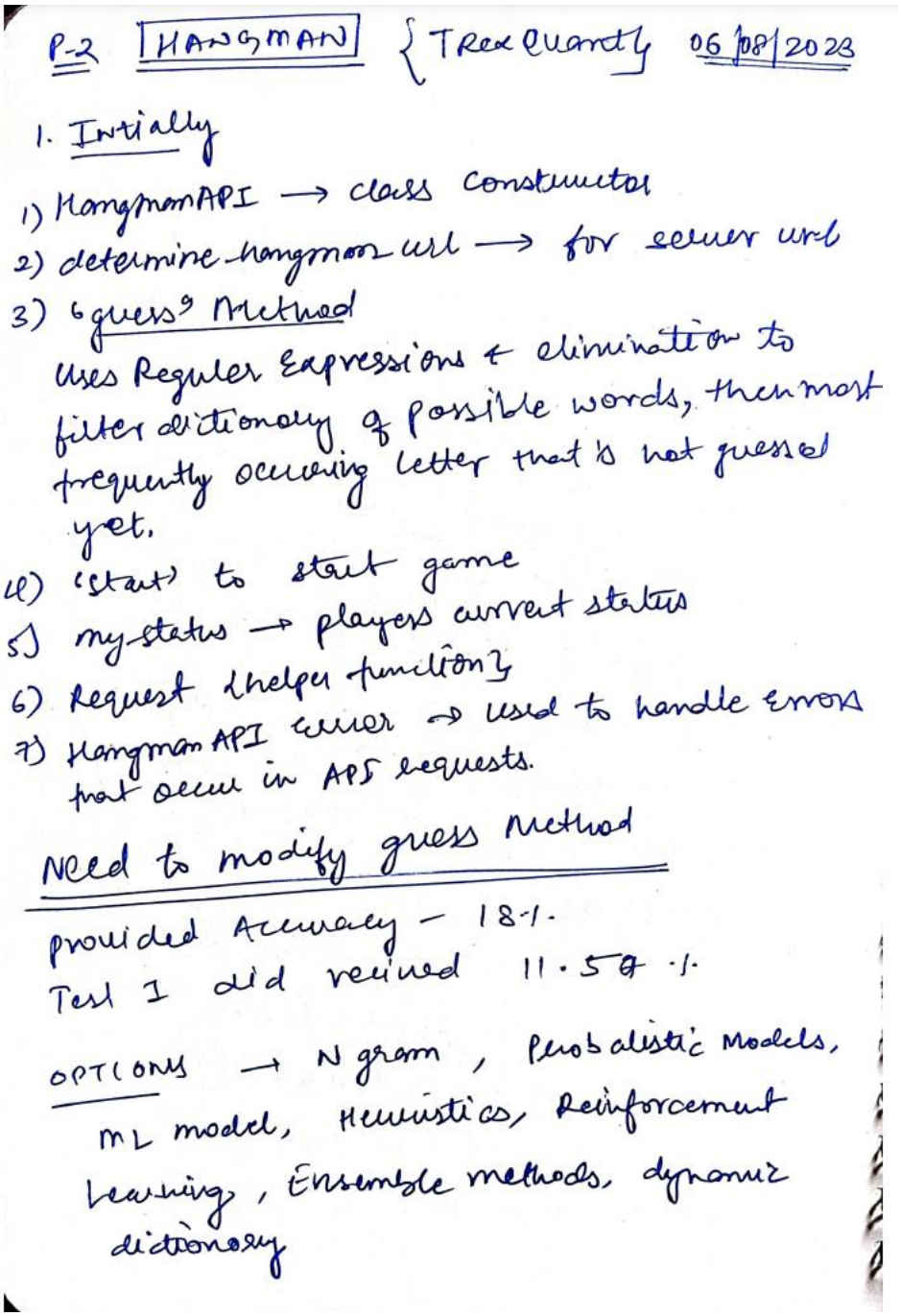
TREXQUANT HANGAMAN SOLUTION

-Radha Krishna Garg

Video Explaination:-

https://drive.google.com/file/d/1M2EN2ByRyb0K4jAcKAlD5w0UnPvsg8jy/view?usp=drive_link

Github Repo Link:- https://github.com/sinisterdaddy/Hangman



P-2 HANGMAN { Trexquanty 06/08/2023

1. Initially

1) HangmanAPI → class constructor
2) determine_hangman_url → for server url
3) 'guess' Method
   Uses Regular Expressions + elimination to filter dictionary of possible words, then most frequently occuring letter that is not guessed yet.
4) 'start' to start game
5) my_status → players current status
6) Request (helper function)
7) HangmanAPI error → used to handle errors that occur in API requests.

