Hangman Algorithm

Harikesh Kushwaha

May 21, 2023

1 Problem Statement

The problem statement, as copied from the given notebook reads:

For this coding test, your mission is to write an algorithm that plays the game of Hangman through our API server.

When a user plays Hangman, the server first selects a secret word at random from a list. The server then returns a row of underscores (space separated)—one for each letter in the secret word—and asks the user to guess a letter. If the user guesses a letter that is in the word, the word is redisplayed with all instances of that letter shown in the correct positions, along with any letters correctly guessed on previous turns. If the letter does not appear in the word, the user is charged with an incorrect guess. The user keeps guessing letters until either:

1. the user has correctly guessed all the letters in the word
2. the user has made six incorrect guesses.

You are required to write a "guess" function that takes current word (with underscores) as input and returns a guess letter. You will use the API codes below to play 1,000 Hangman games. You have the opportunity to practice before you want to start recording your game results.

Your algorithm is permitted to use a training set of approximately 250,000 dictionary words. Your algorithm will be tested on an entirely disjoint set of 250,000 dictionary words. Please note that this means the words that you will ultimately be tested on do NOT appear in the dictionary that you are given. You are not permitted to use any dictionary other than the training dictionary we provided. This requirement will be strictly enforced by code review.

You are provided with a basic, working algorithm. This algorithm will match the provided masked string (e.g. `a _ _ l e`) to all possible words in the dictionary, tabulate the frequency of letters appearing in these possible words, and then guess the letter with the highest frequency of appearance that has not already been guessed. If there are no remaining words that match then it will default back to the character frequency distribution of the entire dictionary.

This benchmark strategy is successful approximately 18% of the time. Your task is to design an algorithm that significantly outperforms this benchmark.

2 Solution

2.1 A Note on the Benchmark Strategy

The benchmark strategy, which uses the frequency of letters in the dictionary to guess the next letter, is a very simple strategy, and it works well, given that the word we want to guess is in the dictionary. In fact, this strategy works almost 93% of time if you are using the same dictionary. I tried it on 80% words of the training dataset, and it gave an accuracy of about 93%. However, on the other 20% of the dataset, it failed miserably, having an accuracy of just over 14%. Since out goal is the play with a dictionary which our model will never see, we have to do better than this. We need some algorithm which does not need the frequency of the letters in the dictionary to guess the next letter. We need an algorithm which can guess the next letter based on the current word, and not the dictionary. This is where the idea of *ngrams* and *Markov Chains* comes in.

2.2 The Algorithm

2.2.1 Ngrams and Markov Chains

Ngrams are a sequence of n letters. For example, the word `apple` has the following bigrams:

```
ap
pp
pl
le
```

The idea of ngrams is to guess what the next letter will be, given the previous letter(s). So, basically, we are talking about conditional probability. For example, we might want to know the probability of the letter `l` given that the previous letter was `p`. This can be written as $P(l|p)$. This is the probability of the letter `l` given that the previous letter was `p`. Now, to guess a correct word, ideally, we need the conditional probability of each next word given all the previous words. For example, the probability of guessing the letter `apple` is given by:

$$P(a) \times P(p|a) \times P(p|ap) \times P(l|app) \times P(e|appl) \tag{1}$$

Of course, this is not feasible to have conditional probabilities of all the words. So, we need to make some assumptions. The first assumption is that the probability of the next letter depends only on the previous n letters. This is called the *Markov Assumption*. So, if we assume that the probability of the next letter depends only on the previous letter, then the above equation becomes:

$$P(a) \times P(p|a) \times P(p|p) \times P(l|p) \times P(e|l) \tag{2}$$

Which is much easier to calculate.

2.2.2 Calculating the Conditional Probabilities

The idea behind calculating the conditional probabilities is to count the number of times a letter appears after a particular sequence of letters. For example, if we want to calculate the probability

of the letter `l` given that the previous letter was `p`, we count the number of times the letter `l` appears after the letter `p`, and divide it by the total number of times the letter `p` appears. One can easily implement this using a dictionary in Python. However, for my case, I have used the NLP library NLTK. The reason behind this is to avoid unnecessary code, and to make the code more readable. Plus, the library provides an easy to use API for working with ngrams and accessing the conditional probabilities.

