50.021 Artificial Intelligence

Wordle - The New York Times Solver using Reinforcement Learning, Clustering and Search Algorithms

---

Final Report

Group 3

Authors	*Student ID*
Lee Min Shuen	1004244
Samuel Sim Wei Xuan	1004576
Riley Riemann Chin	1004147
Tan Hong Yew	1004235

August 1, 2022

Contents

1 Introduction

1.1 Motivation

Every year, many fun simple games take over the internet and social media. These addictive games include the famous "Flappy Bird", which gained immense popularity in 2014. In the current year, 2022, the game that took over the internet is undoubtedly the famous "Wordle" game.

"Wordle" is a web-based word game developed by Welsh software engineer Josh Wardle in 2013 and published in 2021. In 2022, the game was acquired by "The New York Times Company, which exploded into one of the world's most popular games, bringing about tens of millions of new users [1].

Figure 1: Example of Wordle [2]

The game, which refreshes daily, involves guessing the "Wordle" of the day. Each guess must be a valid five-letter word, and only six tries are allowed. After each incorrect guess, the colour of the tiles will change accordingly to give the player clues about how close the player's guess is to the goal word of the day.

Ever since the rise in popularity, there have been many talks about how one solves the game faster and more efficient. For example, a famous math YouTube channel called 3Blue1Brown, run by Grant Sanderson, aimed to find the best starting word using information theory. Following this, there have been many discussions on building a "Wordle AI Bot" to solve the daily word. Therefore, our motivation is if we can apply artificial intelligence techniques and create our own "Wordle AI Bot".

1.2 Task Description

Firstly, we need to understand how the game is being played. Here is a screenshot of the official instructions taken from The New York Times [3]:

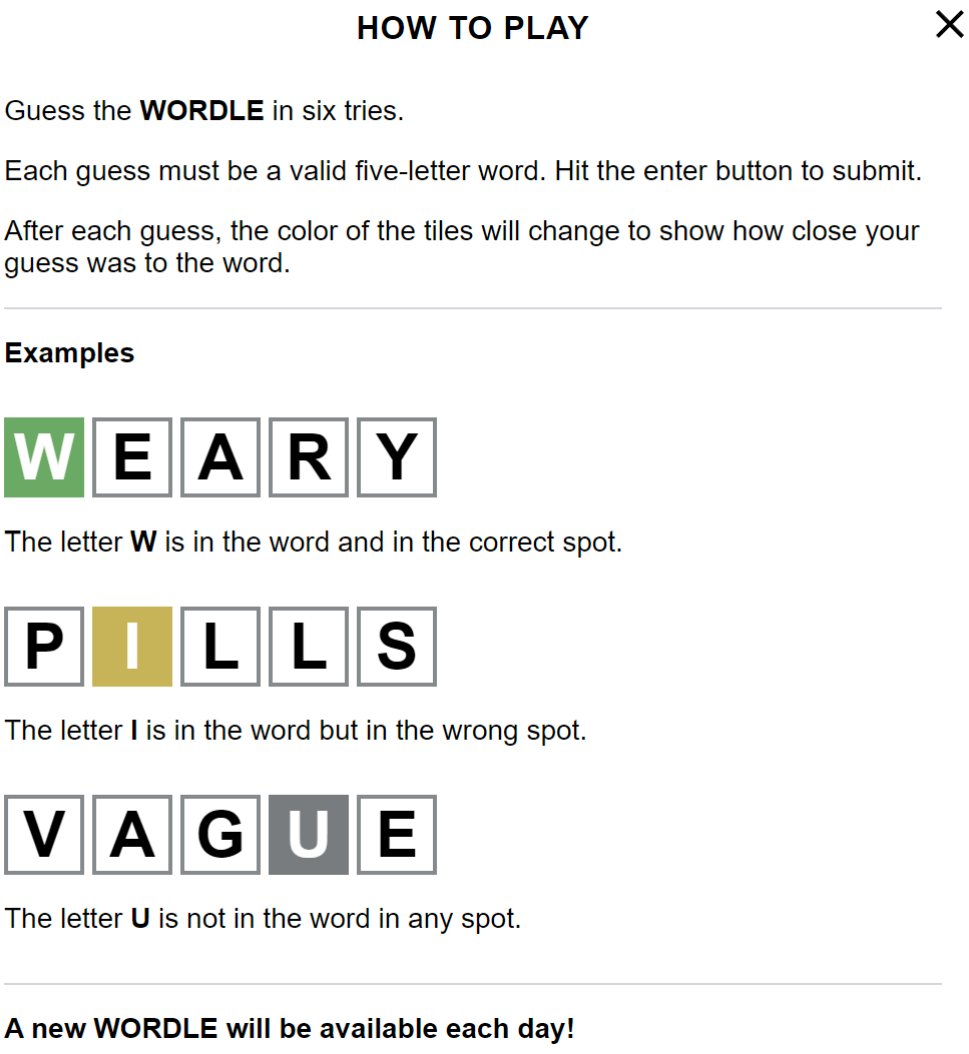

Figure 2: Wordle How to Play Instructions [3]

In the settings, there is a hard mode whereby the player must use any revealed hints in subsequent guesses.

Task: To use artificial intelligence techniques to help build a bot to solve the daily "Wordle" (goal word) in hard mode.

Proposed Algorithms:

1. Reinforcement Learning
2. Clustering Algorithms
3. Search Algorithms

1.3 General Strategy

For our Wordle solver, we decided to start with the word "CRANE" for all models and their variations. This choice was due to Grant Sanderson again, who used information theory on his YouTube channel 3Blue1Brown to find out which starting word offered the greatest chance of success. Additionally, having the same starting word for each Wordle solver approach would make subsequent performance analysis less biased.

2 Dataset Preparation

2.1 Data Description

With regards to the dataset, the only data our models require is the list of 5 letter words. While there are many 5 letter words in the English language, the developer of "Wordle" probably has limited the answer and feasible words. This is from our own experience of playing the game itself.

2.2 Data Extraction

As such, our group indirectly went to the source code by inspecting the web game's HTML and javascript attributes.

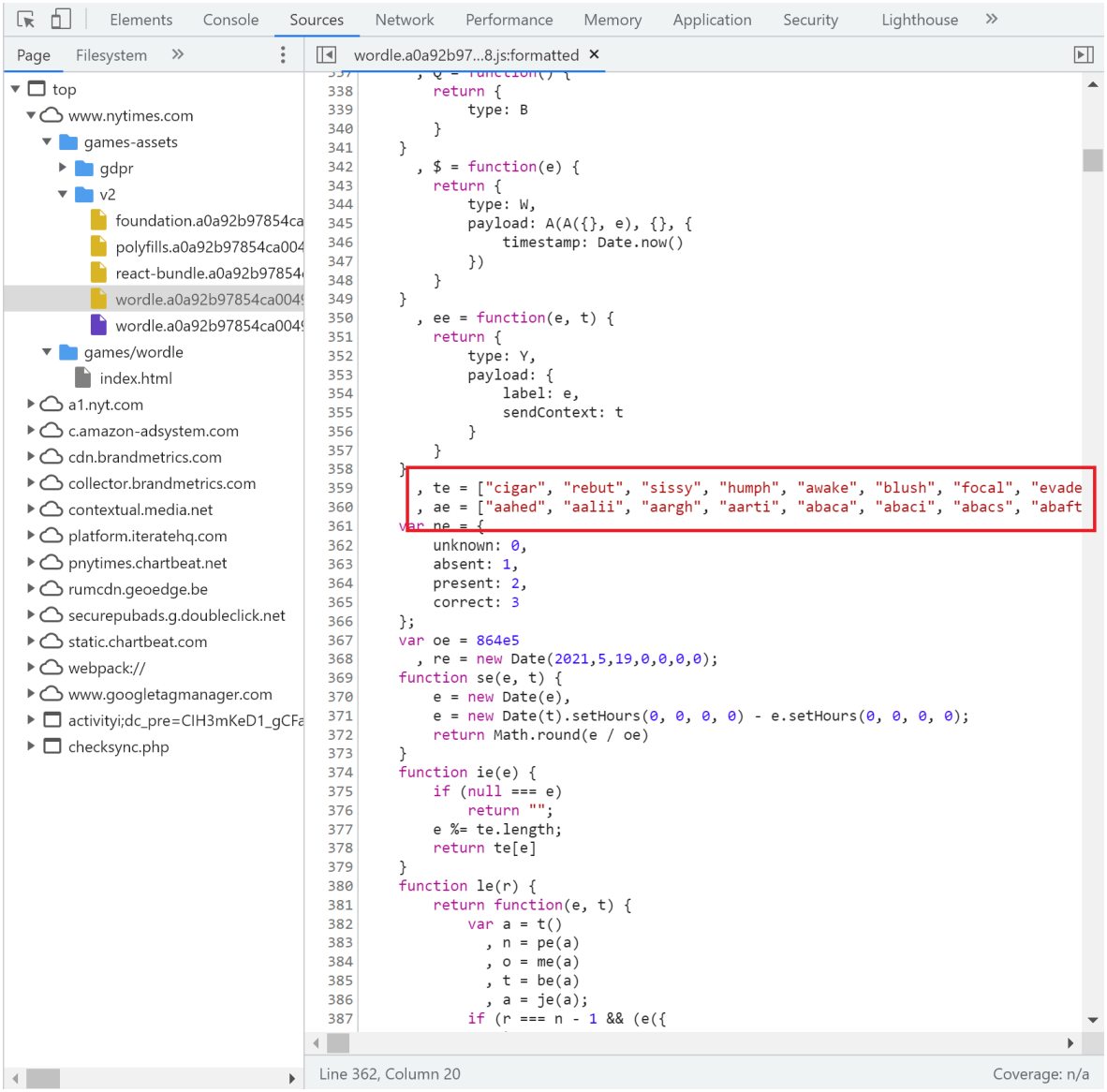

Figure 3: Extracting Data from Source Code [3]

As seen above (red box), we found two unique word lists from one of the source code javascript files. The first list, “te”, consists of 2309 goal words that are looped through daily as the daily wordle. The second list, “ae”, consists of 12974 feasible and accepted 5-letter words.

2.3 Data Loading

Since this is not a typical machine learning problem but rather an application of machine learning to tackle an online game. Our proposed model is specific to the “Wordle” game. As such, we will train our proposed models on the combined

corpus of both lists. Additionally, the daily wordle will be taken from the second (goal) list as per how the game is designed.

3 Models and Strategies

In this section, we will be looking at the following:

1. Baseline Reinforcement Learning (RL) Model
2. RL with Hierarchical Clustering Model
3. Modified RL with Hierarchical Clustering Model
4. Greedy Search Model
5. Modified Greedy Search Model

3.1 Baseline Reinforcement Learning (RL) Model

3.1.1 Q-Learning Approach

As a baseline reinforcement learning model, our group decided to build the most straightforward model we could think of. Our group decided on using Epsilon Greedy Q-Learning, the basic form of vanilla value-based reinforcement learning. Q-learning seeks to learn a policy that informs the agent on what action to take in the given state of the environment.

The limitation and challenge of Q-learning are defining the discrete state space and actions to take such that the agent can explore and exploit his current experience to learn the "best" action to take in the current state. Another challenge is developing our reward system for the agent that balances the current and future rewards.

