# Automating Wordle: A Simple Idea Turned to Frequency-Based Solving

Tom Vija

May 23, 2025

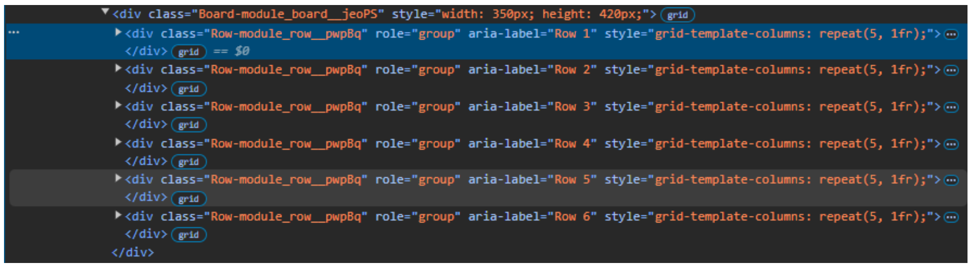
**Abstract**

This report details the creation of a Python-based Wordle solver. Initially conceived out of a personal vendetta with the popular word game, this project evolved from a web-scraping and random-choice mechanism to a more sophisticated solver leveraging word frequency data. The journey involved exploring web browser automation with Selenium, HTML parsing, data processing, and algorithmic refinement. The final program demonstrates significantly improved performance, a way to test the efficiency of different words, and a framework for testing different solving strategies.

## 1 Introduction: Why am I doing this?

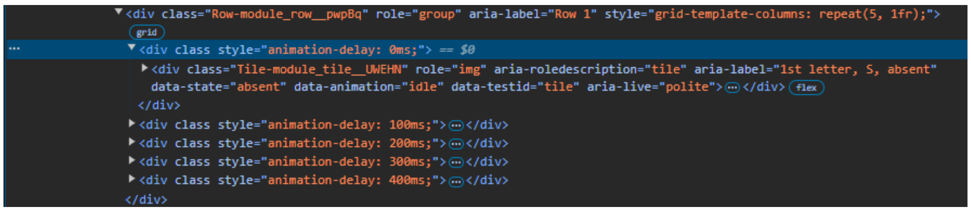
When Wordle first captured global attention, I, like many, was initially a casual observer. However, the game eventually drew me in. One day, I found myself staring at my screen for about 10 minutes, trying to come up with a word that would fit the constraints from my previous guesses. That is when I thought to myself, 'Why am I doing this? I could let Python do the work for me.' This was the moment I decided to create `Wordle_doer.py`.

My first step was to understand how Wordle worked under the hood. Looking into the "inspect element" tool in my web browser revealed a surprisingly straightforward HTML structure for the game grid and keyboard. I observed that each row was contained within a div element, and within each row's div, there were five more div elements (or tiles), one for each letter. After inputting my first two guesses, I noticed the data-state attribute on these tiles, which had one of three values: 'present', 'absent', or 'correct'. Having learned the website's structure, I was ready for my first iteration of the Wordle solver.



```
▼<div class="Board-module_board__jeoPS" style="width: 350px; height: 420px;"> grid
...  ▶<div class="Row-module_row__pwpBq" role="group" aria-label="Row 1" style="grid-template-columns: repeat(5, 1fr);"> ⋯
      </div> grid == $0
     ▶<div class="Row-module_row__pwpBq" role="group" aria-label="Row 2" style="grid-template-columns: repeat(5, 1fr);"> ⋯
      </div> grid
     ▶<div class="Row-module_row__pwpBq" role="group" aria-label="Row 3" style="grid-template-columns: repeat(5, 1fr);"> ⋯
      </div> grid
     ▶<div class="Row-module_row__pwpBq" role="group" aria-label="Row 4" style="grid-template-columns: repeat(5, 1fr);"> ⋯
      </div> grid
     ▶<div class="Row-module_row__pwpBq" role="group" aria-label="Row 5" style="grid-template-columns: repeat(5, 1fr);"> ⋯
      </div> grid
     ▶<div class="Row-module_row__pwpBq" role="group" aria-label="Row 6" style="grid-template-columns: repeat(5, 1fr);"> ⋯
      </div> grid
    </div>
```

Figure 1: Structure for each row on the Wordle website



```
▼<div class="Row-module_row__pwpBq" role="group" aria-label="Row 1" style="grid-template-columns: repeat(5, 1fr);">
 grid
  ▼<div class style="animation-delay: 0ms;"> == $0
    ▶<div class="Tile-module_tile__UWEHN" role="img" aria-roledescription="tile" aria-label="1st letter, S, absent"
     data-state="absent" data-animation="idle" data-testid="tile" aria-live="polite">⋯</div> flex
   </div>
  ▶<div class style="animation-delay: 100ms;">⋯</div>
  ▶<div class style="animation-delay: 200ms;">⋯</div>
  ▶<div class style="animation-delay: 300ms;">⋯</div>
  ▶<div class style="animation-delay: 400ms;">⋯</div>
 </div>
```

Figure 2: Structure for each letter (take note of the 'data-state' attribute for the letter).

