Trexquant Interview Project (The Hangman Game)

Gaurish Bansal

Introduction

The game of Hangman, a popular word-guessing game, has caught the attention of many due to its language and computer challenges. As part of the internship selection process at TrexQuant, we were tasked with creating an algorithm that performs better than a basic model in playing Hangman. The goal was to design an algorithm that can achieve an accuracy rate of over 50%.

This report provides a detailed look at how I developed my Hangman-playing algorithm, including the methods used and the results obtained. My strategy focused on finding patterns in English words from the given dictionary, using conditional probabilities, and applying statistical techniques to create an advanced model capable of guessing letters more accurately.

Intuition

Hangman is a game where one person thinks of a word, and another person tries to guess it by suggesting letters. The word is represented by dashes, and the guessing player tries to fill in these dashes by suggesting letters. The game ends either when the guessing player correctly guesses the entire word or when they make **6 incorrect guesses**.

To guess the word more effectively, players can use strategies based on how often letters appear in English words and common letter combinations. For example, certain letters often appear together, like **'Q' followed by 'U'**, and many words **end with 'TION','MENT','NESS'**. We made the assumption that letters in a word are connected to their neighboring letters, and we used this idea to analyze patterns in the English words from the provided dictionary.

One crucial assumption is that the length of the word matters. If the **word is very short**, like 5 or 6 letters, it becomes **harder to guess correctly** because there are fewer letters to consider, and patterns are less noticeable. Fortunately, short words are not very common.

Additionally, we observed that some letter combinations occur at the end of words, such as "MENT". To take advantage of this, **we added dummy letters "{" and "|"** at the beginning and end of each word in the dictionary. This **helps us identify patterns [because some combinations appear only in the end - eg: ing,tion]** accordingly.

We use the **N-grams approach** here. N-grams are contiguous sequences of n items (usually words) from a given text. For instance, in the sentence "The cat is sitting on the mat," the 2-grams (bigrams) are "The cat," "cat is," "is sitting," etc.

Considering these factors, I developed an algorithm to predict the most likely letter in the word during the Hangman game.

Method

Methodology for Hangman Problem:

1. **Initialization:**
   - Start with the given word, such as APPLE, and represent it with blanks: _ _ _ _ _.
   - Add starting and ending dummy variables: { _ _ _ _ _ |.
2. **Guessing Letters:**
   - Lets say we have guessed letters (e.g., P, E, D, I) with the help of our strategy.
   - So we will get the word with the guessed letters: { _ P P _ E |.
3. **Pattern Breakdown:**
   - Break down the current word pattern into 1-grams, 2-grams, 3-grams, 4-grams, and 5-grams components [more than that would have led to over-fitting, and the model performance was also not increasing significantly after 4 grams so stopped at 5; and also it would have required more pre-computation in more n-grams]. I have just considered the components with 1 or 2 blanks, no more than that [this would again have led to more complexity, and also with 2 blanks i got good accuracy].s

Table 1: Pattern components of the string ("{ _ P P _ E |") with one space.

Length 1 ($\propto_1$)	**Length 2 ($\propto_2$)**	**Length 3 ($\propto_3$)**	**Length 4 ($\propto_4$)**	**Length 5 ($\propto_5$)**
_ ($\propto_{11}$)	{ _ ($\propto_{21}$)	{ _ P ($\propto_{31}$)	{ _ P P ($\propto_{41}$)	P P _ E | ($\propto_{51}$)
_ ($\propto_{12}$)	_ P ($\propto_{22}$)	_ P P ($\propto_{32}$)	P P _ E ($\propto_{42}$)	
	P _ ($\propto_{23}$)	P P _ ($\propto_{33}$)	P _ E | ($\propto_{43}$)	
	_ E ($\propto_{24}$)	P _ E ($\propto_{34}$)		
		_ E | ($\propto_{35}$)		

Table 2: Pattern components of the string ("{ _ P P _ E |") with two spaces.

Length 4 (β_1)	**Length 5 (β_2)**
_ P P _ (β_{11})	{ _ P P _ (β_{21})
	_ P P _ E (β_{22})

4. **Calculate Weights:**
    - For each component, calculate the conditional probabilities of observing each letter.
    - **Assign weights** to each letter based on its probability in the context of the specific component.
    - **Calculate a score for each guessed letter** in the current pattern by summing the weights across all relevant components.

      Let the letter for which i am calculating the score is : K

      Score(K)=

$$\sum_{i=1}^{5} ©_i \sum_{j=1}^{n_i} Probability(\gamma_{ij} = K \mid \gamma_{ij} \in \boldsymbol{\alpha}_{ij}, \boldsymbol{\alpha}_{ij} \notin \{P,E,D,I\}) \ +$$
$$\sum_{i=1}^{5} ©_i \sum_{j=1}^{n_i} [\ Probability(\gamma_{ij1} = K \mid \gamma_{ij} \in \beta_{ij}, \beta_{ij} \notin \{P,E,D,I\})$$
$$+ Probability(\gamma_{ij2} = K \mid \gamma_{ij} \in \beta_{ij}, \beta_{ij} \notin \{P,E,D,I\})$$
$$- \ Probability(\gamma_{ij1} = K \ and \ \gamma_{ij2} = K \mid \gamma_{ij} \in \beta_{ij}, \beta_{ij} \notin \{P,E,D,I\}) \ ]$$

where :

$©_i$ is a parameter which denotes the influence of each pattern on the overall score. It only depends on the length of the pattern.