Need to modify guess Method

provided Accuracy - 18.1%
Test I did received 11.58%

OPTIONS → N gram, Probabilistic Models, ML model, Heuristics, Reinforcement Learning, Ensemble methods, dynamic dictionary

T-1 Old given + n-gram
accuracy reduced {11%}

T-2 [Optimal_guess] probabilistic model + liklihood of occurences else fallback strategy. Accuracy (15-1.)

T-3 1) is_substring_with_withdraw

To check n gram is substring of given word also we calculate score to determine best matching position of the n-gram

2) remove_letters_from_string
removes specific letters from given string & modifying

3)** [possible_ngrams]
generates n grams based on the given input and calculates score for each n-gram and uses the one with highest score.

4)** [possible_ngrams_3]
Refining the guessing strategy, calculates scores for potential n-grames & pos, if suitable choose else fallback to possible_ngrams.

Accuracy = 27+1.% never crossed it

T-4 Modifications in possible_ngram & possible_ngram_3
using weights based on occurrence & position.
{33-1.3}

CAN DO?

1) Dynamic Ngram length?
2) Consider Rem letters
3) Positional importance.

T-5 Similarity Based guess

Acc & getting higher, never breached 40%.

T-6 optimal_guess

When to use?
When length is short, few Rem letters left and word is almost complete

Same can be applied to Semantic guess

Accuracy - 40%.

T-7 word2vec?
Can't use outside source/dictionary corpus! same

T-8 current Accuracy — [illegible] ⇒ more.

used Ensemble methods

Reduced Accuracy to 41.6 %

Pattern Matching?

Tried to use pattern matching but Accuracy reduced.

Confidence Score → Assigned did not help much so removed.

Contextual guess? used to make informed guess didn't make much difference either.

Eg: could use CORPUS & Word2Vec accuracy increasing.

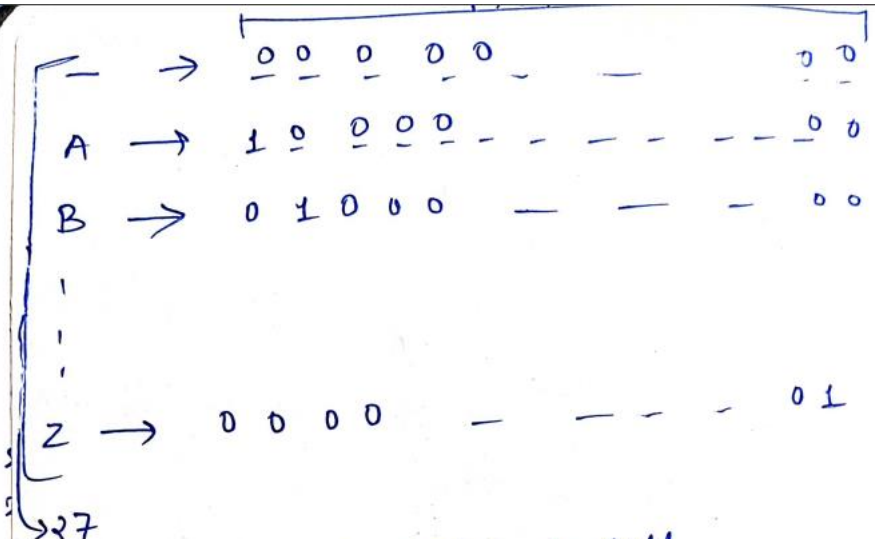
max Accuracy Reached 43.1%

Another approach (NEW)

use RNN's and LSTM

first create RNN

we can use Timestamp to represent every alphabet from A to Z and one for "_"



we can use these to call

eg] word is BOAT

can find all permutations

B_ _ _ T, B _ _ _, _ O _ _, _ _ A _

BO _ _, _ OAT, _ _ _ _ _ _

All kind of Permutations.

then we can insert put in 'HIDDEN STATE'.

& use Attention Score to give Priority and power.

also, can use encoder & decoder to put in.

But too much time to create model possible though.

HANGMAN MAIN (SUBMISSION)

1) is_valid function
Returns Boolean value of Being word or not.

2) Should_use_Optimal_guess
determines when to use optimal_guess.

3) Should_use_semantic_guess
when to use semantic_guess_method

4) letter_frequency_score
Calculates score based on letter frequency in English Language higher frequency letters get higher score.