## 2 Methodology: Building the Solver

### 2.1 Phase 1: The Initial Approach - Random Guessing

The initial goal was to create a bot that could interact with the Wordle website, make guesses, and use the feedback to narrow down possibilities until the correct word was found.

#### 2.1.1 Data Collection and Web Interaction

I began by sourcing a list of 14,855 five-letter words permissible in Wordle (`wordles.txt`). To interact with the game, I chose Selenium, a browser automation tool, to handle navigation on the Wordle website. Selenium allowed the script to navigate to the Wordle website, click buttons (like "Play" and the on-screen keyboard), and input words, all without me lifting a finger. I used a Chrome driver to instantiate and control a Chrome browser instance for the automation tasks.

The script uses Selenium to find and click the letter buttons on Wordle's virtual keyboard:

**Interacting with Wordle's keyboard using Selenium (`Wordle_doer.py`)**

```python
#inputs the value passed into the function.
def convert_keys(word, driver):
    for char in word:
        time.sleep(.1)
        driver.find_element(By.CSS_SELECTOR, f'button[data-key="{char}"]').click()
    driver.find_element(By.CSS_SELECTOR, 'button[aria-label="enter"]').click()
```

After each guess, the script read the color-coded feedback —gray, yellow, or green— for each letter. This information was compiled into a 'hint list,' using 'g' for green, 'y' for yellow, and 'r' for gray (signifying an absent letter). This was critical to be able to correctly filter out incorrect words.

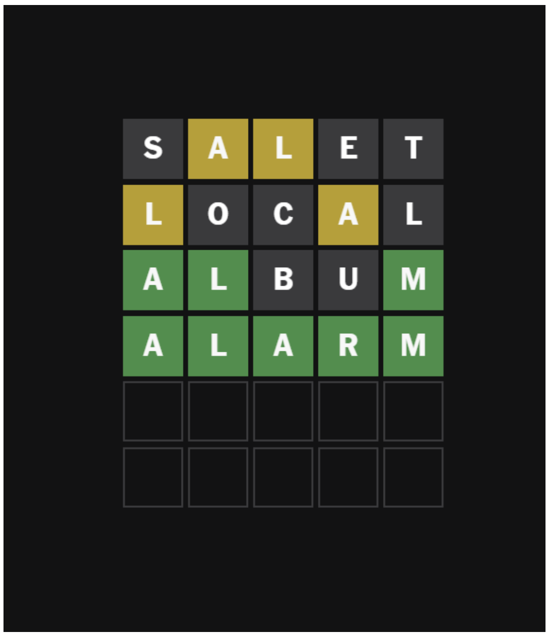
**Extracting guess feedback using Selenium (`Wordle_doer.py`)**

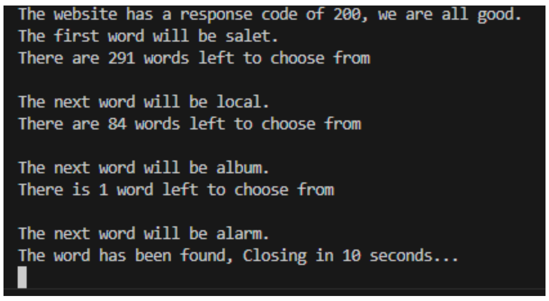
```python
def update_keys(driver, guess_num):
    hint = [' '] * 5
    # Find all rows
    rows = driver.find_elements(By.CLASS_NAME, 'Row-module_row__pwpBq')
    row = rows[guess_num]
    # Find all tiles in the row
    tiles = row.find_elements(By.CSS_SELECTOR, 'div[data-testid="tile"]')

    #fill in the hint list
    for i in range(5):
        aria_state = tiles[i].get_attribute('data-state')
        if aria_state == 'correct':
            hint[i] = 'g'
        elif aria_state == 'present':
            hint[i] = 'y'
        elif aria_state == 'absent':
            hint[i] = 'r'
    return hint
```

### 2.1.2 Filtering Algorithm and Random Selection

With the feedback obtained, the core logic involved filtering the master word list. An algorithm was developed to eliminate words inconsistent with the hints received. For example, if a letter was marked 'green', all remaining candidate words had to have that letter in the same position. If 'yellow', the letter had to be present but not in that position. If 'gray' (and not present elsewhere in the guess), the letter was eliminated entirely from consideration for other words.