In NLTK, to create an ngram model, you start by creating ngrams from the words in the dictionary. Then, you create a conditional frequency distribution from the ngrams. This can be done using the MLE class from nltk.lm module. Here are the steps to create an ngram model:

```
from nltk.lm import MLE
from nltk.util import everygrams, flatten

n = 5
train_dataset = (list(everygrams(word, max_len=n))
 for word in train_data)
vocabulary = flatten([list(w) for w in train_data])
model = MLE(n)
model.fit(train_dataset, vocabulary)
```

This is everything we need to create a model which has 1, 2, 3, 4 and 5 grams. Next, we can call the score method of the MLE object to get the conditional probability. For example, to get the conditional probability of the letter `l` given that the previous letter was `p`, we can do the following:

```
model.score('l', ['p'])
```

Similarly, to get the conditional probability of the letter `l` given that the previous two letters were `ap`, we can do the following:

```
model.score('l', ['a', 'p'])
```

as easy as it can get!

2.2.3 Utilizing the ngram Model

Okay, so now we have a model which can give us the conditional probabilities of the next letter given the previous n letters. But how do we use it to guess the next letter? Well, we can use the model to guess the next letter by calculating the conditional probabilities of all the letters in the alphabet, and then choosing the letter with the highest probability. Only in our case, instead of choosing letters from the whole alphabet, we choose letters from the remaining letters which have not yet been guessed. To understand this, let's take an example.

Suppose we have the word `a _ _ l _`. Our goal is to use, say bigrams, to guess the next letter. We see that there are three possible places where bigrams can be used in the above letter, namely, `a _`, `_ l` and `l _`, where _ represents the blank space. Since we have already guessed the letters a and l, we will replace the underscore with letters other than a and l and calculate the score for each combination using the model defined above. Note that this way, we will get three

probabilities for each letter. We can combine them using some method (say mean, max or just sum). I tried all the three methods, and found that the mean method works best. So, we will use the mean to combine the probabilities. The letter with the highest probability will be chosen as the next letter. This process will be repeated until we have guessed all the letters in the word, or we have reached the maximum number of guesses.

The same algorithm can be used for trigrams, four-grams and five-grams. The only difference is that the number of possible places where the ngrams can be used will be different. For example, in the case of trigrams, there will be three possible places where the trigrams can be used, namely, `a p _`, `a _ p` and `_ p p`.

2.2.4 Combining the ngram Models

So, we have unigram to five-grams. But which one to use? Well, we can use all of them. The idea is to use the ngram models in the order of their accuracy. For example, we can use the five-gram model first, then the four-gram model, then the trigram model, and so on. This way, we will be able to use the most accurate model first, and then use the less accurate models to fill in the blanks. This will help us to guess the word more accurately. Another thing which I have used is to give these models different weights. These weights are hyperparameters, and can be tuned to get better results. I tried to use the given dictionary to tune these hyperparameters, by creating a training and test set from the dictionary. However, I found that there is not a good correlation between the performance of the model on training set and the actual game with the server, neither is there a good correlation between the performance of the model on the test set and the actual game with the server. So, I decided to use the weights which I found to be working best for me. The weights which I used are as follows:

```
weights = {
    w1: 0.1,
    w2: 0.2,
    w3: 0.2,
    w4: 0.25,
    w5: 0.2
}
```

2.2.5 Starting the Game

There is a problem with the ngram model. It cannot be used to guess the first letter of the word. This is because the ngram model requires the previous n letters to calculate the conditional probability of the next letter. So, we need to use some other method to guess the first letter. For this, I have used the frequency approach, as given in the baseline model. It works quite well. In fact, another hyperparameter I have used is the percentage of initial guesses made using the frequency approach before the helm is given to the ngram models. I used the value of 0.5, which means that the first 50% of the guesses will be made using the frequency approach, and the remaining 50% will be made using the ngram models.

This works because even though the actual dataset is not the same as the training dataset, it has almost the same frequency distribution up to a few letters. And hence using the frequency approach to guess the first few letters works quite well. Another small modification to the initial

frequency approach is that even if this method is not able to guess half the letters, it will still give the guessing rights to the ngram models if the number of attempts remaining is less than 3. This is done to remedy the situation where the word length is very small and the letters of the word to rare meaning that the frequency approach is not able to guess even half the letters. In such a case, the ngram models perform better.

2.2.6 The Final Algorithm

Here is an overview of the final algorithm:

1. Create a dictionary of words from the given dictionary file.
2. Create a unigram, bigram, trigram, four-gram and five-gram model from the dictionary.
3. If the number of attempts remaining is greater than 2, and number of guessed letters is less than half the length of the word, then guess the next letter using the frequency approach.
4. If not, then guess the next letter using the ngram models.
5. For ngram models, use the weights defined above.

Please have look at the code for more details.

2.3 Results

The model performs about 40% on the training dictionary. On the server, it performs about 30%.

2.4 Improvements

A number of improvements can be made to the model. Some of them are as follows:

1. The weights can further be tuned.
2. Some other NLP models can be used to improve the performance of the model.