Hangman Challenge Strategy

Initially, when given an input, we first look through all words in the training dataset that matches the same length as the input. Then, we create an initial guess (at this point we do not have any information, so we need to make our first move) by calculating all the character-appearance frequency within the training dataset, and returning the largest one (if this is incorrect, we record the guesses so that we do not make the same mistake, and then select the second frequent appearance). Subsequently, when we receive the corrected answer from the server, we create a constraint from the result, identifying that a certain position should be of a certain character. Utilizing this constraint set, we go through all the words given by the training dataset of equivalent length, and filter the next most-frequent character by applying the constraint to the search space.

If at a certain point we cannot find any letters present in the dataset that we consider as potential matches, we need to produce outputs that are not following certain combinations we have seen before. My initial approach was to encode the word via one-hot encoding for all the letters that I observed from the response of the server, and then use ensemble method from AutoGluon (https://auto.gluon.ai/) to generate classification predictions (a method I utilized during my internship at Amazon). However, I found that since different masking of words (i.e., the different positions where characters are marked as '_') will result in different embeddings passed to the ensemble, a huge amount of training dataset is required. I had neither the time and effort to experiment with up-sampling methods, nor did I have enough trials to submit and obtain the test cases from the server. Therefore, I did some research, and utilized the following method: I followed the tutorial from Azure (https://github.com/Azure/Hangman), using the CNTK library (which I had exposure to during my internship at Autodesk, working on knowledge graphs). The problem can be formulated as a classification task, optimized using cross-entropy loss, and outputting the ASCII code of the 26 English characters with maximum probability. I utilized the given framework to train an LSTM model on simulated games played using the given training dataset, and finetuned the hyperparameters using Optuna (https://optuna.org/), and utilized the practice APIs to evaluate the result after each tuning trial. Eventually, the model returns the best-suited prediction produced from the LSTM model. The best model is saved using CNTK, and loaded back into the inference framework within the notebook, where I stripped off the simulation layers of CNTK and integrated the word received from the server as input to the model.

In summary, the first statistics-based method works well in identifying words that appear (or appear to be similar) to the training dataset, whereas the LSTM model from CNTK library provides probabilistic predictions that are not restricted to the appearances in the training dataset.