After filtering, the bot would pick a random word from the remaining valid options for its next guess. At this point I had achieved my inital goal, I sat back and watched as my program would effortlessly solve the wordle puzzle. While this worked, my natural curiosity got the better of me and I told myself: ”I can make this better”.





```
The website has a response code of 200, we are all good.
The first word will be salet.
There are 291 words left to choose from

The next word will be local.
There are 84 words left to choose from

The next word will be album.
There is 1 word left to choose from

The next word will be alarm.
The word has been found, Closing in 10 seconds...
```

Figure 3: Program guesses with corresponding terminal output. (Remaining word counts added for clarity)

## 2.2 Phase 2: Incorporating Word Frequencies

What if instead of choosing randomly, we selected words more likely to be the answer? But what defines a 'more likely' word in this context? My intuition was that more commonly used words would be better candidates. After all, if Wordle frequently chose obscure words like 'pucer' (yes, it's a valid guess), the game would likely be less fun, and fewer people would play. Therefore, the next goal became finding a dataset that ranked words by their popularity.

### 2.2.1 Data Preparation with `freq_organizer.py`

I found a suitable word frequency dataset (`frequency.csv`), that scraped over 300,000 words from the internet and counted their frequency. However, this list needed to be reconciled with Wordle's specific vocabulary. I wrote a Python script, `freq_organizer.py`, using the Pandas library to:

1. The frequency dataset was filtered to include only words present in the Wordle-approved word list (`wordles.txt`).
2. For words existing in the Wordle list but absent from the frequency dataset, a frequency of 0 was assigned (as these were presumed to be highly obscure).
3. This curated list, now complete with frequency data, was saved as `cleaned_frequency.csv`, where each row was stored as (word, frequency) in descending order. This is to be utilized by the updated solver.

`freq_organizer.py` illustrates this data cleaning process:

**Processing word frequencies (`freq_organizer.py`)**

```python
import pandas as pd
import numpy as np

words = pd.read_csv('frequency.csv')
wordles = set(open('wordles.txt', 'r').read().splitlines())

# Remove the first row
words = words.iloc[1:]
words.columns = ['word', 'frequency']

# Only keep the words in the wordles list
cleaned_words = pd.DataFrame(words[words['word'].apply(lambda x: x in wordles)] )

# Get the words not in cleaned_words but in wordles list
other_words = [word for word in wordles if word not in
    cleaned_words['word'].values]
other_words_df = pd.DataFrame(other_words, columns=['word'])

# Concatenate and fill NaN frequencies with 0
cleaned_words = pd.concat([cleaned_words, other_words_df])
cleaned_words.fillna(0, inplace=True)
cleaned_words['frequency'] = cleaned_words['frequency'].map(lambda x: int(x))

#save the cleaned words to a new CSV file
cleaned_words.to_csv('cleaned_frequency.csv', index=False)
```

### 2.2.2 Updating the Main Solver

The main script, `Wordle_doer.py`, was then updated. After filtering the word list based on Wordle's feedback, instead of picking a random word, it now selects the word with the highest frequency from the remaining candidates. To implement this, 3 functions had to be heavily modified. The `filter_words` function, which now also considers frequencies, and the `next_word` function.

1. The `filter_words` function was adjusted to process a list of (word, frequency) tuples, rather than a simple list of words.
2. A new function, `update_frequency`, was created to reconstruct the list of (word, frequency) tuples from the filtered word list, preserving frequency information.
3. The `next_word` function was updated to select the first word from the frequency-sorted list, effectively choosing the most popular remaining candidate.