3.1.2 Model Definition

Since we are creating a baseline model and understanding how "Wordle" is being played. As an agent, the most straightforward state-action pair possible is the current guess and the next guess to take.

States: List of words from the entire corpus of 15283 words

Actions: The next guess from the same corpus

Regarding the reward system of choosing a particular state-action pair and updating the value in the Q-table, our group decided to leverage the hint colouring system that the game has. We know that a green box implies that a letter of the correct position has been found, while a yellow box suggests a letter of the incorrect position has been found. Lastly, a black box implies that the letter is not part of the current day's wordle.

Reward System:

1. +10 reward points for additional green letter found
2. +5 reward points for additional yellow letter found
3. -1 penalty on reward points for additional black letter found

Using a +10 reward as the baseline, since finding an additional yellow letter would not be ideal compared to finding an additional green letter, we gave half the reward of +5 for finding an additional yellow letter and +10 for an additional green letter. For finding an additional black letter, even though it helps us limit the corpus of words to search from, it wastes a guess of finding additional green or yellow letters that exist in the goal wordle word. Therefore, we gave finding an additional black letter a minor penalty of -1.

3.1.3 Hyper-parameters

Since we are implementing an epsilon greedy Q-learning algorithm, we have the following hyper-parameters from the Q-learning iterative equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R(s,a) + \gamma \max Q'\left(s', a'\right) - Q(s,a)\right]$$

Figure 4: Q-learning Bellman Equation

1. **Learning rate** $0 < \alpha \leq 1$**:** Step size for incrementing the Q-value

2. **Discounting factor** $0 \leq \gamma \leq 1$**:** Discount on the maximum expected future reward, to prevent exploding Q-table values

3. **Rate of Exploration** $0 \leq \epsilon \leq 1$**:** Probability of our model selecting a random action and exploring, instead of exploiting previous knowledge through the Q-table.

For the choice of the hyper-parameter values, we will use a grid search to determine the optimal values to set as the default values based on the win rate of our models. This will be further explored in the evaluation section.

3.1.4 Additional Components

Recall earlier, we wanted to solve in hard mode. The hard mode requires our model to limit the next guess using the hints provided in the earlier guesses. Even though this is harder for a human to do, it will be simple for our wordle AI solver to do so. Our team included a **filtering function** to help reduce the size of the corpus down to the feasible words remaining after each subsequent guess.

Note that our Q-table is a 2-dimensional array of state-action pair values, $Q(s, a)$, with the rows and columns indexed to the corpus of words. The filtering function allows us to slice down the Q-table accordingly. The filtering would be a limitation of our baseline model mentioned under limitations.

Lastly, since the actual "Wordle" game has only 6 tries, we would want to explore in the first few tries; however, exploit once we are near the last few remaining attempts. Therefore our group decided to implement an additional component to the basic Q-learning algorithm by introducing a decay over time to the rate of exploration, ϵ.

$$\epsilon \leftarrow \frac{\epsilon}{\text{step-number}^2}$$

Figure 5: Decaying epsilon inverse square function, step-number is the current number of guess

Hence, our group applied an inverse square decreasing function on ϵ. We used an inverse square function instead of an inverse function, 1/step-number. Our group

wanted to drastically decrease our rate of exploration after the 3rd or more guesses (step-number) since our goal is to solve for the goal wordle within 6 tries.

3.1.5 Limitations

For our baseline model using Epsilon Greedy Q-Learning value-based reinforcement learning, here are some limitations:

1. Huge state space/large Q-table with an initial size of 15283 by 15283 before any guesses and filtering. Therefore, the average number of guesses required might be way more than 6.
2. After each run of "Wordle", the Q-table is reinitialized to a zero-matrix instead of carrying on to the next iteration. This is due to the slicing of the Q-table during each run's filtering.

3.2 RL with Hierarchical Clustering Model

3.2.1 Hierarchical Clustering

For our next model, we decided to address the limitations of our baseline model. The first thing to address is the choice of state and actions to reduce the state space size and allow the re-using of the learned knowledge of the Q-table from previous runs.

As such, the idea of clustering came to our group's mind: cluster similar-looking words together. However, in terms of similarity for "Wordle", the similarity measure should not include any sentiment meaning of the words. Hence instead of visualizing the corpus as a word vector space and using word embedding, our group decided to use a tree structure. Therefore a hierarchical tree structure seems the most appropriate for our model.

The next consideration is to use a top-down or bottom-up clustering approach. Since we are trying to group similar words or nodes, we decided to use a bottom-up approach.

Hence, our group decided to go with an agglomerative hierarchical clustering method.

The last consideration is what type of pair-wise distance measure to use for our clustering. As mentioned earlier, we do not want to consider the sentiment meaning of the words. Since "Wordle" hints evaluate the similarities between the letters of the guess and the goal word, our group decided on the Levenshtein distance measure.

Levenshtein Distance: Minimum number of single-character edits (letter insertions, deletes or substitution) required to change one word to another. For example the Levenshtein distance between *CRANE* and *SHAPE* is 3.

3.2.2 Model Definition

Before running the Q-learning, we will be doing clustering on the corpus. Hence the current state of the game would be the current cluster that we are looking within, and the action, in this case, would be the next cluster to choose from. Note that the Q-tables are square matrices for both current models as the states and actions are similar, which are the current word/cluster and the next word/cluster, respectively. There will be no change to the reward system.

States: List of the cluster number

Actions: The next cluster to select the next guess from

Reward System:

1. +10 reward points for additional green letter found
2. +5 reward points for additional yellow letter found
3. -1 penalty on reward points for additional black letter found

3.2.3 Hyper-parameters

Since we are implementing the same epsilon greedy Q-learning algorithm, we have similar hyper-parameters as before. However, since we are doing clustering, an extra hyper-parameter would be the number of clusters to produce. For the linkage, we will use the average linkage option, which minimizes the average of the distances between all observations of pairs of clusters. Our group will also do a grid search on the following hyper-parameters to get their default values. This will be further

explored in the evaluation section.

3.2.4 Additional Components

Since the model is still required to guess a word from the chosen cluster; our team still kept the **filtering function** which helps to reduce the corpus size down to the feasible words remaining after each subsequent guess.

Unlike the baseline model, our Q-table is a 2-dimensional array of state-action pair values, $Q(s, a)$ indexed to the cluster number. This new model would not need to slice the Q-table and does not need to be reinitialized to a zero-matrix after each run. Therefore our agent would be able to learn from previous runs, which covers our baseline model limitations.

Nevertheless, we kept the decaying rate of exploration, ϵ, from our baseline model.

3.2.5 Limitations

For our agglomerative hierarchical clustering Q-learning-based model, the choice of words from the chosen cluster is still random. A possible room for improvement is to incorporate some form of discretization of words within each cluster.

3.3 Modified RL with Hierarchical Clustering Model

For our last model, based on our results under the evaluation section, we decided on a slight modification. Instead of training on the entire corpus of 15283 words, since the daily "wordle" comes from the same list and loops through it daily. Our group decided to narrow our corpus to the 2309 goal words. Other than that, the architecture of everything else and hyper-parameters are the same.

3.4 Greedy Search Model

3.4.1 Greedy Search Approach

The above three variations use reinforcement learning. However, our group decided to compare the above reinforcement learning approaches to a greedy search algorithm.

3.4.2 Model Definition

Similar to the Q-Learning Approach, as an agent, the most straightforward state-action pair possible is the current guess and the next guess to take.

States: List of words from the entire corpus of 15283 words

Actions: The next guess from the same corpus

At each node, the algorithm selects the next node with the best option available, working in a top-down approach.

The score of each letter in the alphabet is first obtained by frequency of occurrence in words from the entire word list. From this, we are able to obtain a score for each word by summing up the score of the individual letters of the word.

The word scores will be used as our heuristic function which lets us estimate how close the current state is to the goal state. We expand to the node with the lowest heuristic after evaluating the green, yellow followed by grey letters.

3.4.3 Limitations

The Greedy Search Algorithm works by looking for the optimal local solution at every step. However, this does not guarantee that the optimal global solution is found. As a result, the algorithm cannot guarantee the success of finding the correct word within the maximum number of tries.

Another common limitation of this algorithm is the lack of completeness, which is the possibility of getting stuck in loops. However, this is not possible in our case as each state (word) carries information which prevents it from ever coming back as a possible option at a later stage.

3.5 Modified Greedy Search Model

To compare against the Modified RL with Hierarchical Clustering Model. Similarly, instead of using the entire corpus of 15283 words, we narrowed down our corpus to the 2309 goal words themselves. Besides that, everything else remains the same as the Greedy Search Model.

3.6 Overall Strategy

In summary, the following strategies will be used when running each model.

RL baseline Model:

1. Initialize our *Wordle* class which creates the Q-table and randomly selects a daily wordle from the list of goal words
2. Uses the Q-learner to select the next guess.
3. Passes the guess to our Evaluation class which gets the scoring and filters the corpus down to remove any unfeasible words.
4. *Evaluation* class then passes the reward to update the Q-table for the next iteration of the current run
5. Each run terminates when the goal word is reached, Q-table is reinitialized to zero.

RL with Clustering Models:

1. Initialize our *Wordle* class which creates the Q-table and randomly selects a daily wordle from the list of goal words
2. Initialize our *Clustering* class to cluster on the corpus
3. Uses the Q-learner to select a cluster from which to pick the next guess. Currently, the choice of word from the chosen cluster is random.
4. Passes the guess to our Evaluation class which gets the scoring and filters the corpus down to remove any unfeasible words.
5. *Evaluation* class then passes the reward to update the Q-table for the next iteration of the current run
6. Each run terminates when the goal word is reached and is ready to pass the learned Q-table to the next run

Greedy Search Models:

1. Randomly select a daily wordle from the list of goal words
2. Initialize our word score dictionary based on the occurence of words
3. Evaluate guess result based on green letters followed by yellow letters and then grey letters. Select the best option.
4. Repeat step 3 for the next iteration of the current run
5. Each run terminates when the goal word is reached

4 Evaluation

4.1 Hyper-parameters Grid-search

As mentioned in the earlier sections, we have various hyper-parameters to select. In order to justify our choice of hyper-parameters, our group decided to do a grid search on them. Here are the following hyper-parameter values we will be searching on (the number of clusters only applies to the model with clustering):

1. **Learning rate α:** [0.1, 0.01, 0.001]
   We will search on the traditional learning rate values of 0.1, 0.01 and 0.001. Our group decided to stop at 0.001 as we require our bot to solve in as few steps as possible; too low of a learning rate might result in slow convergence timing. On the other hand, we start the search at 0.1, as too high of a learning rate might result in divergence and thus unable to find the goal word.
2. **Discounting factor γ:** [0.5, 0.6, 0.7, 0.8, 0.9]
   Based on an article [4], the typical values are 0.9 or 0.99. However, since we are more focused on the intermediary rewards than the long-term rewards, our bot aims to solve within 6 guesses. Similarly, as discussed in the article, a lower discount factor-like, 0.8, would be ideal. Hence we try values lower than 0.9 up to 0.5; otherwise, our model will never learn.

