Approach to the Hangman Problem

This document contains the approach that I used to solve the hangman problem. Here, I have also mentioned the reasons and strategies that lead me to this solution.

First, I played the hangman game several times to understand how a real person approaches the problem because if we don't know which approach is appropriate, we can not find a good solution. While playing the game, I found that choosing vowels in starting is a good strategy, as in almost all words, at least one vowel always occurs.

The approach that was given to us (checking the highest frequency of characters from the words from the dictionary that match the current word) was not generalising anything; in one way, we can say that it was overfitting on the training data. Whenever we play hangman, we try to predict the next character based on the currently guessed characters that we have with us, and we try to spell it and if it looks good/correct, then we try out that character so in one way, we are not actually guessing the whole word in the starting when we have guessed fewer characters instead we just try to predict the neighbouring characters of the guessed ones. And when we have enough characters with us from which we can predict the whole word we just go for it, but this last step can not be used here as our training and testing data are totally disjoint. So I tried just the first approach, which can be implemented with n-gram models.

I implemented the n-gram model and tested it, but it gave poor accuracy; the accuracy was around 30% (after optimising parameters/weights). Then I tried to use **MLP-based neural networks with embeddings**, but it also gave poor accuracy of ~20% when I was using a context length of 5 (i.e. length of the substring that I was giving as training data), and one thing that I noticed is that to train the MLP-based model, we first need to take all possible combinations of every word like we have to consider if, in a given word, a character is predicted or not and if we go by that way we will end up with a very large dataset, because one word itself will be making nearly 2^n training data, where n is the length of the word, so this was not feasible, but one way to reduce these was to just keep some random number of characters in the word which are most frequent as we can assume that our model will be predicting those character first, so with this approach I trained the MLP-based model on nearly ~50000 words, but still it was not performing well.

So I observed what might be the problem and then realised that I was not using incorrect letters, i.e. letters which have been guessed but were found wrong, in the training or in the prediction, and this can be useful to predict other characters in the word because as we discussed earlier, all words are made up with some relations between their characters so I tried **LSTM model**, in

which I was giving the model an extra 26 length of input where these inputs were containing the information about whether the word of that index is guessed or not and by this I was training the LSTM model, but the accuracy remained almost same (~22%), the main problem in this approach was that the model was not able to predict characters efficiently if it is between two predicted characters, i.e. if some part of the word is “i_g” where “ _” represents the character which we want to predict, here model should guess n because I have observed that in most of the cases while this type of formation is happening n is coming but the model was predicting n after 2-3 trails which was very bad, so clearly it can not perform well.

After trying the above approach, I realised that the model was doing exactly the same thing which n-gram was doing because after all, for one input, it had many possible answers, and it was considering one of them which has the highest probability of occurrence in that situation, and also using n-gram was also beneficial as it can compute the exact probability in seconds while these approaches were taking a long time to train and also training it was very hard. So I again switched back to the n-gram approach, and this time I was updating the current dictionary, which contained the words in which no incorrect letters (i.e. letters which were guessed previously but found wrong) were coming. This was predicting the words with very good accuracy, and after optimising the weights/parameters, it was also guessing some words without any penalties and further optimising the parameters/weights, I reached the current solution.

Implementation details:-

The approach is a very simple n-gram-based approach. I am building n-gram models up to n=5 (i.e., building all 1-gram, 2-gram, 3-gram, 4-gram, and 5-gram models). Here, when training the model (finding the frequency), I am training the 1-gram, 2-gram, and 3-gram models based on the length of the words because in some cases, that also depends on the length, which I found from the link. Additionally, I am using 4-gram and 5-gram models. While using the model, we generally find the probability of the next character given the first n-1 characters, but in this case, if we know letters that are not consecutive, e.g., "i_g", we can also predict the middle character. Therefore, I am exploring all possibilities that the model can be used for. For example, with a 5-gram model, we can use it to predict the character from the word in one of the following forms: ****_, ***_*, **_**, *_***, _****, where * represents the known character in the word and _ represents the place of the character we want to predict. I am using all models in all cases. So, whatever character is most probable, I am returning that element. If any character is found incorrect, then I am updating my current dictionary and also updating the n-gram models. In case the current dictionary becomes empty, which might occur as a consequence of removing all words containing incorrect characters, I am returning the elements based on the highest probable sequence given by the article.