The updated `filter_words` function:

**Updated word filtering logic incorporating frequencies (Wordle_doer.py)**

```python
def filter_words(words, hint, previousWord):

    #filter out the words only from the words list
    newList = set([word for (word, count) in words if word != previousWord])
    letter_counts = Counter()

    # Determine actual counts of hinted letters in the previous word
    for i in range(5):
        if hint[i] in ('g', 'y'):
            letter_counts[previousWord[i]] += 1

    for i in range(5):
        ch = previousWord[i]
        h = hint[i]

        if h == 'g':
            newList = [word for word in newList if word[i] == ch]

        elif h == 'y':
            newList = [word for word in newList if ch in word and word[i] != ch]

        elif h == 'r':
            #if there are no more occurences of the letter in the previous word,
                remove all words with that letter
            if letter_counts[ch] == 0:
                newList = [word for word in newList if ch not in word]
            else:
                #if there are more occurences of the letter in the previous word,
                    remove all words with the letter in the same position
                newList = [word for word in newList if word[i] != ch]

    #convert the newList to a list of tuples
    words = update_frequency(newList, words)

    return words
```

The new `update_frequency` function:

**Reconstructing the frequency-sorted list (Wordle_doer.py)**

```python
def update_frequency(words, freq):
    freq = [(word, count) for (word, count) in freq if word in words]
    return freq
```

The updated `next_word` function:

**Selecting the next guess based on frequency (`Wordle_doer.py`)**

```python
def next_word(words):
    if not words:
        raise ValueError("No words left to choose from.")
    else:
        return words[0][0]
```

This frequency-based selection strategy proved to be much more consistent and generally faster at solving Wordle puzzles, facilitating better testing. For my tests, I manually chose ”SALET” as the starting word, based on common recommendations for statistically good starting words.

## 2.3 Phase 3: Efficiency Testing

Now I had one more question that I wanted to answer. Is my program better than the average human? To answer this, I developed `wordle_eff_tester.py`. This script simulates playing Wordle many times (e.g., 1000 iterations) against randomly chosen words from the `cleaned_frequency.csv` list. It uses the same filtering logic as `Wordle_doer.py` but without browser interaction, making it much faster.

The tester records:

- Average number of guesses.
- Maximum and minimum guesses.
- Number of failures (exceeding 6 guesses).
- A distribution of guess counts (as seen in Figure 4).

The core of the tester simulates the game:

**Efficiency testing simulation (`wordle_eff_tester.py`)**

```python
def eff_tester(num_iter, init_guess):

    #open file
    with open('cleaned_frequency.csv', 'r') as w:
        words = w.read().splitlines()
        words = [tuple(line.split(',')) for line in words]

    #create the variables to hold the statistics
    avg_guesses = 0
    max_guesses = 0
    min_guesses = 100
    failures = 0
    guess_dist = [0] * 7
    for i in range(num_iter):
        if i % 10 == 0:
            print(f'Iteration {i} of {num_iter}')

        word_list = words.copy()
        correct_word = r.choice(words)[0]

        current_game_guess = init_guess
        guesses = 0

        while True:
            guesses = guesses + 1
            hint = make_hint(current_game_guess, correct_word)

            if wd.correct_word(hint):
                break

            if guesses >= 6:
                failures += 1
                break

            word_list = wd.filter_words(word_list, hint, current_game_guess)
            try:
                current_game_guess = wd.next_word(word_list)
            except ValueError:
                failures += 1
                break

        if wd.correct_word(hint):
            guess_dist[guesses] += 1
            avg_guesses += guesses
            if guesses > max_guesses:
                max_guesses = guesses
            if guesses < min_guesses:
                min_guesses = guesses

    successful_solves = num_iter - failures
    avg_guesses_for_success = avg_guesses / successful_solves
    # ... (print and plot results) ...
```

## 3 Results

Executing the `wordle_eff_tester.py` script for 1,000 iterations, using ”SALET” as the initial guess, provided performance statistics that quantify the effectiveness of the frequency-based strategy. The key results were:

- Average guesses: 4.548
- Maximum guesses: 6
- Minimum guesses: 2
- Failures (exceeding 6 guesses): 142

A brief online search suggests the average human player solves Wordle in approximately four guesses. Based on these simulation results, the developed solver does not outperform the average human player. Figure 4 below illustrates the distribution of guesses across these 1,000 simulated games.

