Hangman Report

Priya IIT Kanpur

24 March 2024

INTRODUCTION:

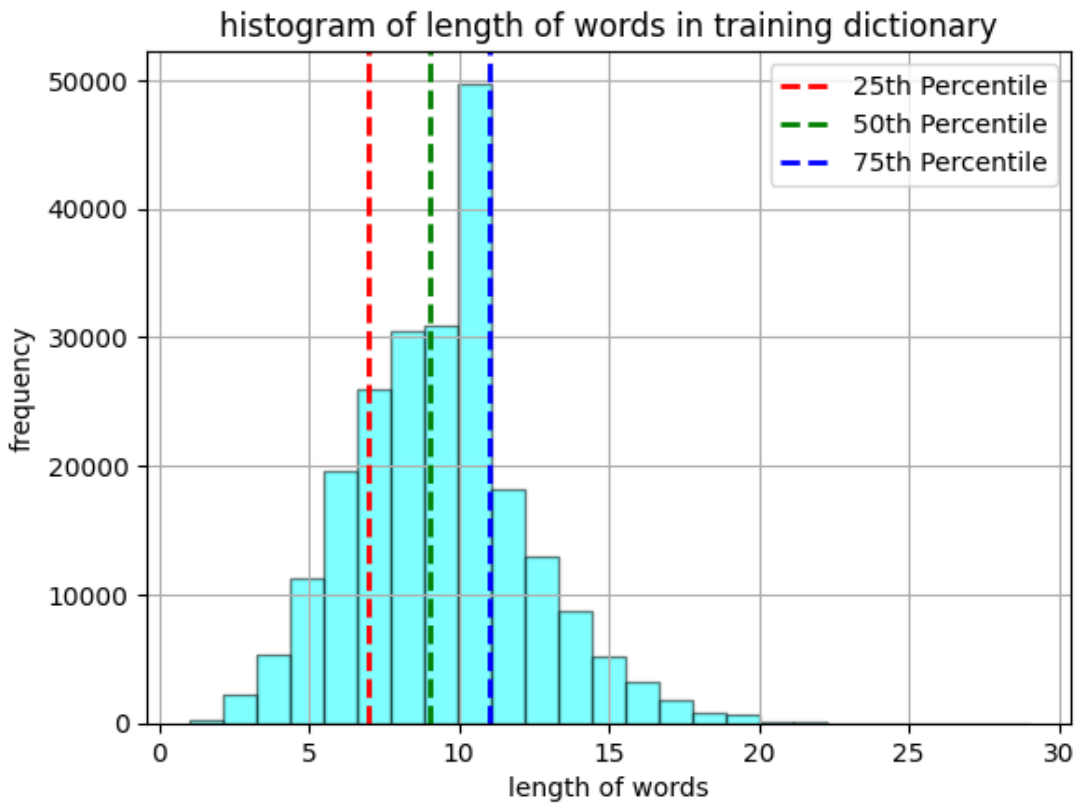
Hangman is a classic game where you guess a hidden word, represented by blanks, one letter at a time, aiming to avoid six wrong guesses. We were required to write a "guess" function that takes the current word (with underscores) as input and returns a guess letter. The main aim was to create a better algorithm(an original ML approach, non n-gram based) with more than 50% accuracy.

In this report, I explain how I made my Hangman-playing program. I looked for patterns in English words and statistics to make a smart model. The idea was to have a better chance of guessing the right letters and improving the success rate.

Approach:

As in the already given sample of code, we were trying to get the frequency of individual letters, and then we were trying to guess the letter with the highest frequency. But this approach was not effective, So intuitively, I considered the frequency of words two at a time, three at a time, four at a time, five at a time, and six at a time. We can consider frequencies taking more letters at a time, but that won't impact the accuracy of the model much and will also take more time and space for the model to run.

The provided training corpus shows that about 75% of the words have a length greater than 7. Assuming our test set has a similar distribution to the training data is a standard practice in machine learning. The graph for the same is attached below. Therefore, I have considered up to six letters for frequency calculation.



Method:

In my model, I have considered the frequency of occurrence of two, three, four, five and six letters together from the given data set and then found their probability, later on while guessing letter, each time my model broke the word to be guessed into substrings of length ranging from 1 to 6.

Later, it considers substrings with only one unknown letter, it replaces that blank space with each letter in the English alphabet and then from our data provided it records its probability.

In the model I have made, I have calculated the probability of each alphabet considering substrings of length one, two, three, four, five, and six depending upon the length of word to be guessed and then sum up this probability (by multiplying by suitable weights) to get the cumulative probability for each English alphabet. Then, Using the argmax() function, I considered the alphabet with maximum probability.

To optimize the model, I have considered a weight array whose values are multiplied by that of probabilities of each alphabet when substrings of length one, two, three, four, five, and six are considered. The value of this array is determined by parameter tuning. Which I have figured out by running practice games for around 1800 hangman games.

This basic approach is based on the frequency of occurrence and probability and patterns observed from the training data set.

In the model function of my class, I have considered the frequency of occurring any alphabet once, together, in a pair of three, four, five, and six. I have employed training data for this

purpose and stored these frequencies into different matrices for future use. Cases of repeated alphabet in a word has been taken care by the seen_letters list, which keeps track of the alphabet in a word encountered.

We divide our word to be guessed into substrings and then act upon substrings with only one blank, as this way of approach gives a clear idea of how the algorithm is working, avoids any source of confusion and is helpful in debugging.

As mentioned above, for that one blank space, we consider each alphabet and find out its frequency from already calculated frequency data and then calculate a probability for that letter to fill that space. The probability of already guessed letters is assigned zero for efficient guessing.

This approach is suitable for words of any length as it computes required probabilities only, based on the length of the word to be guessed. So this method can be considered as a general approach.

Analysis:

As I am considering the combination of alphabets with larger lengths (like 4/ 5/ 6), their probability is decreasing compared to the combination of alphabets with smaller length (like, 2/ 3), therefore comparatively larger weights needed to be assigned to combinations of higher order.

Result:

I employed the strategy mentioned above and achieved an accuracy of **62.5%** surpassing the target accuracy of **50%.**