3. **Rate of Exploration ϵ:** [0.5, 0.6, 0.7, 0.8, 0.9]

   We chose the same values as the discounting factor. As mentioned previously, our model aims to solve within 6 tries; hence we applied the decaying inverse square function. Therefore, we need a higher starting rate of exploration that is not too small. As such, the lower bound value is 0.5.

4. **Number of Clusters:** [6, 7, 8, 9, 10]

   For the number of clusters, we do not want the number of clusters to be too large; otherwise, the state-space and Q-table will be large. As such, our group chose an upper bound value of 10. Furthermore, from another person's attempt [5] at using another form of clustering for a similar task, the choice of the number of clusters was 5. Therefore, we set the lower bound value to just one higher value arbitrarily.

With all the grid-search values in place, our group ran every combination over 100 simulation runs of the game. Each run selects a random goal word from the list of goal words. To pick the best combination of hyper-parameters, our group decided to evaluate the combinations on three metrics (in order of highest importance on top):

Three Metrics:

1. Highest win rate (Percentage of successful "Wordle", solved within 6 tries)
2. Lowest average number of guesses (not capped to 6 tries)
3. Shortest time taken for 100 simulations

If there are two combinations of hyper-parameters with the same win rate, we will choose the combination with the lower average number of guesses. Likewise, if there is the same win rate and average number of guesses, we would choose the combination with a lower time taken.

Also, our group did the grid-search on only 100 runs, as the entire grid-search takes around 2 days to execute finish due to the vast number of possible combinations of hyper-parameters.

4.2 Results of Grid-search

Here are the results of our grid-search and our default values for the hyper-parameters:

	RL Base	RL + Clustering	Modified RL + Clustering
Learning rate, α	0.1	0.1	0.001
Discounting factor, γ	0.8	0.9	0.9
Rate of Exploration, ϵ	0.8	0.5	0.9
Number of Clusters	NA	6	9

Table 1: Grid-search hyper-parameters results

4.3 Model Performances

Our group decided to do a simulation study to evaluate each model's performance properly. Since our simulations are time-consuming, our group used the batch-means approach of simulation:

Simulation steps for each of the 5 models:

1. Run one long simulation run of length 10000
2. Plot out the number of guesses required to find the goal wordle over time
3. Specifically for the reinforcement with clustering models, determine if the number of guesses reduced over time which signifies that our agent is learning better after each individual run
4. Divide the 10000 simulations into batches of 1000
5. Discard the first batch to prevent any possible initial bias even if no burn-in period is observed.
6. Compute the mean and variance of the average guesses and the average win-rate of each batch
7. Compute the t-confidence interval with those mean and variance estimates at 95% confidence

4.3.1 Number of guesses over 10000 runs

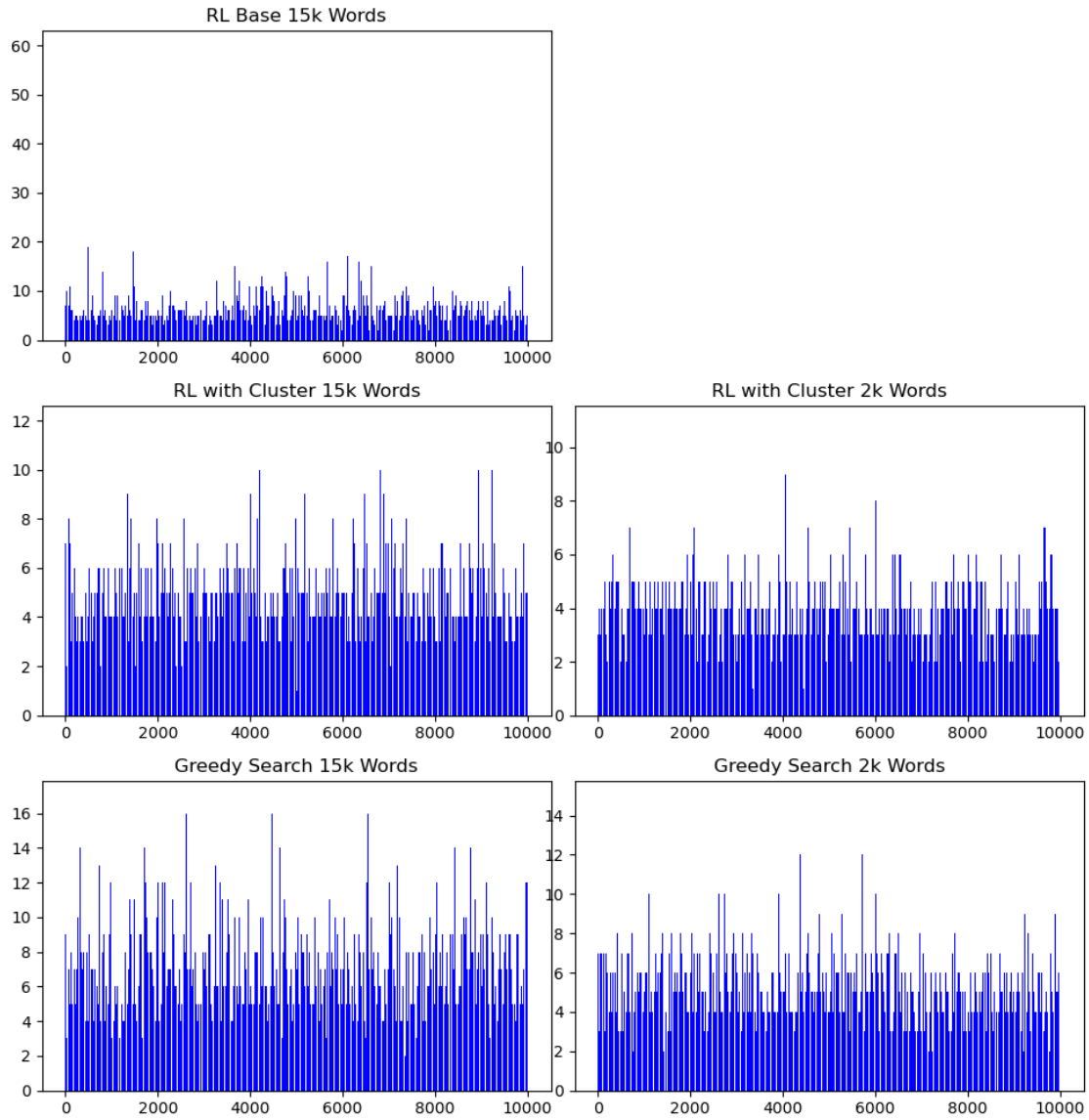

Figure 6: Number of guesses over 10000 runs for the 5 Models

From the above bar plot of the number of guesses over the 10000 runs, specifically for the "RL with Cluster" models in the second row, we do not observe a decrease over time in terms of the number of guesses required. Therefore, this could imply that while the agents learn from each previous run (recall, the Q-table is not reinitialized to a zero-matrix after each run), the performance does not improve much. This will be covered under the improvement section.

4.3.2 Average guess metric

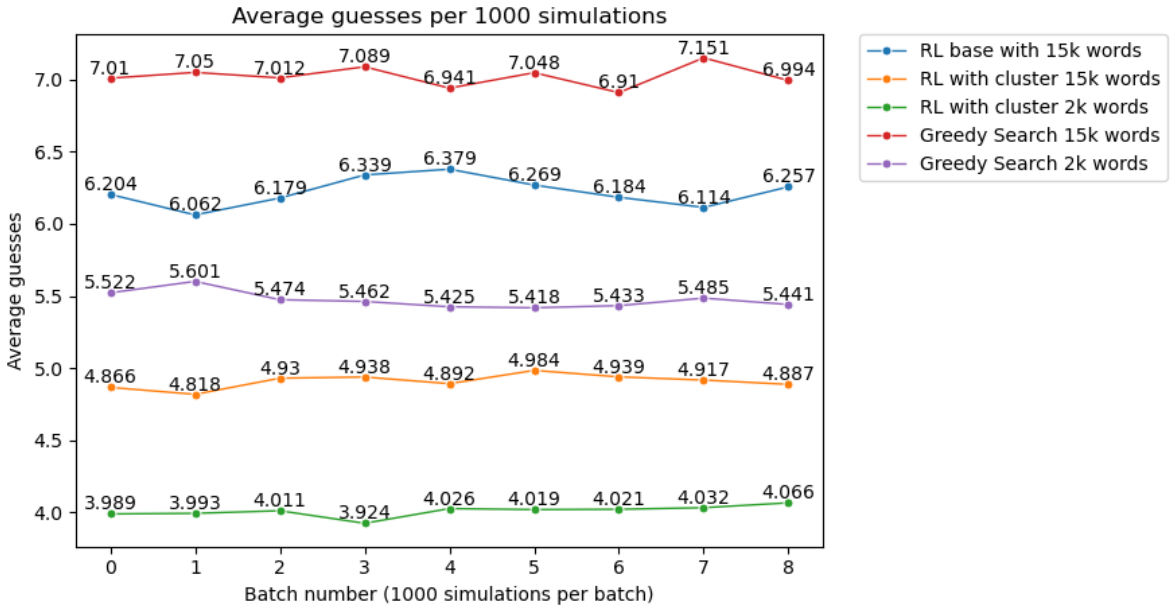

Figure 7: Average guesses per batch for the 5 Models

As mentioned earlier, after removing the first 1000 runs, our group divided the remaining runs into 9 batches of 1000 runs each. Then we calculated the average number of guesses for each batch for all 5 models and plotted them into a line plot shown above.

	Mean of average-guesses	Variance of average-guesses	95% t-Confidence interval of average guesses
RL Base (15k words)	6.22	0.00918	(6.213, 6.228)
RL with Clustering (15k words)	4.91	0.00208	(4.906, 4.91)
RL with Clustering (2k words)	4.01	0.00136	(4.008, 4.01)
Greedy Search (15k words)	7.02	0.00475	(7.019, 7.027)
Greedy Search (2k words)	5.47	0.003	(5.471, 5.476)

Table 2: Mean, Variance and 95% Confidence Interval of average-guesses for the 5 Models

Next, after averaging the number of guesses for each batch for all 5 models, we calculated the mean and variance of the result and found the 95% t-confidence interval. Our group used a t-test as the number of batches (samples) is 9, which is smaller than 30. Hence, using a t-distribution is a better choice than a z-distribution.

We can see that in terms of the number of guesses, with the aim of a lower number of guesses:

- For reinforcement learning models, clustering outperforms the baseline that does not include clustering
- For the same corpus size, reinforcement learning with a clustering approach outperforms the greedy search approach
- For the same model type, narrowing down the entire corpus to the 2309 goal words outperforms using the entire corpus of 15283 words

Result: Using the reinforcement learning with clustering model that is trained only on the 2309 goal words is the best choice based on the average guesses since we can be 95% confident that true average guesses over 1000 runs would lie between 4.008 and 4.01 guesses.