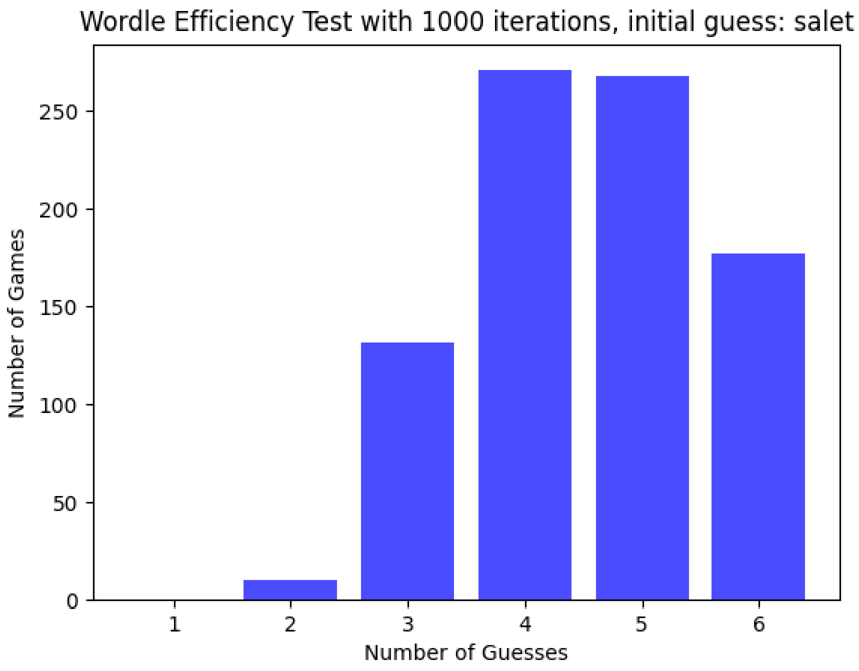


Figure 4: Distribution of guesses over 1000 simulated Wordle games using "SALET" as the initial guess.

## 3.1 So What Happened?

The simulation results, with an average of 4.548 guesses and a failure rate of 14.2%, initially suggest the solver performs worse than the average human player. However, a deeper analysis reveals a discrepancy between the efficiency tester's word selection mechanism and my presumed behavior of Wordle. The tester selects target words for simulation purely at random from the entire list of possible solutions. This contrasts with the hypothesis that Wordle itself prioritizes more common or popular words. Consequently, when the tester randomly selects a word with an assigned frequency of 0 (which occurred in approximately 41% of these test cases), the solver's `next_word` function, by design, deprioritizes such words. This means the solver might only select these very obscure target words as a last resort, potentially leading to more guesses or failures in these specific, artificially challenging scenarios not representative of typical Wordle play.

## 3.2 Refining Target Word Selection in Efficiency Testing

Given the discrepancy highlighted in Section 3.1, an adjustment was made to the testing methodology. Wordle itself favors more common words, so to better simulate this, the pool of words from which the ‘wordle_eff_tester.py‘ script selects the ‘correct_word‘ for each simulated game was modified. Specifically, words with an assigned frequency of ’0’ were excluded from this pool. This refinement aims to create a test set more aligned with the presumed distribution of words encountered in actual Wordle play.

The modification to the word selection process within the tester is illustrated below:

**Modified target word selection in `wordle_eff_tester.py`**

```
# Original list of (word, frequency_string) tuples is 'words'
filtered_target_pool = [word_tuple[0] for word_tuple in words if word_tuple[1] !=
    '0']
# ... other code ...
correct_word = r.choice(filtered_target_pool) # Now chooses from words with
    non-zero frequency
```

Re-evaluating the solver's performance with this updated testing protocol, again using "SALET" as the initial guess over 1,000 simulated games, yielded the following revised statistics:

- Average guesses: 4.319
- Maximum guesses: 6
- Minimum guesses: 2
- Failures (exceeding 6 guesses): 82

