Hangman Game Strategy

Sanket Pramod Bhure

July 27, 2023

Hangman is a classic word guessing game. The player needs to guess a word, given a certain number of chances. In each turn, the player guesses a letter. If the letter is in the word, the positions of the letter are revealed. If the letter is not in the word, the player loses a chance. The game ends either when the word has been guessed (i.e., all letters are revealed) or when all chances are exhausted.

This document aims to describe a **Python**-based strategy for the Hangman game. The strategy used in this code is based on statistical **machine learning**, specifically in the use of **N-grams**. N-gram models are widely used in statistical natural language processing. This strategy uses a combination of n-gram statistics and frequency counts of letters to guess the most probable next letter. In my analysis of the dataset, we found that over 50% of the words have lengths of 7, 8, or 9 characters. Based on this observation, we decided to generate n-grams of a maximum size of 6 for our subsequent analyses.

The rationale behind this decision lies in the nature of n-grams. An n-gram of size 6 can effectively capture the structural nuances of words that are slightly larger (e.g., 7, 8, or 9 characters long). Thus, we believe that this choice will adequately represent the majority of the words in our dataset.

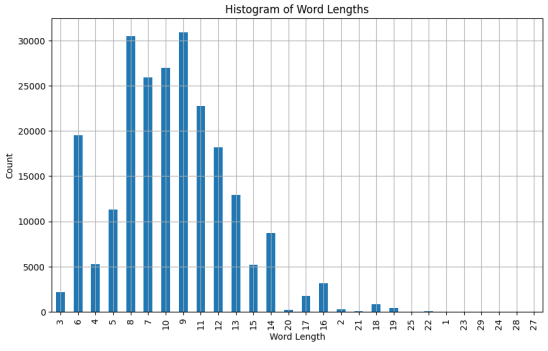

Figure 1: Histogram of word lengths

1 Code Breakdown

This code is designed to guess the word in the Hangman game. It first prepares a cleaned version of the word and then finds a set of potential words that could match the partially guessed word. Let's break down the code:

1. **Prepare Clean Word:** The given word is cleaned by removing spaces and replacing unknown letters (represented by underscores) with a period.
2. **Find Length of Word:** The length of the given word is calculated.
3. **Find Existed Letters:** The code finds the number of unique letters that have already been guessed correctly.
4. **Update Current Dictionary:** The code then constructs a new dictionary of possible words that match the known parts of the word.
5. **Determine N-gram Size:** The code then determines the size of the n-grams to use based on the length of the word.
   (a) The line $n = min(lenword-2, 6)$ is setting the value of n for n-grams. The min function is used to choose the smaller of two values. So, n is being set to the smaller of $lenword - 2$ or 6. There are two main reasons for this: To ensure that n is not greater than the length of the word being guessed minus 2. If the word being guessed is very short, it doesn't make sense to use large n-grams because they wouldn't fit within the word. Subtracting 2 provides a little extra buffer. To set an upper limit on the size of the n-grams. In this case, the upper limit is 6. Even if the word being guessed is very long, this code will not consider sequences of more than 6 letters. This might be to limit the computational complexity of the guessing algorithm, as larger n-grams require more processing to handle.
   (b) If the word being guessed is very short (3 letters or less), it may not make sense to use n-grams of the size calculated by the previous line of code ($n = min(lenword - 2, 6)$). For example, if the word is 3 letters long, $lenword - 2$ would be 1, resulting in 1-grams or individual letters. Setting n to $lenword - 1$ for words of 3 letters or fewer means that for a 3-letter word, **bigrams** (2-grams, or pairs of letters) will be used. For a 2-letter word, 1-grams (individual letters) will be used. And for a 1-letter word, n would be 0, although in a typical game of Hangman, it's unlikely to have a 1-letter word. This condition thus adjusts the n-gram length for very short words to make the guessing strategy more suitable.
6. **Calculate N-gram Frequencies:** This code generates a list of n-grams (contiguous sequences of n characters) from a dictionary of words. It then counts the frequency of each n-gram and sorts them in descending order of frequency.

(a) First, it iterates over every word in the dictionary, and for each word that is long enough, it generates all possible n-grams and stores them in a list.

(b) Next, it uses collections.Counter to count the frequency of each n-gram in the list.

(c) Finally, it sorts these frequencies in descending order using the most-common() method, resulting in a list of tuples, where each tuple contains an n-gram and its frequency.

7. **Select Most Matched N-gram:** The code selects the n-gram that matches the most with the current word. It generates all n-grams of length n from a given word clean word and appends them to the list word.

