General Strategy

To approach the Hangman problem, the model I used was a Transformer that was implemented with reference to *The Annotated Transformer* from harvardnlp. However, the training techniques and problem solving approach were completely original.

Overview

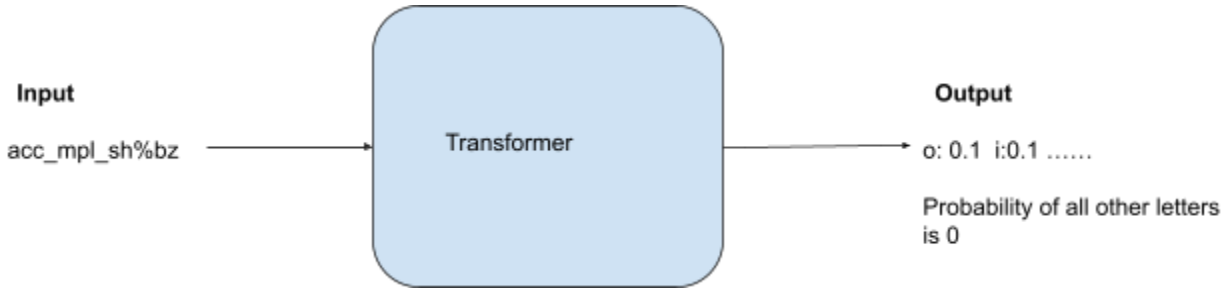

Fig 1: Model Overview

Input

For each of the 250 000 words in the dataset, I randomly chose which letters to mask inside of a word. Letters after the “%” sign (Fig 1) were also randomly generated to represent letters guessed which were not part of the word. The intention was that the Transformer would be able to map patterns within the masked word and guesses to a probability distribution for letters behind the blanks. Including incorrectly guessed letters in the label slightly improved win ratio (Fig 2)