4.3.3 Average win-rate metric

Repeating the same steps for the win-rate (if the number of guesses is 6 or below, it is considered a win):

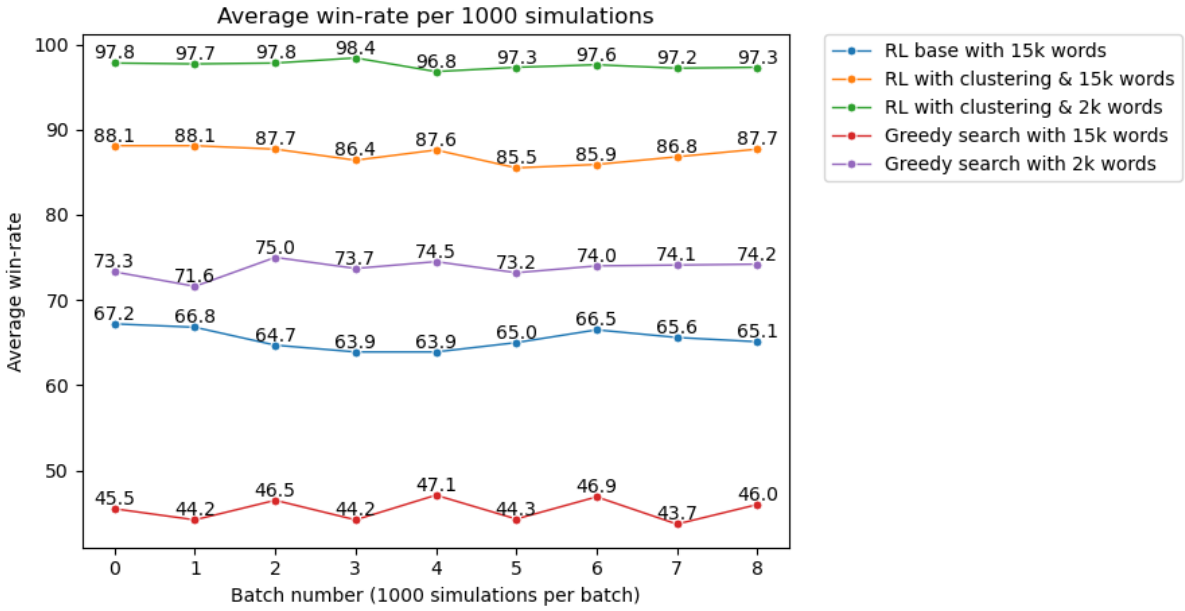

Figure 8: Average win-rate per batch for the 5 Models

	Mean of average win-rate	Variance of average win-rate	95% t-Confidence interval of average win-rate
RL Base (15k words)	65.4%	1.3	(64.4%, 66.5%)
RL with Clustering (15k words)	87.1%	0.839	(86.4%, 87.8%)
RL with Clustering (2k words)	97.5%	0.187	(97.4%, 97.7%)
Greedy Search (15k words)	45.4%	1.52	(44.1%, 46.6%)
Greedy Search (2k words)	73.7%	0.849	(73.0%, 74.4%)

Table 3: Mean, Variance and 95% Confidence Interval of average win-rate for the 5 Models

Again we can see that in terms of the win-rate, with the aim of a higher win-rate, the same observations are observed:

- For reinforcement learning models, clustering outperforms the baseline that does not include clustering
- For the same corpus size, reinforcement learning with a clustering approach outperforms the greedy search approach

- For the same model type, narrowing down the entire corpus to the 2309 goal words outperforms using the entire corpus of 15283 words

Result: Using a reinforcement learning with clustering model that is trained only on the 2309 goal words is the best choice regarding win-rate since we can be 95% confident that the true average win-rate over 1000 runs would lie between 97.4% and 97.7%.

4.3.4 Total time taken metric

The last metric we can compare all the models against each other is the time taken to run a "Wordle" game by averaging across the 10000 runs.

	RL Base (15k words)	RL with Clustering (15k words)	RL with Clustering (2k words)	Greedy Search (15k words)	Greedy Search (2k words)
Time taken per run/s	0.123	0.136	0.00923	0.0513	0.00735

Table 4: Time-taken per run of "Wordle" for the 5 Models

Result: The greedy search model that uses only the 2309 goal words is the best choice regarding the time taken with an average run time of 0.00735s.

4.4 Choice of Model

Given the results from our simulation analysis, our group have finalized in using the reinforcement learning with clustering model, that is trained only on the 2309 goal words, as our choice of model for our "Wordle" AI solver. Afterall, the time taken is the least important performance metrics, while the win-rate is the most important.

However, this exploit of the goal words list and training specifically on it, is only plausible given that our AI solver is to win this specific game. A variation of "Wordle" whose daily wordle words come from the entire corpus of feasible 5 letters word, would then force us to use the more generic reinforcement learning model that trains on an entire corpus of feasible 5 letter words.

5 Graphical User Interfaces (GUIs)

5.1 Model Performances GUI using Kivy

5.1.1 GUI Objective

To allow our target readers to understand and interpret the results, we utilized a library called *Kivy* available in Python to develop a Graphical User Interface (GUI). The GUI would include selections for different models, allow selection for different values for the Learning Rate, Exploration Rate, Exploitation Rate based on the Q-Learning formula, and also provide a visualisation of the model performance.

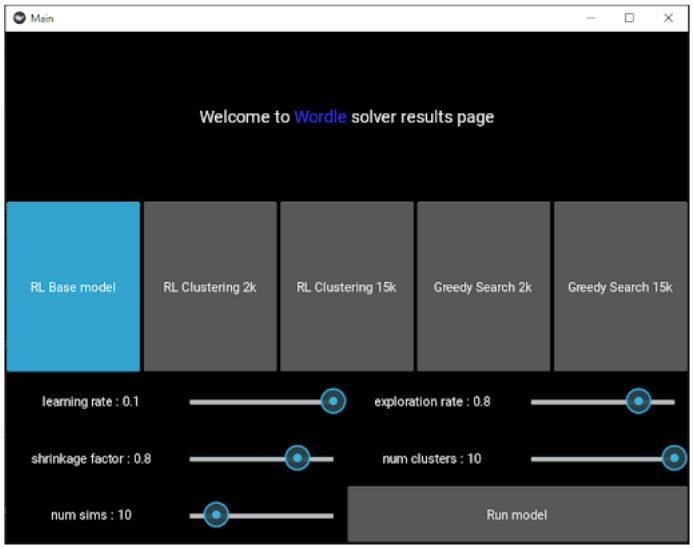

Figure 9: Kivy GUI

5.1.2 Slider Parameters

From our grid-search, we were able to converge the hyper-parameter values for the Baseline and Cluster-based models. These values were set as the default values for the different models.

5.1.3 Interpretation of Results

With the GUI, we can properly see and compare how the different models vary in performance with respect to Average Guesses, Time Taken, Win Rate and overall guesses per epoch. We set the range of the radar chart to be for 1-15 for

Average Guesses, 0-100 for Win Rate, and the max value of time taken to be from 0 to the maximum ran model time.

Figure 10: RL Base performance

The above figure shows 2 visualizations, the bar chart on the top shows the number of guesses per epoch, and the below radar chart shows in 3 axes, the Average Guesses, Time Taken and the Overall Win Rate. By itself, it is hard to interpret what is a good result, but in the next figure we can see how the comparisons are actually made.

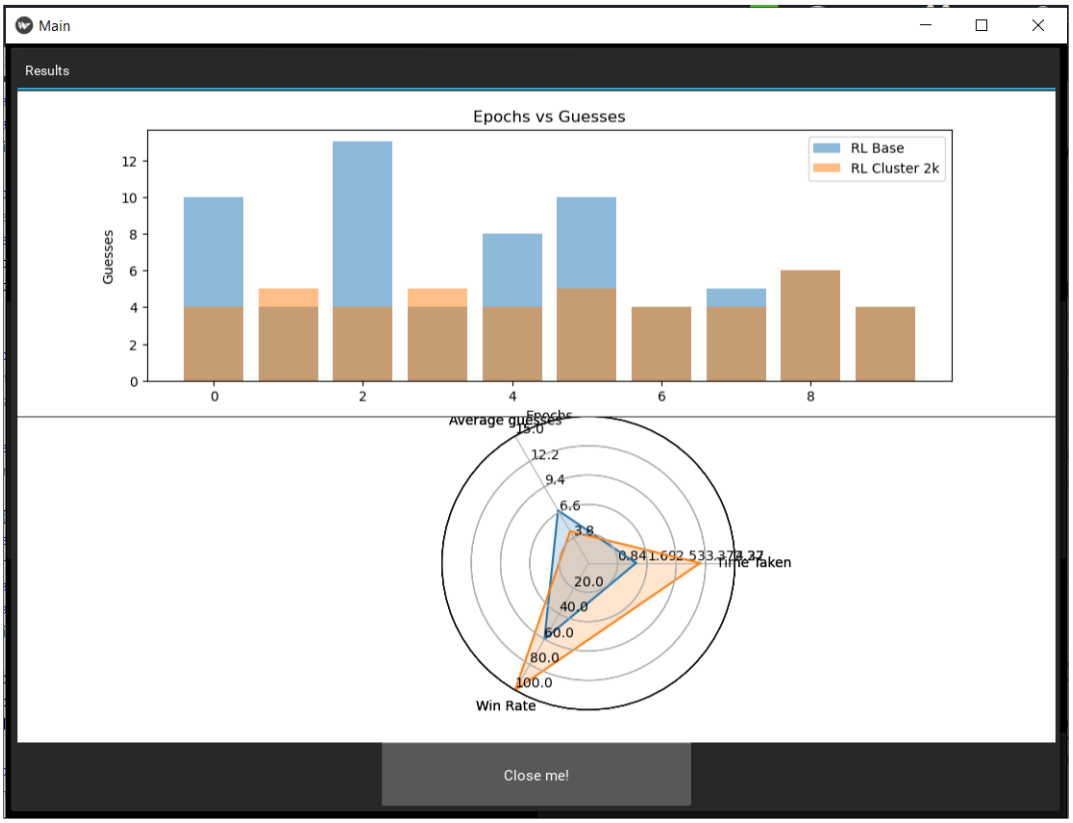

Figure 11: RL Base vs RL Cluster 2k performance

With these 2 models, we can clearly see that RL Cluster (orange color) performs better than the base model (blue color) based on certain metrics as the Win Rate extends more, the Average Guesses extends less but also with the trade off that the time taken significantly increases. This GUI also allows us to confirm our test results from the evaluation section where we expect a certain model to be the best performing.

5.1.4 Custom Parameters

This slider parameters also enables us some form of testing if we want to change some of the slider values and see which ones affect our model more. The user can adjust the sliders from a preset value of 0-1 for the Exploration Rate, Learning Rate and Shrinkage Factor. While the default values might necessarily provide the best performance, we also are able to see how each parameter affects individually.

