N-Grams approach for hangman:

Methods

1. **Term Frequency**
   - Computes the term frequency of each letter in a list of words, adjusted with logarithmic scaling.
2. **Generate N-grams**
   - Produces all n-grams (substrings of length n) for a given word to capture common patterns.
3. **Find Incorrect Guesses**
   - Identifies letters that have been guessed incorrectly so far.
4. **Check All Letters in Word**
   - Validates if all given letters are present in a dictionary word.
5. **Match Phrase**
   - Matches a partially known word (with unknowns represented by dots) against a dictionary word.
6. **All Possible Subproblems**
   - Generates all possible subproblems (n-grams with at least one unknown letter) of a given length.
7. **Contains Incorrect Guesses**
   - Checks if a word contains any previously incorrect guesses.
8. **Generate All N-grams**
   - Builds a dictionary of n-grams for all words in the full dictionary, tracking their frequencies.
9. **Prune Dictionary**
   - Removes words from the dictionary that are longer than the current word being guessed.
10. **Guess the Next Letter (guess method):**
    - Implements the core guessing strategy:
      - Cleans and prepares the input word.
      - Prunes the dictionary based on word length.
      - Identifies incorrect guesses.
      - Finds potential solutions through subproblems if necessary.
      - Calculates term frequency for candidate words.
      - Selects the letter with the highest frequency that hasn't been guessed yet..

Implementation Steps

1. **Generate N-grams:**
   - For each word in the dictionary, generate all possible n-grams of lengths ranging from 3 to the word's length.
   - Store the frequency of each n-gram in a hashmap.

2. **Prune Dictionary:**
    - For a new word to guess, prune the dictionary to exclude words longer than the target word.
    - Identify subproblems (n-grams with unknown letters) within the target word.
3. **Match and Predict:**
    - Find matching dictionary words for the subproblems.
    - If sufficient matches are found, predict the most frequent character.
    - If not, recursively generate and solve subproblems until enough matches are found.
4. **Iterative Guessing:**
    - Continue guessing letters based on term frequency and subproblem matching until the game is resolved.

Implementation Example

For a given dictionary word, generate all possible n-grams and store their frequencies in a hashmap.

Example Word**:** "hangman"

N-grams:

- Length 3: ["han", "ang", "ngm", "gma", "man"]
- Length 2: ["ha", "an", "ng", "gm", "ma", "an"]
- Length 1: ["h", "a", "n", "g", "m", "a", "n"]

Stored Frequency (Hashmap):

```
{
    "han": 1, "ang": 1, "ngm": 1, "gma": 1, "man": 1,
    "ha": 1, "an": 2, "ng": 1, "gm": 1, "ma": 1,
    "h": 1, "a": 2, "n": 2, "g": 1, "m": 1
}
```

For a new word to guess, prune the dictionary to exclude n-grams longer than the word length.

Example Partial Word: "_ _ a _ n _" (6 letters)

Pruned Dictionary: Only n-grams of length 6 or shorter are kept.

Identify subproblems from the partial word. Match these against the pruned dictionary to find possible solutions.

Example Partial Word**:** "_ _ a _ n _"

Possible Subproblems:

1. `..a.n.` (where dots represent unknowns)
2. `.a.n..` (alternative subproblem)

For each subproblem, find matching words from the pruned dictionary.

Matching Words:

- `..a.n.`: "hangman"
- `.a.n..`: "banana" (assuming "banana" is in the dictionary)

Term Frequency in Matching Words:

```
{
    "h": 1, "a": 5, "n": 3, "g": 1, "m": 1, "b": 1
}
```

Guess the letter with the highest frequency from the matching words that hasn't been guessed yet.

Guessed Letters**:** []

Highest Frequency Letter: 'a'

First Guess: 'a'

With the server's response, update the partial word and repeat the process of pruning the dictionary, generating subproblems, matching, and guessing until the word is guessed or attempts run out.

Assuming the server updates the word to `_ a _ _ n _` after the first guess.

New Subproblems:

1. `_a..n_`
2. `.a_n..`

Continue matching, finding frequencies, and guessing until the game is resolved.

LSTM Method (Failed attempt) -

It failed because LSTM took too long to train on the given data and my system kept crashing. Further, after training on smaller data it gave me a loss of close to 1.5.

1. Generate and Store N-grams

- Read the full dictionary file.
- Generate all possible n-grams for each word and store them in a hashmap.

2. Create Augmented Data

- Create a DataFrame containing word lengths, vowels present, and unique character counts.
- Filter out words that do not meet certain criteria (e.g., words with fewer than 3 unique characters or no vowels).

3. Create Masked Dictionary

- **Permute All:**
  - Generate all permutations of a word by masking different letters (replacing them with underscores).
- **Permute Consonants:**
  - Create masked versions of the word by replacing consonants while keeping vowels in place.
- **Masked Dictionary:**
  - For each word, generate a list of all possible masked versions and store these in a dictionary.

4. Vowel Handling

- Calculate the prior probabilities for each vowel in words of different lengths.
- Store these probabilities for use in making educated guesses about vowels.

5. Train the BiLSTM Model

- **Encode Words:**
  - Convert words into numerical format using a character mapping.
  - Encode input words (masked versions) and target words (actual words).
- **Convert to Tensor:**
  - Convert encoded data into tensors suitable for model training.
- **Model Architecture:**
  - Define a BiLSTM model with embedding layers, LSTM layers, and a linear layer.
- **Model Training:**
  - Train the model using the encoded tensors, optimizing the parameters over multiple epochs.

6. Guess the Next Letter

- **Prune Dictionary:**
    - Remove entries from the dictionary that are irrelevant based on the current word's length and state.
- **Predict Next Letter:**
    - Use the trained BiLSTM model to predict the next letter.
    - If there are enough potential solutions, guess the letter with the highest frequency.
    - If not, break the problem into subproblems and find solutions for the subproblems.

Example of the Vowel and Permutations Part

Permute All: Generate Masked Versions:

- For the word "apple", generate masked versions by replacing different letters:
    - ["*pple", "a_ple", "ap_le", "app_e", "appl*", "__ple", "_p_le", ...]
- Store these masked versions in a list.

Permute Consonants: Mask Consonants:

- For the word "apple", replace all consonants with underscores, resulting in "a__le".
- Generate all permutations by keeping the vowels in place and replacing different consonants:
    - ["a__le", "a__l_", "a___e", "__ple", "_p_le", ...]

Vowel Prior: Calculate Probabilities:

- Calculate the probability of each vowel (a, e, i, o, u) appearing in words of different lengths.
- Store these probabilities in a dictionary for use in guessing.