$\boldsymbol{\alpha}_i$ denotes the set of patterns of length i in the table 1.

γ_{ij} denotes the letter in the blank space of the pattern $\boldsymbol{\alpha}_{ij}$ (containing one blank space).

γ_{ij1} denotes the letter in the first blank space of the pattern β_{ij} (containing two blank spaces).

γ_{ij2} denotes the letter in the second blank space of the pattern β_{ij} (containing two blank spaces).

$$P(\gamma_{ij} = K \mid \gamma_{ij} \in \ P\_E, \ \gamma \notin \{P,E,D,I\}) = \frac{N(PKE) \cdot \mathbb{I}(K \notin \{P,E,D,I\})}{\sum_{\delta \in \{A,B,\dots Z\}} N(P\delta E) \cdot \mathbb{I}(\delta \notin \{P,E,D,I\})}$$

$$P(\gamma_{ij1} = K \mid \gamma_{ij} \in \ \_PP\_, \ \gamma \notin \{P,E,D,I\}) = \frac{\sum_{\delta \in \{A,B,\dots Z\}} N(KPP\delta) \cdot \mathbb{I}(K \notin \{P,E,D,I\})}{\sum_{\delta \in \{A,B,\dots Z\}} N(P\delta E) \cdot \mathbb{I}(\delta \notin \{P,E,D,I\})}$$

$$P(\gamma_{ij2} = K \mid \gamma_{ij} \in \ \_PP\_, \ \gamma \notin \{P,E,D,I\}) = \frac{\sum_{\delta \in \{A,B,\dots Z\}} N(\delta PPK) \cdot \mathbb{I}(K \notin \{P,E,D,I\})}{\sum_{\delta \in \{A,B,\dots Z\}} N(P\delta E) \cdot \mathbb{I}(\delta \notin \{P,E,D,I\})}$$

$$P(\gamma_{ij1} = K \ and \ \gamma_{ij2} = K \mid \gamma_{ij} \in \ \_PP\_, \ \gamma \notin \{P,E,D,I\}) = \frac{\sum_{\delta \in \{A,B,\dots Z\}} N(KPPK) \cdot \mathbb{I}(K \notin \{P,E,D,I\})}{\sum_{\delta \in \{A,B,\dots Z\}} N(P\delta E) \cdot \mathbb{I}(\delta \notin \{P,E,D,I\})}$$

5. **Letter Selection:**
    - Choose the letter with the highest score as the next guess. This letter is likely to maximize the chances of completing the word correctly.
6. **Update Pattern:**
    - Incorporate the guessed letter into the current word pattern.
    - Repeat the process by going back to step 3 until the word is completely guessed or 6 incorrect number of guesses are reached.
7. **Algorithm Iteration:**
    - Iterate through steps 3 to 7 for each new guessed letter based on the evolving word pattern.
8. **Termination:**
    - Stop the algorithm when the word is successfully guessed or when the maximum number of allowed incorrect guesses is reached.

This methodology leverages n-gram components to capture the contextual information in the evolving word pattern. By assigning weights based on conditional probabilities, the algorithm intelligently selects the most promising letters to maximize the accuracy of the guesses in the Hangman game.

I have precomputed the frequencies of each pattern in our dictionary using the `build_freq_tables()` function. This process is executed only once, enabling us to access the frequency of a specific pattern, denoted as N(x), in constant time.

Function	Description
guess(word)	This is the primary function responsible for combining probabilities and determining the most likely letter to be returned.
find_score(word,ngram)	Breaks the word into patterns of length ‘l’ and returns the sum of probabilities for each letter in the alphabet within those patterns.
Conditional_Prob1(pat)	This function takes a pattern ‘p’ as input, which contains exactly one blank space, and calculates the probability mass function of the missing letter given the other letters in the pattern ‘p’.
Conditional_Prob2(pat)	This function takes a pattern ‘p’ as input, which contains exactly two blank spaces, and calculates the probability mass function of the missing letter given the other letters in the pattern ‘p’.
normalize1(pat)	This function takes as input the raw frequencies of each letter and returns the corresponding probabilities. (in case of 1 blank space)
normalize2(pat)	This function takes as input the raw frequencies of each letter and returns the corresponding probabilities. (in case of 2 blank spaces)
build_freq_tables()	This function is executed only once, calculating the frequencies of every pattern in our dictionary. This enables us to calculate N(x) in constant time.

Conclusion

In summary, this statistical algorithm for the Hangman game represents a substantial improvement compared to the basic model. By utilizing conditional probabilities, linguistic patterns, and statistical analysis, I attained an accuracy rate of 66.5%, exceeding the targeted 50%.