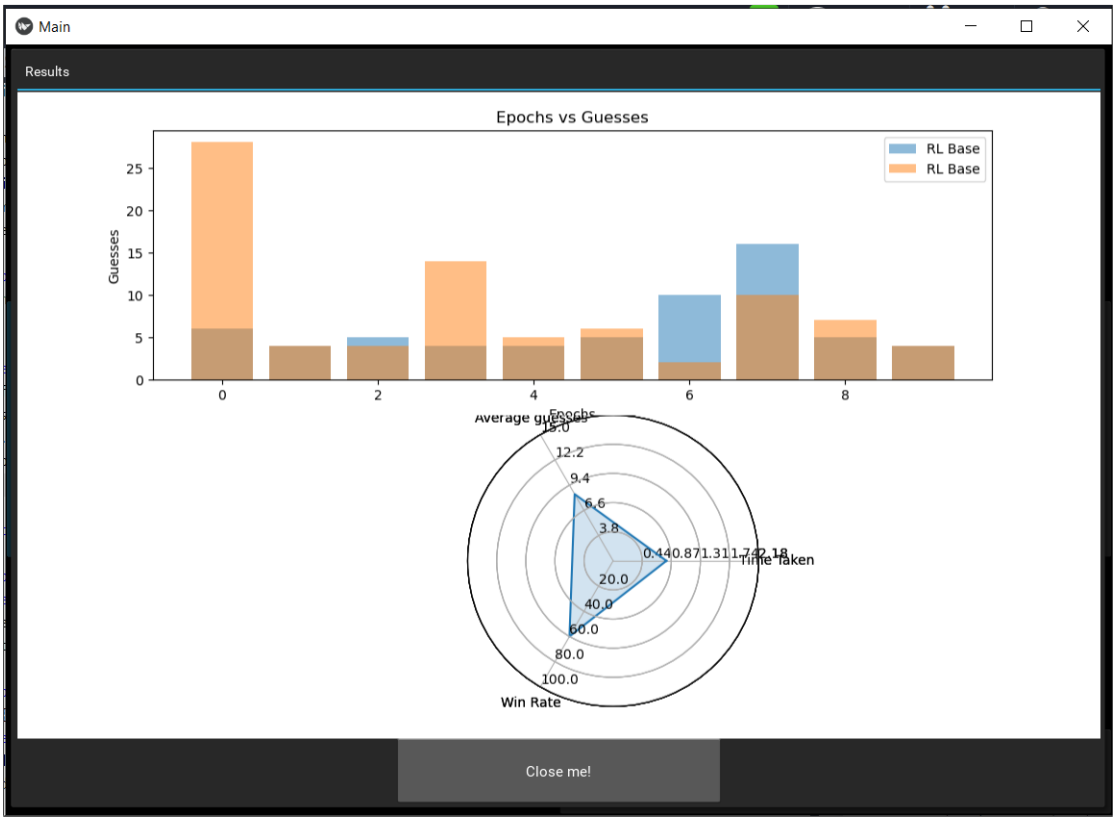

Figure 12: RL Base vs RL Base varied Learning Rate performance

Take Figure 12 above for example, the variation of learning rate from the optimal 0.1 on the blue bar chart to 1.0 on the orange bar chart shows that initially it would take longer for the model to guess the word correctly. This inference holds the same for the different parameters as well and it helps to visualize this in a GUI format.

5.2 Daily Wordle GUI using Pygame

5.2.1 GUI objective

To allow our users to solve the daily wordle using our choice of the modified reinforcement learning approach and clustering with the default optimal parameters.

5.2.2 Code Setup

Given our choice of the reinforcement learning approach with clustering on the goal word list, we ran an even larger number of 100000 runs and saved the final Q-table. This learned Q-table will be initialized during every new run of our

"Wordle" AI bot in Pygame.

The pygame python file is split into two main portions:

- AI "Wordle" algorithm consisting of our custom classes for reinforcement learning with clustering on 2k word list
- Pygame environment, whose UI components are adapted from the following git repository[6]

Upon running an instance of the game, the Pygame environment will run our algorithm using a single function listed below and return a list of visited words to be displayed on the GUI.

```
visited_words = reinforcement_learning(learning_rate=0.001,
↪  exploration_rate=0.9,shrinkage_factor=0.9,number_of_cluster=9)
```

Listing 1: Running our RL with Clustering

From the source code, our group realised that the daily wordle goal word loops through the goal word. From our own understanding, the goal word on the 25^{th} of June 2022 is "BEADY". Therefore our AI solver is able to solve for the goal word daily by leveraging the code below to get the daily goal word.

```
reference_goal = words.index('BEADY')
date_diff = (date.today() - date(2022,6,25)).days
CORRECT_WORD = words[reference_goal + date_diff].lower()
```

Listing 2: Getting the daily wordle

5.2.3 PyGame Demonstration

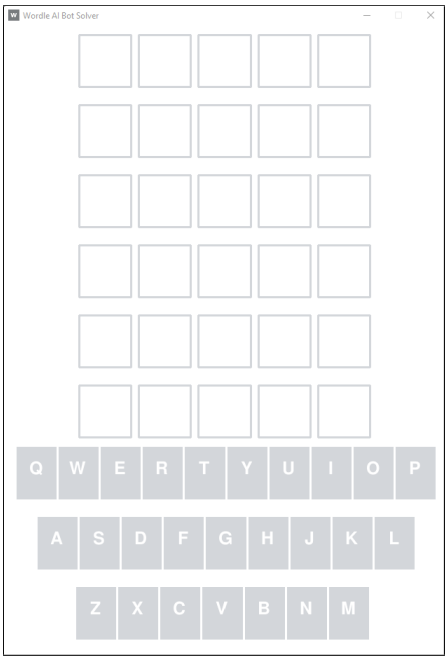

Figure 13: Pygame starting page

The starting page takes around a second to initialize, and the player just needs to press **enter** to begin.

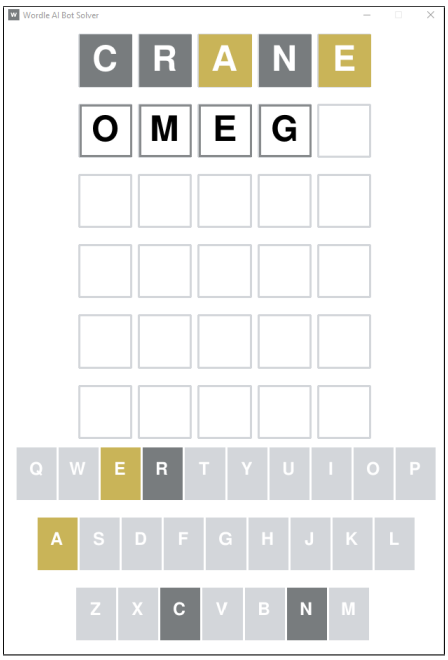

Figure 14: Pygame game in progress

For example, the above image is after the player has pressed **enter** multiple times.

The letter box colour will change accordingly similar to the actual “Wordle” game.

Figure 15: Pygame one run

As the player keeps pressing **enter**, the UI will keep updating until it eventually reaches the goal word. As seen in the image above, the goal word of the day is “STEAD”. Additionally, should our solver be unsuccessful or the player would like to re-run another iteration, the player can press **ESC**.

Figure 16: Pygame another run

For example, the above image is a re-run of the game which results in a different list of guesses to reach the daily wordle.

6 Conclusion

6.1 Improvements

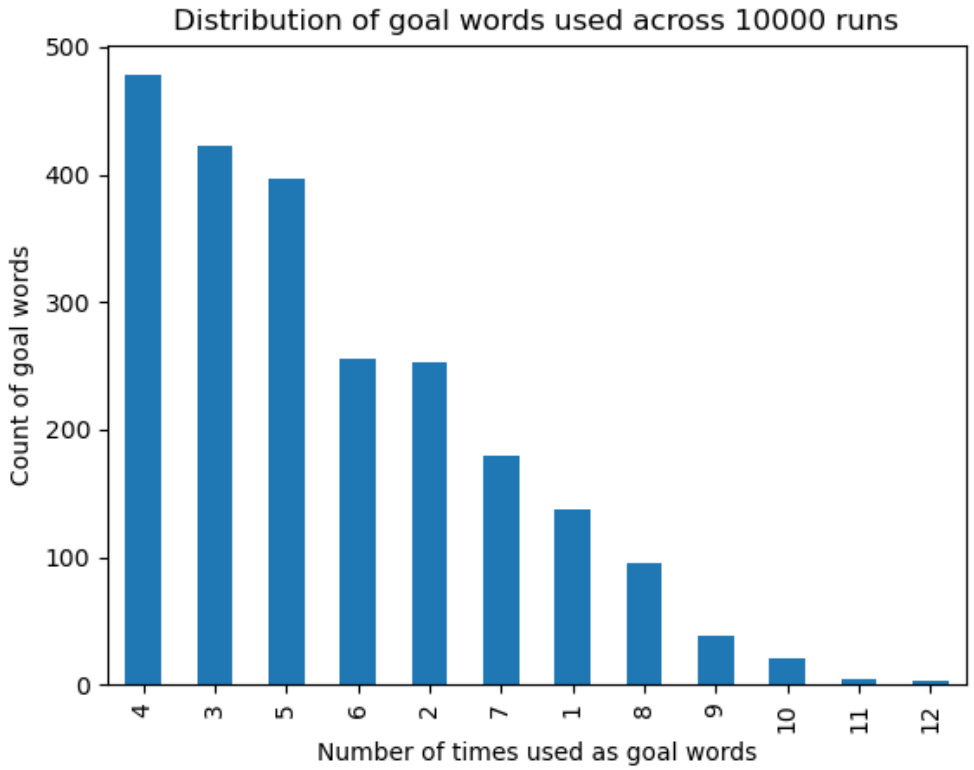

Figure 17: Bar-plot of count of goal words used across 10000 runs

From our results, as mentioned earlier, it seems that our agent does not learn much from the Q-table iterations, as the performance of the reinforcement learning model does not improve over time in terms of the average guesses. The reason might be that we might not be running enough game runs. Since the entire goal word list consists of 2309 words and our approach selects a random word from it. It might be the case that some words are not equally chosen, causing imbalanced learning for our agent. As seen from the bar plot above, the selection of goal words from the list of goal words does not follow a uniform distribution. Therefore, one training epoch

could be running over the entire list of 2309 goal words. However, this would be very costly in terms of time taken.

A better improvement would be instead targeting how our agent selects a word from the chosen cluster. As mentioned before, under its limitations, our reinforcement learning with clustering model selects a random word within the chosen cluster. Notably, another git repository[5] had attempted to discretize the distribution of words among the clusters. Their approach achieved a higher win-rate of 99%, however, at the cost of a much higher run time of 2.7 seconds per run.

6.2 Improvement Using Deep-Q Learning

One other possible suggestion that might improve performance would be to use Deep-Q Learning. For the purposes and the limited time constraints for the project this was not attempted. Deep-Q Learning operates similar to Vanilla Q-Learning, but instead of updating the Q-table, we train it using a Neural Network. The inputs would correspond to current states, and the actions are the outputs of the Neural Network. A Neural Network might learn the relationship better than decisions based on a Q-table, so this might be a possible future solution to explore.

Given that our Wordle solver would have a large amount of states and a similarly large amount of actions, we would require better resources to run the training stage to produce satisfying results.

6.3 Discretization of words within clusters

Another improvement to our reinforcement learning models is to attach a probability distribution to within each cluster and apply information theory within them. Instead of clustering on Levenshtein distance to seek out the next action to take, we could do this using a probabilistic approach, similar to how information theory was used to derive the best starting word.