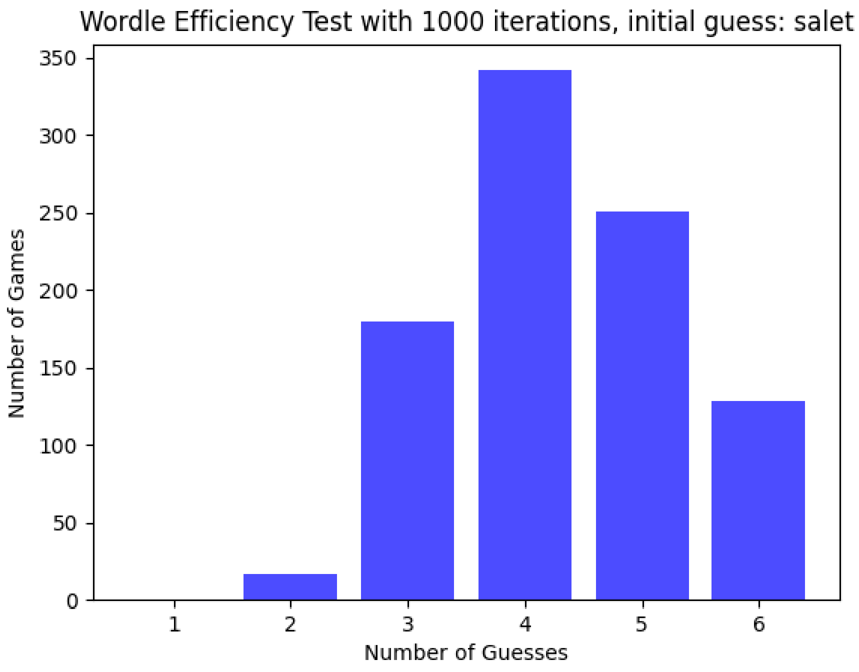


Figure 5: Distribution of guesses over 1,000 simulated games (SALET start), with target words restricted to non-zero frequencies.

A comparison with the previous results (Section 3) reveals a notable improvement: both the average number of guesses and the failure rate decreased. This outcome aligns with the expectation that the solver would perform more effectively when tested against a set of target words that excludes highly obscure (zero-frequency) entries. Furthermore, as illustrated in Figure 5, the distribution of guesses now shows a more pronounced peak around 3 guesses, whereas previously, the frequencies for 3 and 4 guesses were nearly identical. This shift suggests a more consistent and efficient solving pattern when the test conditions better reflect the solver's underlying frequency-based strategy.

## 4 Discussion and Future Work

Now that my goal of automating wordle has been completed, evolving from random guessing to a frequency-based approach, there are still several points to improve on should I choose to come back to this:

1. **Optimizing Second Guess Strategy:** Currently, the bot picks the most frequent valid word. An alternative strategy, particularly for the second guess, could be to choose a word that maximizes the information gained, i.e., a word that is most likely to eliminate the largest number of remaining possibilities, even if that word itself is less frequent. This involves evaluating potential guesses based on how they would partition the remaining word set given all possible color-feedback combinations. This could be tested against the current frequency-based strategy.

2. **Parallelizing Efficiency Tester:** The `wordle_eff_tester.py` script runs trials sequentially. For a very large number of iterations or more complex strategies, this can be time-consuming. Parallelizing the trials (e.g., using Python's `multiprocessing` module) could significantly speed up the testing process and allow me to test every word at once to see which one is the "best".

3. **Advanced Starting Word Analysis:** While "SALET" is a good starting word, I could use more complex methods to find the best starting word. For example, find the word that leaves the least number of words after the first iteration, regardless of what the actual word is.

4. **Further Refining the Efficiency Tester's Word Selection:** While excluding zero-frequency words from the target pool marked an improvement in aligning the test conditions with the solver's assumptions, the selection mechanism could be enhanced further. Currently, all non-zero frequency words are chosen with equal probability within the refined tester. A more sophisticated approach would involve implementing a weighted random selection system, where words with higher recorded frequencies would have a proportionally greater probability of being chosen as the target word for simulation. This would more closely emulate the hypothesized behavior of Wordle prioritizing popular words and could provide an even more accurate assessment of the solver's real-world performance.

5. **Improving the Underlying Word Frequency Dataset:** The overall efficacy of the frequency-based Wordle solver is fundamentally dependent on the quality and accuracy of the word frequency dataset used (`cleaned_frequency.csv`). The current dataset, while useful, was compiled from a general web scrape. Sourcing or creating a more comprehensive and contextually relevant frequency dataset—perhaps one specifically tailored to common English vocabulary, literary sources, or even analysis of past Wordle solutions if such data were available—could significantly enhance the solver's ability to prioritize the most likely candidates and, consequently, improve its solving efficiency and reduce failures. A dataset with finer granularity in frequency distinctions, rather than many words simply being 'non-zero', would also be beneficial.

## 5 Conclusion

The automation of Wordle ended up being a lot more rewarding than I initially thought. What started as a simple project ended up teaching me more about merging web technologies, data handling, and algorithmic thinking. Through this project, I gained deeper insights into web scraping using Selenium, acquired practical experience in cleaning and transforming data, and further honed my understanding of the iterative development process. Even though initially this was just a project for my amusement, it ended up transitioning from a basic concept to a data-driven solver, a journey that highlighted the power of refining ideas and leveraging available information. Ultimately, the resulting tool not only conquers Wordle, but also provides many opportunities to explore future optimizations in the wordle solving world.