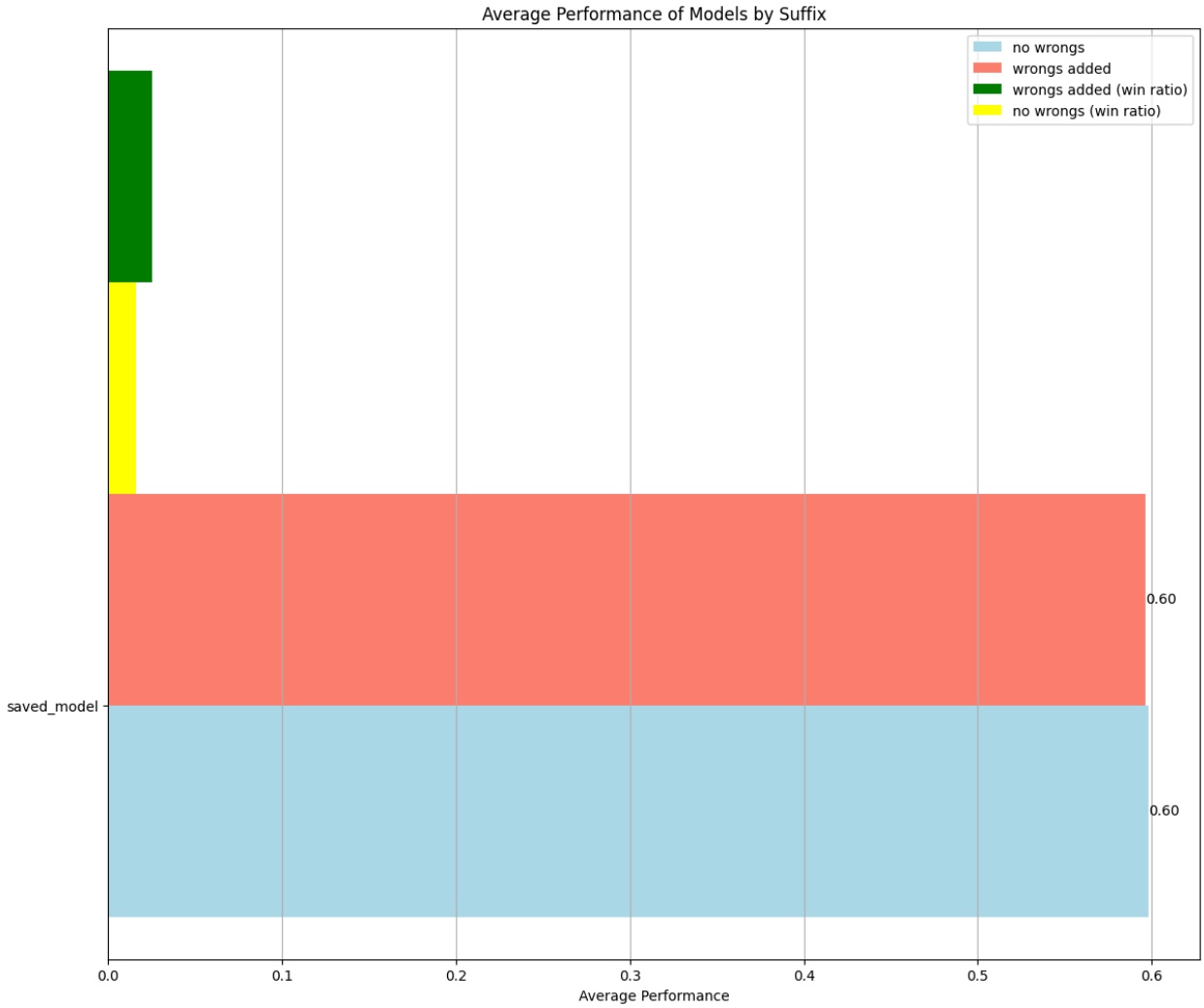

Fig 2: Performance of model with unsuccessful letters in input (wrongs) vs inputs without them.

Output

$ is the padding character

Items	[0 , 0 , 0 , 0 , 0, 0, 0,...0.1,...,0.1, ...0]
Probability of	$, _ , a , b , c , d , e ... i, ... o, ... z
index	0 1 2 3 4 5 6 10 28

Fig 3: Output displaying probabilities of all letters given a masked word.

The output is a list where each index represents a certain character within our vocabulary. For example: a = 2, b = 3, c = 4 and so on (Fig 3). The value at each index represents the probability of the letter inside of the word **if and only if** the letter is still masked. Letters which

are unmasked will have their probability set to 0. The reasoning behind this is that I did not want the machine learning model to give preference to words which have already been picked. probabilities are scaled proportionally to the number of blanks in a word. The more blanks that are in a word, the lower the probabilities will be.

Training

In order to save time I trained the Transformer on GPUs using Google Colab. Later, I saved the model so I would not have to retrain it again. This is why in my notebook you will see that rather than training the model, I am calling the code below inside of my guess function.

```
model = torch.load(saved_model, map_location=torch.device('cpu'))
```

However, all the code needed for training is still within the notebook. It is just commented out.

Prediction

As shown in the diagram in the first page, for each given state in the game, the input will be

Masked version of word + % unsuccessful guessed letters

For example “appl_%d” means that currently the word is “appl_” and the only unsuccessful guess so far is “d”.

After predicting with the Transformer and doing some processing on the output, the algorithm gets a list of characters from most to least likelihood of being inside of the word. The algorithm will pick the most likely letter unless it has already been guessed. In that case it takes the next most likely.

Why Transformer?

The reasoning for using a Transformer was due to its ability to process the entire letter sequence simultaneously rather than sequentially. This allows it to capture more complex patterns which cannot be observed by simply seeing what letters are before and after each other. This is especially useful in Hangman where many letters are masked.

Sources

https://nlp.seas.harvard.edu/2018/04/03/attention.html