i ş l a n d e r i n d a h a y ı l ı k a n l ı n ı n a n	l a r a k a r a k i s i n i k i n i n e k l a r ı c a k i n g ü n e l i k t i l m e r i h a s ı n d a n ı l ı k e n i n d a b i r . a m a s ı n d a n ı l a r ı n a d ı r . b u r a l a r ı n d e k a r i n i k t i r d i l e r i n g e r e t i r . a l i k i n i s t a r a n l e n b e y i n a n d a n d e k t e r l a r a k ı m ı ş t u r . y ı l m ı ş t u r i k o y n a l e r a n a l i k i l m i s t i r l

Perplexity Results

	Model	N-Gram	Perplexity
0	Syllable-Based	1-Gram	375.80
1	Syllable-Based	2-Gram	19540.44
2	Syllable-Based	3-Gram	357015.48
3	Character-Based	1-Gram	21.72
4	Character-Based	2-Gram	299.47
5	Character-Based	3-Gram	2890.79

Analysis and Conclusion

Model Performance Analysis Based on Perplexity Values

The performance analysis of syllable-based and character-based N-gram models is essential to understanding their suitability for modeling the Turkish language. Perplexity values serve as a core indicator here, where lower values imply higher certainty and predictive accuracy.

Model	1-Gram Perplexity	2-Gram Perplexity	3-Gram Perplexity
Syllable-Based	375.81	19540.44	357015.48
Character-Based	21.73	299.48	2890.79

1. **Character-Based Model**:
    - This model maintains lower perplexity across all n-gram levels due to the limited set of Turkish characters (28 letters and a few punctuation marks).
    - With fewer unique elements to predict, the model can confidently choose from a constrained set, reducing uncertainty. Thus, even as n increases, the growth in perplexity is moderate and manageable.
    - The character-based model is suitable when accuracy and predictability are priorities, as it can handle shorter contexts without significant losses in certainty.
2. **Syllable-Based Model**:
    - The syllable-based model, while advantageous for capturing morphological details, exhibits significantly higher perplexity values.
    - Turkish is morphologically rich, with many possible syllables. The model's high perplexity values are expected given the extensive variety and complexity of syllables, especially in multi-syllabic words.
    - Although the syllable model holds potential for generating more semantically rich text, it struggles to predict accurately without a sufficient context, leading to higher perplexity.
3. **Trade-Offs**:
    - The trade-off between character and syllable models revolves around **prediction certainty** and **linguistic richness**. Character models offer simpler, more predictable outputs, while syllable models bring depth but at a cost of increased perplexity.
    - For Turkish, which relies heavily on morphology, syllable-based models could yield richer, more meaningful phrases, but their practical utility depends on the required accuracy and context length.

Quality of Generated Sentences

When examining the sentence quality across different models and n-gram sizes, several patterns emerge:

1. **1-Gram Syllable-Based Model**:
    - The instructions specify selecting one of the top five probable n-grams at each step, which significantly impacts sentence quality in the 1-gram syllable-based model.
    - In this setup, the five highest-frequency syllables—`la`, `le`, `da`, `.` and `,`—often appear in repetitive patterns. This limited selection contributes to the repetitive, unnatural sentences generated by the 1-gram syllable model, as these syllables form simple, repetitive sequences with minimal semantic coherence.
2. **Higher N-Gram Levels (2-Gram and 3-Gram)**:
    - As n increases to 2 and 3, the syllable-based model's quality notably improves. The context provided by multi-syllable patterns reduces the repetition seen in the 1-gram output.
    - Similarly, character-based models benefit from increased n-gram levels, producing sequences that appear more coherent and linguistically plausible. However, as expected, the syllable-based model still yields more semantically relevant sentences due to the use of entire syllables, which align more closely with Turkish word formation.
3. **Comparing Character-Based and Syllable-Based Outputs**:
    - The character-based model, while yielding coherent character sequences, lacks the richness seen in syllable-based outputs, especially at higher n-grams. The character model is suitable for tasks where simpler, predictable text generation is needed, while the syllable-based model is preferable when the output requires deeper morphological representation.

Conclusion

Based on the results:

- **Character-Based Model** is advantageous for applications requiring **lower perplexity** and **higher prediction certainty**, especially at 1-gram and 2-gram levels.
- **Syllable-Based Model** becomes more beneficial at **higher n-grams**, as it captures the morphological structure of Turkish more effectively, though it requires longer contexts to manage perplexity.

Overall, the **syllable-based 3-gram model** emerges as the most suitable for modeling Turkish text where a more realistic language representation is required. It balances coherence with linguistic depth, though it may not be as efficient for simpler applications where the character-based model would suffice. This project illustrates that the choice between syllable and character n-grams hinges on the specific requirements of the task—character models for predictability and syllable models for rich language representation.

Table on LLM Usage

The following table indicates the parts of this submission that were generated or assisted by ChatGPT, alongside the sections that were originally written by me.

Section	Description of LLM Assistance	Original/Assisted
Makefile Flags with `argparse`	Guidance on `argparse` usage tailored for Makefile flags and customizations	Assisted
`split_data()` function	Assistance in choosing libraries and functions, along with parameter explanations for file splitting	Assisted
N-gram Smoothing Calculations	Detailed assistance in designing the smoothing algorithms used in N-gram probability adjustments	Assisted
Perplexity Calculations	Help with algorithmic structures and code generation for perplexity calculations	Assisted
`tqdm` Library Customization	Explanation of customization options for `tqdm` used to tailor output displays	Assisted
Regex (`re`) Usage	Guidance on regex patterns and usage within functions	Assisted
General Code and Function Development	Primary development and coding, with ChatGPT consulted for debugging and solutions for specific issues	Mostly Original
Report Writing and Markdown Formatting	Minor assistance in structuring certain sections and tailoring specific explanations within the report	Assisted

This table reflects a breakdown of where large language model assistance was utilized in developing specific parts of the project. Although much of the core development was done independently, ChatGPT provided support for technical issues, library guidance, and specialized coding advice, particularly for parts requiring advanced implementations.