6.4 Other Wordle variations

Our current solution focuses on the classic 5-letter Wordle game. It is possible to expand the solution to include 4 letter, 5 letter, 6 letters and higher Wordle variation as seen in the next figure

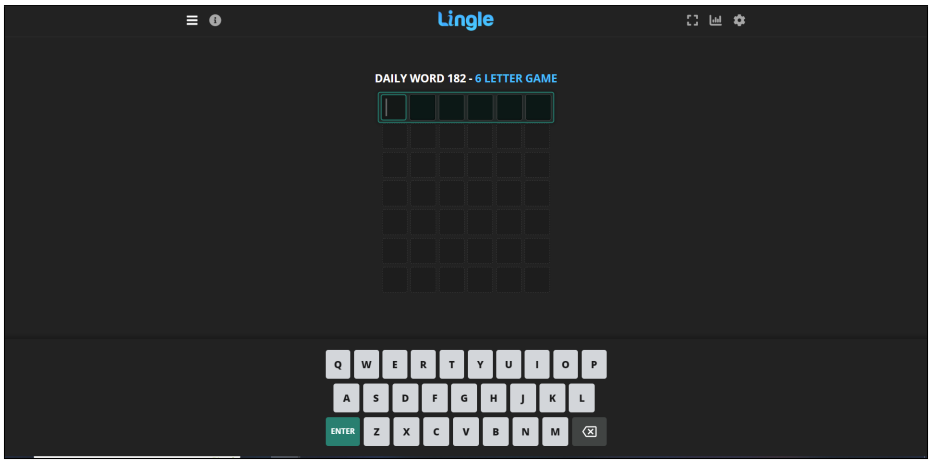

Figure 18: Example of 6-letter Wordle Variation

Our current implementation of the solution uses a corpus of only 5-letter words. It is then possible to incorporate a corpus of different word lengths on the condition that all words are of the same length for our solution to run. This can further prove our testing and evaluation results and also add diversity to the solution.

6.5 Summary of Project

Overall as a recap, for this project we decided to implement the concepts of Reinforcement Learning. Search and Clustering into the increasingly popular online word game, Wordle. We started off with gathering the data needed for training by checking the unique ID from the Wordle HTML code itself, where we split into two datasets, with the official word list consisting of 2309 goal words, and the second list consisting of 12974 goal words.

From there, we looked at several strategies ranging from a basic Reinforcement Learning model, to a variation consisting of Hierarchical Clustering, and also considered some Search Models as a comparison.

We then evaluated the models based on three metrics of: Highest Win Rate, Lowest

Average Number of Guesses and the Shortest time taken for 100 simulations. We found that with a certain configuration of hyper-parameters found by our grid-search, the Reinforcement Learning with Clustering model works the best based on these metrics.

Finally we ended off by developing a GUI to visually display our results using the Kivy python library, and also developed a mock up of the game using Pygame.

Overall we had enjoyed the project very much, considering it was a very different approach to traditional data-driven projects that was done previously. We learned how to explore more options and compile our own models and effectively improve existing solutions as well.

7 References

[1] D. Pierce, "Buying wordle brought 'tens of millions of new users' to the new york times," May 2022. [Online]. Available: https://www.theverge.com/2022/5/4/23056688/wordle-new-york-times-subscribers

[2] P. Ryan, "New york times purchases wordle." [Online]. Available: https://fssfalcon.org/5464/news/5464/#photo

[3] "Wordle - a daily word game." [Online]. Available: https://www.nytimes.com/games/wordle/index.html

[4] W. van Heeswijk, "Why discount future rewards in reinforcement learning?" Oct 2021. [Online]. Available: https://towardsdatascience.com/why-discount-future-rewards-in-reinforcement-learning-a833d0ae1942

[5] Danschauder, "Wordlebot: Using spectral clustering and reinforcement learning to win at wordle," Jan 2022. [Online]. Available: https://github.com/danschauder/wordlebot/blob/main/Wordle_Bot.ipynb

[6] Baraltech, "Baraltech/wordle-pygame: A clone of the ever-popular wordle game made in python using pygame for my youtube tutorial." [Online]. Available: https://github.com/baraltech/Wordle-PyGame

8 Appendices

8.1 Codes

The entire project can be found at this git repository:

https://github.com/Samthesimpsons/AI-Reinforcement-Learning-Project.

8.1.1 Models

Under the "models" folder, we have the 5 models in their respective python files, the saved Q-table for our pygame, as mentioned earlier under the Pygame code set-up and the two lists of feasible words and goal words.

1. **wordle_base_15k.py:** Our baseline RL model using all 15283 words

```
import re
import random
import time
import numpy as np
from tqdm import tqdm

''' List of feasible words that our reinforcement learning model will be
↪  trained on, 5-letter words from Wordle. Source:
↪  https://www.nytimes.com/games/wordle/index.html. Extracted the 2309
↪  goal words from the source code javascript file and then sorted
↪  accordingly.  Extracted the 12974 accepted words from the source code
↪  javascript file and then sorted accordingly.'''

words = []
with open('models/accepted_words.txt', 'r') as file:
    for word in file:
        words.append(word.strip('\n').upper())

goal_words = []
with open('models/goal_words.txt', 'r') as file:
    for word in file:
        goal_words.append(word.strip('\n').upper())
```

```
''' Custom Wordle class that defines the state of the wordle and the
↪  actions (and reward) that can be taken also includes getter methods for
↪  the state and the goal word. '''

class Wordle():
    def __init__(self, initial_word='CRANE'):
        self.current_word = initial_word
        self.goal_word = random.choice(goal_words)
        self.reached_goal = False

    # State is the current word itself
    def get_state(self):
        return self.current_word

    def get_goal(self):
        return self.goal_word

    # Action is picking the next word
    def make_action(self, action):
        # scoring based on yellow, green & black letters
        current_score = eval.get_score(self.current_word, self.goal_word)

        # select next word and get a new scoring
        self.current_word = action
        new_score = eval.get_score(self.current_word, self.goal_word)

        # calculate reward of previous word to new word
        reward = eval.get_reward(current_score, new_score)

        # if ever the case the goal state is reached, True is returned
        if self.current_word == self.goal_word:
            return reward, True
        return reward, False
```

```
''' Custom Evaluation class that contains the getter methods for the
↪  scoring and reward of the wordle. The scoring is based on the number of
↪  yellow, green and black letters in the wordle. The reward is based on
↪  the number of yellow (+/-5), green (+/-10) and black letters (-/+1) in
↪  the wordle. Reward 10 for each additional green letter, +5 for each
↪  additional yellow letter, penalty of -1 for each additional black
↪  letter. Includes filter function to help reduce the search space of the
↪  wordle in terms of the feasible words remaining. '''

class eval():
    def __init__(self):
        pass

    def get_score(word_1:str , word_2:str):
        scoring = {'green': 0, 'yellow': 0, 'black': 0}
        for i in range(5):
            if word_1[i] == word_2[i]:
                scoring['green'] += 1
            elif word_1[i] in word_2:
                scoring['yellow'] += 1
            else:
                scoring['black'] += 1
        return scoring

    def get_reward(new_scoring:dict, previous_score:dict):
        reward = 0
        reward += (new_scoring['green'] - previous_score['green'])*10
        reward += (new_scoring['yellow'] - previous_score['yellow'])*5
        reward -= (new_scoring['black'] - previous_score['black'])*1
        return reward

    def filter(filter_word:str, goal_word:str, corpus:list):
        black_letters = []  # list of black letters
        yellow_letters = {}  # key-val pair of yellow letters and their
        ↪  positions
        green_letters = {}  # key-val pair of green letters and their
        ↪  positions
```

```
# Get the list or dict of black letters, yellow letters and green
↪  letters
for i in range(5):
    if filter_word[i] != goal_word[i] and filter_word[i] not in
    ↪  goal_word:
        black_letters.append(filter_word[i])
    elif filter_word[i] == goal_word[i]:
        green_letters[filter_word[i]] = i
    elif filter_word[i] != goal_word[i] and filter_word[i] in
    ↪  goal_word:
        yellow_letters[filter_word[i]] = i

# Remove any words with the black letters
if len(black_letters) != 0:
    strings_to_remove = "[{}]".format("".join(black_letters))
    corpus = [word for word in corpus if (
        re.sub(strings_to_remove, '', word) == word or word ==
        ↪  filter_word)]

# Keep only words with correct green position
if len(green_letters) != 0:
    for key, value in green_letters.items():
        corpus = [word for word in corpus if (
            word[value] == key or word == filter_word)]

# Do not keep words with yellow letters in current position
if len(yellow_letters) != 0:
    for key, value in yellow_letters.items():
        corpus = [word for word in corpus if (
            word[value] != key or word == filter_word)]

# Do not keep words without yellow letters in other positions
if len(yellow_letters) != 0:
    for yellow_letter in yellow_letters.keys():
        corpus = [word for word in corpus if (
            yellow_letter in word or word == filter_word)]

# Return filtered corpus
```

```
        return corpus

''' RL function that contains the Q-learning algorithm.'''

def reinforcement_learning(learning_rate: int,
                           exploration_rate: int,
                           shrinkage_factor: int):

    epsilon = exploration_rate  # probability of exploration
    alpha = learning_rate  # learning rate
    gamma = shrinkage_factor  # discounting factor

    wordle = Wordle()
    done = False
    steps = 1  # Since we start off with an initial word already

    # initialize Q-table, goal word and the current corpus
    goal_word = wordle.get_goal()
    if goal_word == 'CRANE':
        return 1, ['CRANE']

    curr_corpus = words.copy()
    q_table = np.zeros((len(curr_corpus), len(curr_corpus)))

    visited_words = []
    while not done:
        state = wordle.get_state()
        word_to_filter_on = state
        visited_words.append(word_to_filter_on)

        # keep track of the corpus before and after filtering (cutting
        ↪  search space)
        prev_corpus = curr_corpus.copy()
        curr_corpus = eval.filter(word_to_filter_on, goal_word,
        ↪  curr_corpus)

        # Similarly, reduce the search space of the Q-table (since our
        ↪  state-action pairs are word-word pairs too)
```

```
indices_removed = []
for i, word in enumerate(prev_corpus):
    if word not in curr_corpus:
        indices_removed.append(i)

q_table = np.delete(q_table, indices_removed, axis=0)
q_table = np.delete(q_table, indices_removed, axis=1)

state_index = curr_corpus.index(state)
epsilon = epsilon / (steps ** 2) # Decaying epsilon, explore lesser
↪  as it goes on
if random.uniform(0, 1) < epsilon: # Explore
    action = random.choice(curr_corpus)
    action_index = curr_corpus.index(action)
else: # Exploit
    # Q-table is very sparse in beginning, hence if the row of
    ↪  Q-table all similar still (0), do exploration still
    if np.all(q_table[state_index][i] == q_table[state_index][0]
    ↪  for i in range(len(curr_corpus))):
        action = random.choice(curr_corpus)
        action_index = curr_corpus.index(action)
    else: # Exploit
        action_index = np.argmax(q_table[state_index])
        action = curr_corpus[action_index]

# Get reward and update Q-table
reward, done = wordle.make_action(action)
new_state = wordle.get_state()
new_state_max = np.max(q_table[curr_corpus.index(new_state)])

q_table[state_index, action_index] = (1 -
↪  alpha)*q_table[state_index, action_index] + alpha*(
    reward + gamma*new_state_max - q_table[state_index,
    ↪  action_index])

# Increment the steps
steps = steps + 1
```

```
        # Exit condition in case search too long, set currently to total
        ↪  length of initial corpus
        if steps >= len(words):
            break

    visited_words.append(goal_word)
    return steps, visited_words

''' Define a function where one simulation/run is one run of the wordle
↪  game'''

def run_simulations(learning_rate: int,
                    exploration_rate: int,
                    shrinkage_factor: int,
                    num_simulations: int):

    epochs = np.arange(num_simulations)
    guesses = np.zeros(num_simulations)
    toc = time.time()
    for epoch in tqdm(range(num_simulations)):
        steps, visited_words = reinforcement_learning(learning_rate,
        ↪  exploration_rate, shrinkage_factor)
        guesses[epoch] = steps
    tic = time.time()

    time_taken = tic - toc
    average_guesses = np.mean(guesses)
    win_rate = (num_simulations-np.sum(guesses>6))/num_simulations*100

    # print(f'Time taken: {tic - toc}')
    # print(f'Average guesses: {np.mean(guesses)}')
    # print(f'Total game losses out of {num_simulations}:
    ↪  {np.sum(guesses>6)}')
    # print(f'Overall win rate:
    ↪  {(num_simulations-np.sum(guesses>6))/num_simulations*100}%')

    return time_taken, average_guesses, win_rate, guesses
```

```
if __name__ == '__main__':
    run_simulations(learning_rate=0.9, exploration_rate=0.9,
    ↪  shrinkage_factor=0.9, num_simulations=100)
```

