Analysis of Problem Statement -

The challenge involves guessing a word based solely on its length. As we successfully guess letters within the word, we gain more information to aid in our next letter guess. Each time, we are limited to guessing only one letter and the outcome can either be right or wrong. In case of an incorrect guess, we proceed by guessing the second most probable letter. Conversely, if the guess is correct, we use the new information to make our next educated guess.

Let me take on the hangman challenge manually. When given a placeholder with a length of 'n,' the only clue we can utilize for the first letter guess is its length. Generally, we opt for the most common letters like 'e' or 'i' based on the word length 'n'. We then begin our guessing process. As we successfully guess one letter and determine its position in the word, we apply our algorithm to guess the next potential letter. This pattern continues as we gradually uncover more letters from the word, leaving only a few empty spaces in the placeholder. At this point, we rely on the existing information generated by the other letters to make our guesses. The faster we predict the first correct word, the faster our algorithm will solve the problem because then it had to look for fewer words in the dictionary.

This challenge involves working with both the practice API and the final API. The practice API has a usage limit of 100k, so I have decided to split my task into three parts:

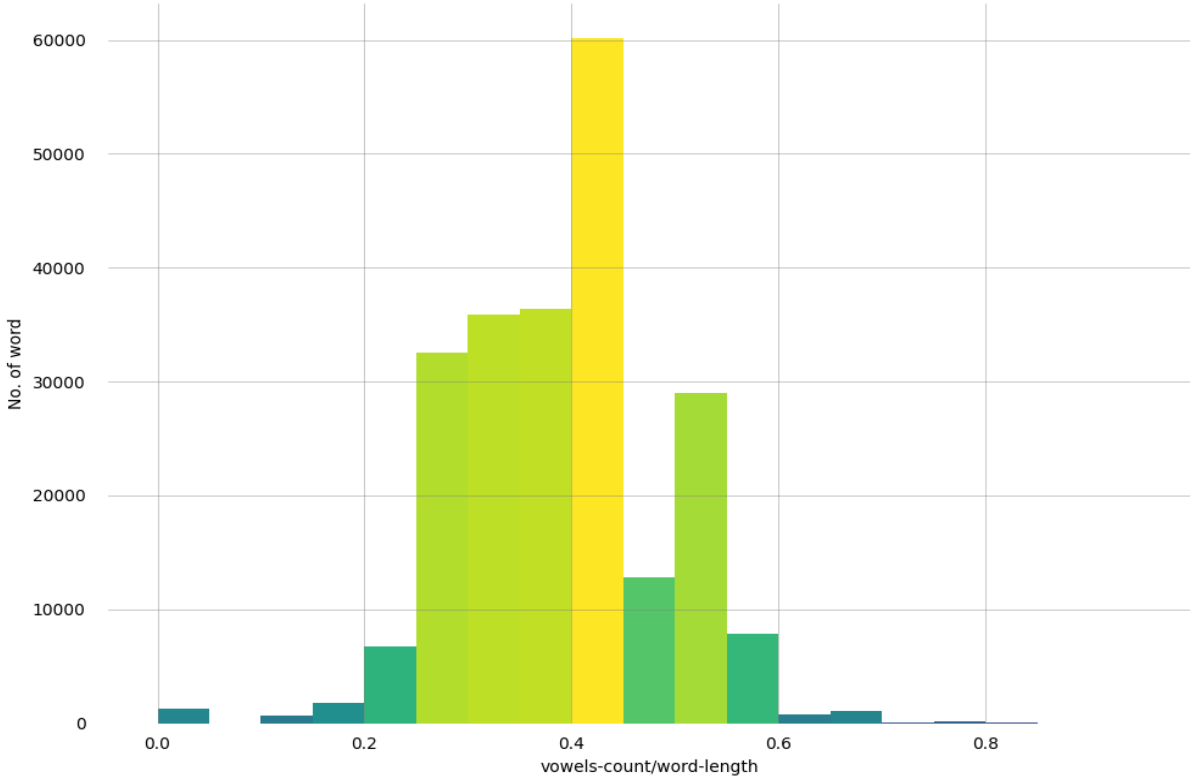
1. Deciding which Hangman Strategy to use
2. Build and Train the Hangman model.
3. Continuously refine and fine-tune the algorithm until the deadline, maximizing its performance.

