- Project report
  - Introduction
  - The strategy (also see remark 0 below)
    - Step 1. Construct n_dictionary from training set
    - Step 2. Guess a letter for the game word
    - Remarks
  - The result
  - Discussion and future improvements

Project report

Introduction

This project implements a guess strategy for the hangman game. When provided with a word in the current state, e.g., "a _ _ l _ ", we need to guess a letter for any of the blanks. If our guess was successful, all blanks with that letter will be automatically filled. For example, if we guess letter "p" for the above word (suppose the underlying word is "apple"), then the letter is correct and the next state of the word becomes "a p p l _ ". If we guess the letter incorrectly for six times in total for a single word, we fail that word. The goal is to guess a letter as accurately as we can.

We were provided with a training set called `words_250000_train.txt`, and our strategy will be developed unpon it. The words in the game, however, are a distinct set from the training set. Therefore, we are supposed to find patterns in words in the training set and leverage them to guess the game word.

The strategy (also see remark 0 below)

Step 1. Construct n_dictionary from training set

Our approach is based on similarity between the game word and the words from our training set. Particularly, we first create a word truncation dictionary called n_dictionary, with key being the truncated word length and value is a python list that contains all

truncated words with that length from the training set. For example, if the current training word is "cat", then "ca","at" will be appended to key "2"'s list in the n_dictionary (since these truncations are of length 2), and "cat" will be appended to key "3"'s list in the n_dictionary. No more trunctaions from this word is possible. All words from the training set is processed in such a way to n_dictionary.

Step 2. Guess a letter for the game word

When it comes to a game word, e.g., a _ _ l _, we first look up the n_dictionary with a key equals to the word's length, in such a case, 5. We then iterate through all truncated words in n_dictionary[5], select word that matches the existing letters in a _ _ l _ (e.g., "arole" is a match, but "sober" isn't), and create a counter that aggregates the total number of occurence for different letters from those selected/matching words, with repetetive count of the same letter in a single truncated word being deactivated (that is, for example, if we want to count letters in "edeem", then "e" is counted only once for this word, but we allow multiple counts among different truncated words, i.e., if "reali" is another word we would like to count letters on, then "e" is counted again. The reason to do so is to maximize the occurence of a letter word-wise).

After the first lookup, we sort the counter from most frequently occurring letter to least frequently, and we choose the top-frequency letter THAT IS NOT GUESSED BEFORE to be our current guess.

If all letters from the counter has been guessed already, we start to truncate the game word with a step size of 2. That is, our next lookup for to the n_dictionary will be based on "a _ _ ", "_ _ l", "_ l _ " (step size is 2 means from the previous length 5 to the current truncation length 3, also see remark 1). We aggregate all counters from n_dictionary[3] from "a _ _ ", " _ _ l", and "_ l _", and return the highest-frequency letter THAT IS NOT GUESSED BEFORE as our current guess.

If all letters from the counter has been guessed already, we continue to subtract the length by 2 and use that length to truncate the game word, lookup the n_dictionary based on those truncations, add up counters, and return the highest-frequency letter THAT IS NOT GUESSED BEFORE as our current guess.

If the minimum truncation length 3 is reached (see remark 2), we go back to the default strategy. That is, return the most occurred letter (can count repeatedly for the same letter in a single word) in the training set as our next guess (see remark 3).

Remarks

0. Our strategy is highly intuitive and systematic, as the patterns in the words are exactly syllables, which are identified using our matching criteria (see above description). Beyond that, one word may be contained as a sub-word in another word, which makes word truncation reasonable.
1. The reason for choosing a decrement step size as 2 is, we don't want it to be too small, e.g., a step size 1, because too similar truncated lengths will likely result in similar counters, which is a waste of computing time. We don't want it to be too large as well, otherwise the minimum truncation length of 3 will be quickly reached and the default strategy will be used which doesn't utilize any game word information.
2. The reason for setting the minimum truncation length to 3 is, the minimum-size syllable shall contain 3 letters (an experienced guess), any size below that doesn't provide much useful patterns but is rather a probability game.
3. The default strategy is reasonable, since if no patterns can be found in the game word, we would rather guess a mostly occuring letter in the training set, which is a probability game that represents/resembles the whole dictionary.

The result

After playing 1,000 recorded games, our strategy achieved an accuracy of 57.7%. That is, 577 game words are successfully guessed out of 1,000 game words.

Discussion and future improvements

During game play, we observed that the lengthier the game word, the more accurate the strategy. The reason might be that lengthier words provide more information, or more sub-patterns to be discovered/utilized, which can greatly improve the guessing accuracy.

Future improvements can focus on hyperparameter optimization/tuning. The step size we used as decrement in the game word truncation is a crucial hyperparameter. In addition, the minimum truncated word length is also a hyperparameter that worth future research.