2. **wordle_cluster_15k.py:** Our RL with clustering model using all 15283 words

```
import re
import time
import random
import numpy as np
from tqdm import tqdm
from leven import levenshtein
from sklearn.cluster import AgglomerativeClustering

''' List of feasible words that our reinforcement learning model will be
↪  trained on, 5-letter words from Wordle. Source:
↪  https://www.nytimes.com/games/wordle/index.html. Extracted the 2309
↪  goal words from the source code javascript file and then sorted
↪  accordingly.  Extracted the 12974 accepted words from the source code
↪  javascript file and then sorted accordingly. '''

words = []
with open('models/accepted_words.txt', 'r') as file:
    for word in file:
        words.append(word.strip('\n').upper())

goal_words = []
with open('models/goal_words.txt', 'r') as file:
    for word in file:
        goal_words.append(word.strip('\n').upper())
```

```
''' Instead of the words themselves being the state of the game, and also
↪  to further reduce the search space, the idea of clustering comes into
↪  mind. In order to measure the differences between two words without the
↪  sentiment value, we can make use of the levenshtein distance or better
↪  known as the edit distance, which is really the minimum number of
↪  single-character edits required to change from one word to another. For
↪  clustering wise, instead of viewing the space of words as a vector
↪  space, since we are comparing words between each other, an hierarchical
↪  tree structure seems the most appropriate. Next consideration, is
↪  whether a top-down or bottom-up clustering approach is more feasible.
↪  Since the nodes of the tree are the
words themselves and we want to group similar words together, a bottom-up
↪  approach is more suited. Hence the choice of clustering would be to use
↪  agglomerative hierarchical clustering based on levenshtein distance
↪  measure.

Custom Clustering class that does the clustering based on the levenshtein
↪  distance measure'''

class Clustering():
    def __init__(self, number_of_clusters:int):
        self.number_of_clusters = number_of_clusters

    # Calculate the distance matrix based on the levenshtein distance
    ↪  measure
    def get_dist_matrix(self, corpus:list):
        n = len(corpus)
        distance_matrix = np.zeros((n, n))
        for i in range(n):
            for j in range(i, n):
                distance_matrix[i, j] = levenshtein(corpus[i], corpus[j])
                distance_matrix[j, i] = distance_matrix[i, j]
        return distance_matrix

    # Get the indexes of the words with the chosen cluster number
    def get_indexes_of_cluster(self, cluster_number:int, clusters:list):
        indexes = []
        for index, number in enumerate(clusters):
```

```
            if cluster_number == number:
                indexes.append(index)
        return indexes

    # Pick a random word from the chosen cluster
    def get_chosen_word(self, indexes:list, corpus:list):
        chosen_word_index = random.choice(indexes)
        return corpus[chosen_word_index]

    # Get the clusters based on the levenshtein distance measure
    def get_clusters(self, corpus:list):
        distance_matrix = self.get_dist_matrix(corpus)
        # Can do simulation analysis to test the parameters
        clusters = AgglomerativeClustering(
            n_clusters=self.number_of_clusters,
            affinity='precomputed',
            linkage='average').fit_predict(distance_matrix)
        return clusters

''' Custom Wordle class that defines the state of the wordle and the
↪  actions (and reward) that can be taken
also includes getter methods for the state and the goal word '''

class Wordle():
    def __init__(self, initial_word='CRANE'):
        self.current_word = initial_word
        self.current_state = None
        self.goal_word = random.choice(goal_words)
        self.reached_goal = False

    # State is the current cluster number itself
    def get_state(self):
        return self.current_state

    def get_curr_word(self):
        return self.current_word

    def get_goal(self):
```

```
        return self.goal_word

    # Action is the next cluster number, and then the chosen word from the
    ↪  cluster for evaluation
    def make_action(self, action, state):
        # scoring based on yellow, green & black letters
        current_score = eval.get_score(self.current_word, self.goal_word)

        # select next word and get a new scoring
        self.current_word = action
        self.current_state = state
        new_score = eval.get_score(self.current_word, self.goal_word)

        # calculate reward of previous word to new word
        reward = eval.get_reward(current_score, new_score)

        # if ever the case the goal state is reached, True is returned
        if self.current_word == self.goal_word:
            return reward, True
        return reward, False

''' Custom Evaluation class that contains the getter methods for the
↪  scoring and reward of the wordle. The scoring is based on the number of
↪  yellow, green and black letters in the wordle. The reward is based on
↪  the number of yellow (+/-5), green (+/-10) and black letters (-/+1) in
↪  the wordle. Reward 10 for each additional green letter, +5 for each
↪  additional yellow letter, penalty of -1 for each additional black
↪  letter. Includes filter function to help reduce the search space of the
↪  wordle in terms of the feasible words remaining. '''

class eval():
    def __init__(self):
        pass

    def get_score(word_1:str , word_2:str):
        scoring = {'green': 0, 'yellow': 0, 'black': 0}
        for i in range(5):
            if word_1[i] == word_2[i]:
```

```
            scoring['green'] += 1
        elif word_1[i] in word_2:
            scoring['yellow'] += 1
        else:
            scoring['black'] += 1
    return scoring

def get_reward(new_scoring:dict, previous_score:dict):
    reward = 0
    reward += (new_scoring['green'] - previous_score['green'])*10
    reward += (new_scoring['yellow'] - previous_score['yellow'])*5
    reward -= (new_scoring['black'] - previous_score['black'])*1
    return reward

def filter(filter_word:str, goal_word:str, corpus:list):
    black_letters = []  # list of black letters
    yellow_letters = {}  # key-val pair of yellow letters and their
    ↪ positions
    green_letters = {}  # key-val pair of green letters and their
    ↪ positions

    # Get the list or dict of black letters, yellow letters and green
    ↪ letters
    for i in range(5):
        if filter_word[i] != goal_word[i] and filter_word[i] not in
        ↪ goal_word:
            black_letters.append(filter_word[i])
        elif filter_word[i] == goal_word[i]:
            green_letters[filter_word[i]] = i
        elif filter_word[i] != goal_word[i] and filter_word[i] in
        ↪ goal_word:
            yellow_letters[filter_word[i]] = i

    # Remove any words with the black letters
    if len(black_letters) != 0:
        strings_to_remove = "[{}]".format("".join(black_letters))
        corpus = [word for word in corpus if (
```

```
                re.sub(strings_to_remove, '', word) == word or word ==
                ↪  filter_word)]

        # Keep only words with correct green position
        if len(green_letters) != 0:
            for key, value in green_letters.items():
                corpus = [word for word in corpus if (
                    word[value] == key or word == filter_word)]

        # Do not keep words with yellow letters in current position
        if len(yellow_letters) != 0:
            for key, value in yellow_letters.items():
                corpus = [word for word in corpus if (
                    word[value] != key or word == filter_word)]

        # Do not keep words without yellow letters in other positions
        if len(yellow_letters) != 0:
            for yellow_letter in yellow_letters.keys():
                corpus = [word for word in corpus if (
                    yellow_letter in word or word == filter_word)]

        # Unlike worle_base we can remove the word we filtering on, since
        ↪  our state-action pair is cluster-cluster and not word-word
        if filter_word in corpus:
            corpus.remove(filter_word)

        # Return filtered corpus
        return corpus

''' RL function that contains the Q-learning algorithm.'''

def reinforcement_learning(learning_rate: int,
                           exploration_rate: int,
                           shrinkage_factor: int,
                           number_of_cluster: int,
                           pairwise_distance_matrix: np.ndarray,
                           cluster_assignment: np.ndarray,
                           Q_table: np.ndarray):
```

```
epsilon = exploration_rate  # probability of exploration
alpha = learning_rate  # learning rate
gamma = shrinkage_factor  # discounting factor

wordle = Wordle()
done = False
steps = 1 # Since we start off with an initial word already

# initialize Q-table, goal word and the current corpus
goal_word = wordle.get_goal()
if goal_word == 'CRANE':
    return 1, ['CRANE']

curr_corpus = words.copy()
q_table = Q_table

# initialize distance matrix (similarities) and the clustering results
distance_matrix = pairwise_distance_matrix
cluster_results = cluster_assignment

# initialize the first word cluster numer
wordle.current_state = cluster_results[curr_corpus.index(
    wordle.get_curr_word())]

visited_words = []
while not done:
    state = wordle.get_state()
    word_to_filter_on = wordle.get_curr_word()
    visited_words.append(word_to_filter_on)

    # keep track of the corpus before and after filtering (cutting
    ↪  search space)
    prev_corpus = curr_corpus.copy()
    curr_corpus = eval.filter(word_to_filter_on, goal_word,
    ↪  curr_corpus)
```

```
# Similarly, reduce the search space of the distance_matrix and
↪  cluster_results
indices_removed = []
for i, word in enumerate(prev_corpus):
    if word not in curr_corpus:
        indices_removed.append(i)

distance_matrix = np.delete(distance_matrix, indices_removed,
↪  axis=0)
distance_matrix = np.delete(distance_matrix, indices_removed,
↪  axis=1)
cluster_results = np.delete(cluster_results, indices_removed,
↪  axis=0)

epsilon = epsilon / (steps ** 2) # Decaying epsilon, explore lesser
↪  as it goes on
if random.uniform(0, 1) < epsilon: # Explore
    list_of_states_to_explore = list(set(cluster_results))
    if len(list_of_states_to_explore) != 1:
        if state in list_of_states_to_explore:
            list_of_states_to_explore.remove(state)
    action_index = random.choice(list_of_states_to_explore)

else: #Exploit
    # Q-table is very sparse in beginning, hence if the row of
    ↪  Q-table all similar still (0), do exploration still
    if np.all(q_table[state][i] == q_table[state][0] for i in
    ↪  range(len(curr_corpus))):
        list_of_states_to_explore = list(set(cluster_results))
        if len(list_of_states_to_explore) != 1:
            if state in list_of_states_to_explore:
                list_of_states_to_explore.remove(state)
        action_index = random.choice(list_of_states_to_explore)
    else: # Exploit
        action_index = np.argmax(q_table[state])

c = Clustering(number_of_cluster)
```

```
            chosen_word =
            ↪  c.get_chosen_word(c.get_indexes_of_cluster(action_index,
            ↪  cluster_results), curr_corpus)

            # Get reward and update Q-table
            reward, done = wordle.make_action(chosen_word, action_index)
            new_state_max = np.max(q_table[action_index])

            q_table[state, action_index] = (1 - alpha)*q_table[state,
            ↪  action_index] + alpha*(
                reward + gamma*new_state_max - q_table[state, action_index])

            # Increment the steps
            steps = steps + 1

            # Exit condition in case search too long, set currently to total
            ↪  length of initial corpus
            if steps >= len(words):
                break

        visited_words.append(goal_word)
        return steps, visited_words

    ''' Define a function where one simulation/run is one run of the wordle
    ↪  game'''

    def run_simulations(learning_rate: int,
                        exploration_rate: int,
                        shrinkage_factor: int,
                        num_simulations:int,
                        number_of_cluster: int):

        epochs = np.arange(num_simulations)
        guesses = np.zeros(num_simulations)

        toc_1 = time.time()
        clust = Clustering(number_of_cluster)
        distance_matrix = clust.get_dist_matrix(words)
```

```
    cluster_results = clust.get_clusters(words)
    tic_1 = time.time()

    # Note unlike wordle_base, we are not reinitializing the Q-table each
    ↪  time,
    # instead we are going to keep updating it and learn from prev
    ↪  simulations
    Q_table = np.zeros((number_of_cluster, number_of_cluster))

    toc_2 = time.time()
    for epoch in range(num_simulations):
        steps, visited_words = reinforcement_learning(learning_rate,
                                                      exploration_rate,
                                                      shrinkage_factor,
                                                      number_of_cluster,
                                                      distance_matrix,
                                                      cluster_results,
                                                      Q_table)
        guesses[epoch] = steps
    tic_2 = time.time()

    time_taken = tic_2 - toc_1
    average_guesses = np.mean(guesses)
    win_rate = (num_simulations-np.sum(guesses>6))/num_simulations*100

    # print(f'Time for clustering: {tic_1 - toc_1}')
    # print(f'Time for learning: {tic_2 - toc_2}')
    # print(f'Average guesses: {np.mean(guesses)}')
    # print(f'Total game losses out of {num_simulations}:
    ↪  {np.sum(guesses>6)}')
    # print(f'Overall win rate:
    ↪  {(num_simulations-np.sum(guesses>6))/num_simulations*100}%')

    return time_taken, average_guesses, win_rate, guesses

if __name__ == '__main__':
    run_simulations(learning_rate=0.1, exploration_rate=0.9,
    ↪  shrinkage_factor=0.9, num_simulations=1000, number_of_cluster=10)
```