I tried a bunch of other Ideas like using the Probability of occurrence of words for which I got an accuracy of 19%, then tried using Conditional Probability and it’s accuracy increased to 21% then I tried using the most frequent characters as starting guess but these ideas produced a model with an accuracy less than 50%, So after some trials, I found the best combination for which I got an accuracy of approximately 60% which I have mentioned in the Algorithm Used Section. There was one more approach of using reinforcement learning but due to time constraint I could not tried out that approach much.

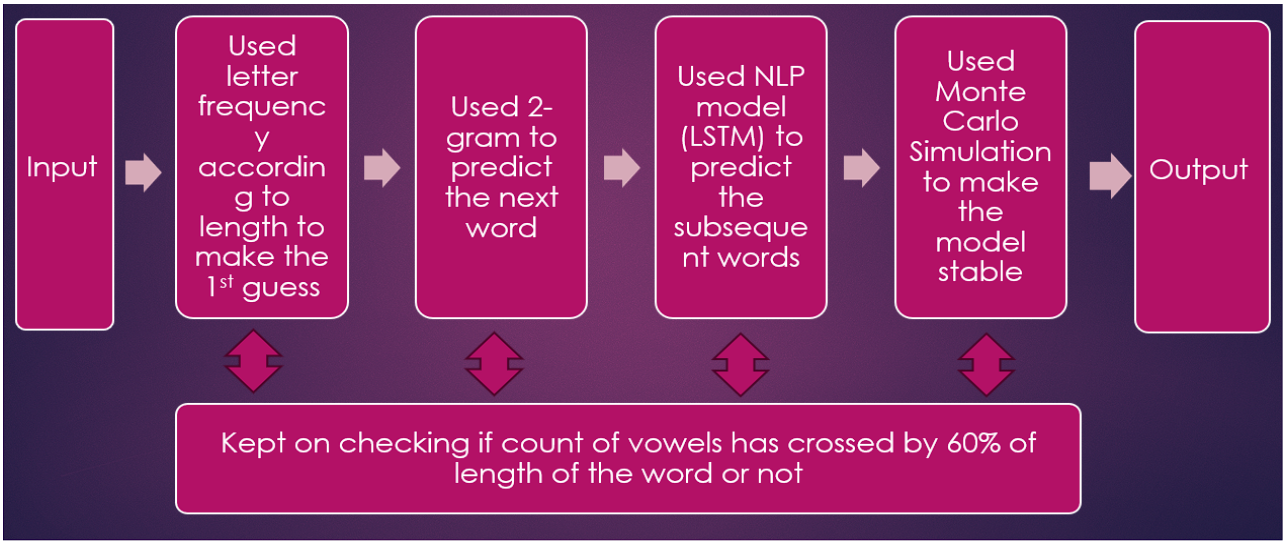
Algorithm Used -

1. Guess the first letters. In this step, the algorithm will make the guess based on the **letter frequency** in the corresponding word group. Word group here refers to all the words of the same length in the training set. For example, if the train set is ["ab", "cde", "adf"] then we have two-word groups: length == 2 group and length == 3 group. In length == 2 group, "a" has the highest frequency, then our first guess will be "a". If the letter with the highest frequency is not in the hidden word, the algorithm will continue to guess the letter with the second-highest frequency. This step will keep going on until the algorithm has made a valid guess.

2. Guess other letters by **2-gram**. After dividing all words into different groups, I calculated the frequency of the 2 grams in each group. Starting from the first letter guessed in the first step, the algorithm will calculate all the 2 grams containing the letter and choose the 2-gram with the highest frequency. This step will keep going on until the algorithm has made a valid guess.
3. There is also a chance that no matched 2-gram will be found. In this case, I pre-trained an **LSTM model** to fill in the remaining blanks. The main benefit of the model is to introduce uncertainty so that we can make predictions other than combinations we have seen in the training set. I have mentioned the details of my LSTM model in the LSTM heading .
4. Then I used **Monte Carlo simulation** to sample multiple predictions from the LSTM model and then selected the most likely guess based on the probabilities of letters predicted by the LSTM model in the sampled guesses. I did this so that the prediction from my LSTM becomes stable and it has also reduced its vulnerability to occasional incorrect predictions.
5. I also did some **Exploratory Data Analysis** and find out that if 60% of the length of a word is a vowel, it is unlikely (from the distribution) that there will be more vowels so my algorithm will skip vowels guess if already the guess has reached more than 60 % of word length. The graph below is just a proof of my statement.