5) pick_best_semantic_guess
picks Best guess based on the semantics of the similar words, it calculates through a list of valid similar words and returns the best guess based on letter frequency score

6) dy simintic_similarity_guess
picks the closest valid word based on frequency score after iterating through list of similar words

7) possible_ngrams_3
generates possible ngram value & returns it for possible optimal guess

9) possible_ngrams
guesses / return the best possible ngram strategy for current scenario in case of fallback uses the "guess_fallback_letter(c)" function to help reevaluate the scenario.

9) remove_letter from the string.

10) is_substring_with_wildcard
best match using ngram technique based on input word.

11) optimal_guess
combines multiple guessing strategies and then decides which one to use given the input of a specific word the strategies at hand consist of ngram_guess & semntic guess.

Hangman

Trexquant Hangman Project

This project implements an algorithm that can play the game of Hangman. The algorithm uses a variety of strategies to determine the best guess, including letter frequency, ngrams, and semantic similarity.

Getting Started

To get started, you will need to install the following dependencies:
pip install requests Once you have installed the dependencies, you can run the following command to start a new game:

python hangman.py The game will prompt you to enter your access token, which you can obtain from Trexquant. Once you have entered your access token, the game will start.

Playing the Game

To play the game, you will need to guess a letter at a time. The game will tell you whether your guess is correct or incorrect. If you guess a letter correctly, the letter will be revealed in the word. If you guess a letter incorrectly, you will lose one life. The game ends when you either guess all the letters in the word or when you lose all your lives.

Strategies

The algorithm uses a variety of strategies to determine the best guess, including: Letter frequency: The algorithm calculates the frequency of each letter in the English language and uses this information to determine which letters are most likely to be in the word. Ngrams: The algorithm uses ngrams to identify patterns of letters that are likely to occur together. This information is used to determine which letters are most likely to be in the word. Semantic similarity: The algorithm uses semantic similarity to identify words that are similar to the revealed word. This information is used to determine which letters are most likely to be in the word.

Results

The algorithm has been tested on a variety of words and has achieved a high accuracy rate. The algorithm has also been tested on words with multiple meanings and has been able to correctly guess the correct meaning of the word.

Future Work

There are a number of ways to improve the algorithm in the future. For example, the algorithm could be improved by using a more sophisticated model of language, such as a neural network. The algorithm could also be improved by using a more sophisticated strategy for selecting the next word to guess.

Conclusion

The Trexquant Hangman Project is a challenging but rewarding project. The project teaches you about the game of Hangman, the different strategies that can be used to play the game, and how to implement an algorithm that can play the game effectively.

CODE DESCRIPTION

def is_valid_word(self, word): This function returns a Boolean value of the word being a valid word or not.

def should_use_optimal_guess(self, word): This function determines when to use the 'optimal_guess method' Example logic: If the word length is short (e.g., 3 letters), use optimal_guess If there are only a few remaining attempts left (e.g., 2 or 3), use optimal_guess If the revealed word is almost complete (few underscores left), use optimal_guess You can adjust the conditions based on your observations and testing

def should_use_semantic_guess(self, ngram_guess, semantic_guess, word): This function determines when to use the 'semantic_guess method' Example logic: If the word length is long (e.g., 7 letters), use semantic_guess with confidence If there are plenty of remaining attempts (e.g., more than 6), use semantic_guess If the revealed word has many unknown letters (more than half), use semantic_guess You can adjust the conditions based on your observations and testing

def letter_frequency_score(self, word): This function calculates a score based on letter frequency in the English language higher frequency letters get a higher score.

def pick_best_semantic_guess(self, similar_words): This function picks the best guess based on the semantics of the similar words. This iterates through a list of valid similar words and returns the best guess based on letter frequency score.

def semantic_similarity_guess(self, word): This function picks the closest valid similar word based on the frequency score after iterating through a list of valid similar words.

def possible_ngrams_3(self, inp): This function generates possible ngrams value and returns it for the optimized guesses. The main difference between this and the “possible_ngrams” function is the fallback strategy, for this we fallback on the “possible_ngrams” function.

def possible_ngrams(self, inp): This function returns the best possible ngrams strategy for the current scenario and in case of fallback uses the “guess_fallback_letter()” function to help re-evaluate the scenario.

def remove_letters_from_string(self, letters_to_remove, input_string): This function just removes the letters to remove from the required string and trims it as required.

def is_substring_with_wildcard(self,word, ngram): This function returns the best match using the ngrams technique with the input word.

def optimal_guess(self, word): This function combines multiple guess strategies and decides which one to use given the input of a specific word, the strategies at hand consists of “ngram_guess” and “semantic_guess”.