8. **Optimize Regular Expression:** The function creates a new dictionary of n-grams that match the selected n-gram from the word to be guessed. It is selecting the n-gram from the list that has the minimum number of dots ('.') but more than zero. It assigns this n-gram to the selected n-gram variable. The dots represent unknown characters. This n-gram could then be used in subsequent parts of the function to guide the guessing process.

9. **Find Regular Expression N-gram Dictionary:** The code finds the dictionary of n-grams that match the selected n-gram. The function then counts the occurrences of each character in the n-grams dictionary and sorts them.

10. **Guessing the Letter:** Finally, the function tries to guess the next letter based on several strategies:

    (a) It first checks the most common letters in the full dictionary and chooses one that hasn't been guessed yet.

    (b) If the first method fails, it then checks the most common letters in the matched n-grams and chooses one that hasn't been guessed yet.

    (c) If both methods fail, it defaults back to the most common letters in the full dictionary and chooses one that hasn't been guessed yet.

```
# Python code goes here
    # The function guess is defined with self and word as
    parameters. self refers to the instance of the class where this
     function is defined.
    # word is the word to be guessed, with unknown letters as
    underscores (e.g., "_ p p _ e").
    def guess(self, word):

      # The function then creates a "clean" version of the input
    word. It removes spaces and replaces underscores with dots (.),
      # which are used as a wildcard in regular expressions.
      clean_word = word[::2].replace("_", ".")
      len_word = len(clean_word)

```

```
    # The function counts the number of known letters in the word and stores them in the existed_letters list.
    Current_existed_letters = []
    existed_letters_length = 0
    for letter in clean_word:
        if letter != '.':
            existed_letters_length += 1
            if letter not in Current_existed_letters:
                Current_existed_letters.append(letter)

    # The function then updates the current_dictionary with words from the dictionary that match the current state of the word to be guessed.
    # If a word in the dictionary matches the regular expression created from the clean word, it's added to the new_dictionary.
    current_dictionary = self.current_dictionary
    new_dictionary = []
    for dict_word in current_dictionary:
        if len(dict_word) != len_word:
            continue
        if re.match(clean_word, dict_word):
            new_dictionary.append(dict_word)
    self.current_dictionary = new_dictionary




    # Next, the function decides the length n for the n-grams to be used based on the length of the word.
    # An n-gram is a contiguous sequence of n items from a given sample of text or speech.It then creates a list of n-grams from the full dictionary.
    n = min(len_word - 2, 6)
    if len_word <= 3:
        n = len_word - 1
    n_character_dictionary = []
    for word in self.full_dictionary:
        if len(word) > (n - 1):
            for i in range(0, len(word) - (n - 1)):
                n_character_dictionary.append(word[i:i + n])
    n_character = collections.Counter(n_character_dictionary)
    n_sorted_letter_count = n_character.most_common()
    word_n_gram_collection = []
    for i in range(0, len(clean_word) - (n - 1)):
        word_n_gram_collection.append(clean_word[i:i + n])


    # The function then creates a list of n-grams from the word to be guessed and
    # selects the n-gram with the least number of dots (i.e., unknown characters).
    selected_n_gram = ""
    periodcount = 100
    for grams in word_n_gram_collection:
        if grams.count('.') < periodcount and grams.count('.') > 0:
            selected_n_gram = grams
```

```
            periodcount = grams.count('.')


    # The function creates a new dictionary of n-grams that match the selected n-gram from the word to be guessed.
    # The function then counts the occurrences of each character in the n-grams dictionary and sorts them.
    n_gram_dictionary = []
    for dict_word in n_character_dictionary:
        if re.match(selected_n_gram, dict_word):
            n_gram_dictionary.append(dict_word)
    n_gram_string = "".join(n_gram_dictionary)
    c4 = collections.Counter(n_gram_string)
    matched_n_gram_sorted = c4.most_common()


    # Finally, the function tries to guess the next letter based on several strategies:
    # 1) It first checks the most common letters in the full dictionary and chooses one that hasn't been guessed yet.
    guess_letter = '!'
    for letter, instance_count in self.full_dictionary_common_letter_sorted:
        if letter not in self.guessed_letters and existed_letters_length < int(len_word / 2):
            guess_letter = letter
            break
    # 2) If the first method fails, it then checks the most common letters in the matched n-grams and chooses one that hasn't been guessed yet.
    if guess_letter == '!':
        for letter, instance_count in matched_n_gram_sorted:
            if letter not in self.guessed_letters:
                guess_letter = letter
                break
    # 3) If both methods fail, it defaults back to the most common letters in the full dictionary and chooses one that hasn't been guessed yet.
    if guess_letter == '!':
        sorted_letter_count = self.full_dictionary_common_letter_sorted
        for letter, instance_count in sorted_letter_count:
            if letter not in self.guessed_letters:
                guess_letter = letter
                break
    return guess_letter
```