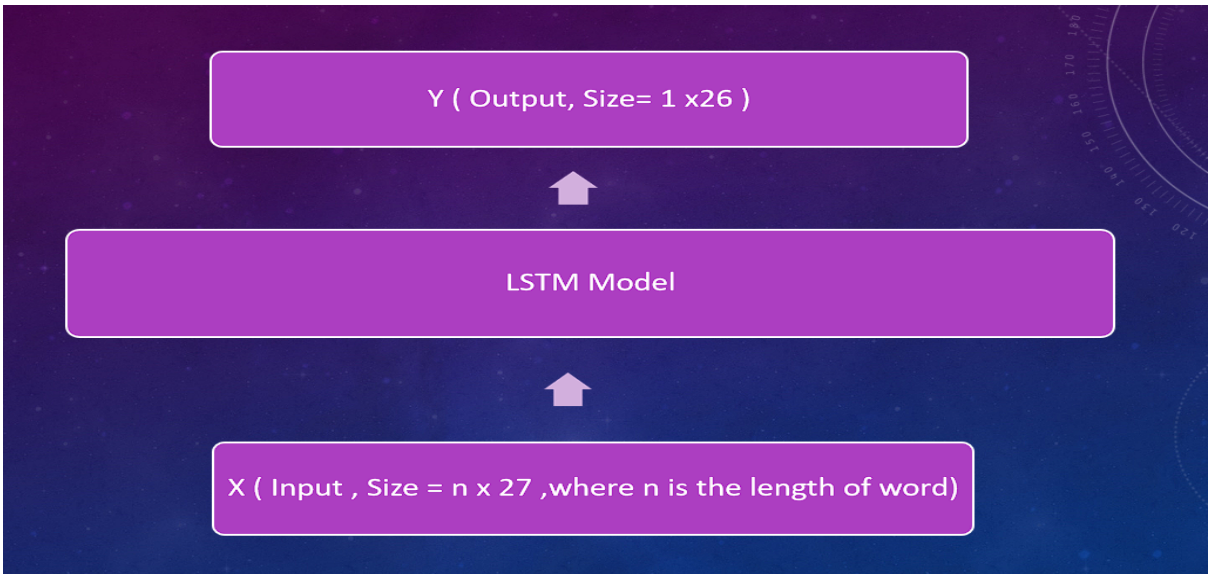


Flow Chart for the Algorithm -



LSTM Model -

The LSTM model takes as input the history of guessed letters, the unguessed word (with known letters represented as underscores), and the previously guessed letter's one-hot encoded representation. It then predicts the most likely next letter.
The structure of the LSTM model is as follows: Input: encoded Undiscovered word(X of size n*27) where 'n' represents the length of the word, the shape of X is 27 because, in addition to the 26 letters, we also need to account for a space represented by '_'. and the output is a 1 by 26 vector with each element between 0-1, indicating the probability of the next letters



The network receives as input a representation of the word (total number of characters, the identity of any revealed letters as well as a list of which letters have been guessed so far. It returns a guess for the letter that should be picked next.
For the hyperparameter I found out on fine tuning that I was getting best accuracy on for no. of epochs =6 and learning rate =0.001 . I have used Adams Optimizer as the optimizer and BCE with logit loss as the loss function .

Training set generation -

I trained the model using the simulated games generated from the training set (implemented in Word2Batch() class.
The training set was created following these procedures:

1. Randomly selected a word and replace all its characters with blanks.
2. Generated the encoded label for this modified word.
3. Allow the model to make predictions at each step and generate the complete training set.

Pros of My Model -

1. It produces stable results even on testing for small datasets.
2. It is very less vulnerable to occasional incorrect predictions.
3. It has a good Success Rate of approximately 60 %.

Challenges Faced & What I did to overcome them -

Limited Time - As we were given only 7 days to complete this challenge I tried my best to try as many as things as possible but my algorithms take almost 11 min for running 100 games so each time I was making any change and tested it, that took a lot of time .
To overcome this I made a detailed planning in the beginning So that I am not wasting time on doing some random things .

References -

1. http://datagenetics.com/blog/april12012/index.html
2. https://azure.microsoft.com/en-us/blog/