3. **wordle_cluster_2k.py:** Our RL with clustering model using only 2309 words. Only slight modification made to previous code in terms of the list of words used.

4. **wordle_greedy_search_15k.py:** Our greedy search model using all 15283 words

---

```
'''References:
https://github.com/Nk-Kyle/Wordle (this one)
https://www.mattefay.com/wordle
https://towardsdatascience.com/automatic-wordle-solving-a305954b746e
https://www.linkedin.com/pulse/solving-wordle-kohsuke-kawaguchi'''

import string
import random
import time
import numpy as np

#Edited function from interface.py
def evalGuess(guess, target):
    res = ["w" for i in range(5)]
    checkedTarget = [False for i in range(5)]
    checked = [False for i in range(5)]
    #Check for right letter and position
    for i in range(5):
        if (guess[i] == target[i]):
            res[i] = "g"
            checked[i] = True
            checkedTarget[i] = True

    #Checked for right letter but wrong position
    for i in range(5):
        if (not(checked[i])):
            for j in range(5):
                if (guess[i] == target[j] and not(checkedTarget[j])):
                    res[i] = "y"
                    checkedTarget[j] = True
                    break
```

```
    return res

#Gets number of occurence of letter in word from wordlist
def calcAlphabetScorebyOccurence(wordlist):
    alphabetVal = {}
    for word in wordlist:
        for letter in set(word):
            alphabetVal[letter] = alphabetVal.get(letter, 0) + 1
    alphabetVal = sorted(alphabetVal.items(), key=lambda x: x[1],
    ↪  reverse=True)
    return dict(alphabetVal)

#Calculate score of each words from given wordlist and alphabet score
def calcWordScorebyOccurence(wordlist):
    alphabetVal= calcAlphabetScorebyOccurence(wordlist)
    wordScore = {}
    for word in wordlist:
        for letter in set(word):
            wordScore[word] = wordScore.get(word, 0) +
            ↪  alphabetVal.get(letter, 0)
    wordScore = sorted(wordScore.items(), key=lambda x: x[1], reverse=True)
    return dict(wordScore)

#Gets matrix of letters an occurence in position
def calcAlphabetScorebyPosition(wordlist):
    alphabetVal = {}
    for alphabet in string.ascii_uppercase:
        alphabetVal[alphabet] = [0 for _ in range (5)]
    for word in wordlist:
        for i in range(len(word)):
            alphabetVal[word[i]][i] += 1
    return alphabetVal

#Gets dictionary of word and its score using position occurence
def calcWordScorebyPosition(wordlist):
    alphabetScore = calcAlphabetScorebyPosition(wordlist)
    wordScore = {}
    for word in wordlist:
```

```
        for i in range(5):
            wordScore[word] = wordScore.get(word, 0) +
            ↪  alphabetScore[word[i]][i]
    wordScore = sorted(wordScore.items(), key=lambda x: x[1], reverse=True)
    return dict(wordScore)

#Get words from wordlist
def getGoalWords():
    words = []
    with open('models/goal_words.txt', 'r') as f:
        for lines in f:
            words.append(lines.strip().upper())
    return words

def getGuessWords():
    words = []
    with open('models/accepted_words.txt', 'r') as f:
        for lines in f:
            words.append(lines.strip().upper())
    return words

def getRandomTarget(keys):
    return random.choice(keys)

def buildTree(wordlist):
    tree = {}
    for word in wordlist:
        tree[word] = {}
        for referWord in wordlist:
            if(word != referWord):
                eval = str(evalGuess(word, referWord))
                tree[word].setdefault(eval, [])
                tree[word][eval].append(referWord)
    return tree

#Decrease words from wordscores given information green (correct place)
def solveGreen(wordscores, guessWord, toEvaluate, guessResult):
    wordscores.pop(guessWord,None)
```

```
    key_list= list(wordscores)
    indices = []
    for i in range(5):
        if(guessResult[i] == "g"):
            toEvaluate[i] = guessWord[i]
            indices.append(i)
    for wordsLeft in key_list:
        for i in indices:
            if (wordsLeft[i] not in toEvaluate[i]):
                wordscores.pop(wordsLeft,None)
                break
    return wordscores, toEvaluate

def solveYellow(wordscores, guessWord, toEvaluate, guessResult):
    wordscores.pop(guessWord,None)
    key_list= list(wordscores)
    indices = []
    for i in range(5):
        if(guessResult[i] == "y"):
            toEvaluate[i]= toEvaluate[i].replace(guessWord[i], '')
            indices.append(i)
    for wordsLeft in key_list:
        for i in indices:
            if (guessWord[i] not in wordsLeft or wordsLeft[i] not in
            ↪  toEvaluate[i]):
                wordscores.pop(wordsLeft,None)
                break
    return wordscores, toEvaluate

def solveGray(wordscores, guessWord, toEvaluate, guessResult):
    wordscores.pop(guessWord,None)
    key_list= list(wordscores)
    indices = []
    for i in range(5):
        if(guessResult[i] == "w"):
            indices.append(i)

    for i in indices:
```

```
            toEvaluate[i] = toEvaluate[i].replace(guessWord[i], '')

        for wordsLeft in key_list:
            for i in indices:
                if(wordsLeft[i] not in toEvaluate[i]):
                    wordscores.pop(wordsLeft,None)
                    break
        return wordscores, toEvaluate

    def solveWithAll(wordscores,guessWord, toEvaluate, guessResult):
        wordscores,toEvaluate = solveGreen(wordscores,guessWord, toEvaluate,
        ↪  guessResult)
        wordscores,toEvaluate = solveYellow(wordscores,guessWord, toEvaluate,
        ↪  guessResult)
        wordscores,toEvaluate = solveGray(wordscores,guessWord, toEvaluate,
        ↪  guessResult)
        return wordscores, toEvaluate

    def solveTree(wordscores,guessWord,tree, guessResult):
        filter = tree[guessWord][str(guessResult)]
        for word in list(wordscores):
            if word not in filter:
                wordscores.pop(word,None)
        return wordscores

    #Edited function from interface.py
    def printGuess(guess, target):
        res = evalGuess(guess, target)
        #Print Top box lines
        top = ""
        for color in res:
            top += f" |{color}| "
        print(top)

        #Print middle box lines with letter
        mid = ""
        for i in range(5):
            mid += f" |{guess[i]}| "
```

```
    print(mid)
    print()

    return res

def checkGuess(evaluations):
    for eval in evaluations:
        if (eval != "g"):
            return False
    return True

def run_simulations(num_simulations:int):
    toc = time.time()
    guesses = np.zeros(num_simulations)
    goalwords = getGoalWords()
    guesswords = getGuessWords()

    for epoch in range(num_simulations):
        attempt = 1


        ''' Line 8 and 9 interchangable for scoring words with different
        ↪  methods'''
        wordScore2k = calcWordScorebyOccurence(goalwords)
        wordscore13k = calcWordScorebyOccurence(guesswords)
        # print(wordScore)
        # wordScore = calcWordScorebyPosition(words)

        '''Get a random word to use as a target to guess'''
        targetWord = getRandomTarget(list(wordScore2k))

        '''Preprocess'''
        toEvaluate = [string.ascii_uppercase for _ in range(5)]

        '''Start guessing'''
        guess = "CRANE"
        # printGuess(guess,targetWord)
        evaluation = evalGuess(guess,targetWord)
```

```
            while(not(checkGuess(evaluation))):
                wordScore13k,toEvaluate = solveWithAll(wordscore13k, guess,
                ↪  toEvaluate, evaluation)
                guess = list(wordScore13k)[0]
                evaluation = evalGuess(guess,targetWord)
                # printGuess(guess,targetWord)
                attempt+=1
            # print("************************")
            # print()

            guesses[epoch] = attempt
        tic = time.time()

        time_taken = tic - toc
        average_guesses = np.mean(guesses)
        win_rate = (num_simulations-np.sum(guesses>6))/num_simulations*100

        # print(f'Time taken: {time_taken}')
        # print(f'Average guesses: {average_guesses}')
        # print(f'Total game losses out of {num_simulations}:
        ↪  {np.sum(guesses>6)}')
        # print(f'Overall win rate: {win_rate}%')

        return time_taken, average_guesses, win_rate, guesses

    if __name__ == '__main__':
        run_simulations(1000)
```

5. **wordle_greedy_search_2k.py:** Our greedy search model using only 2309 words. Similarly, only slight modifications made to previous code in terms of the list of words used.

8.1.2 Analysis

Within the root directory, there is a juypter notebook, **"analysis.ipynb"** which contains all our analysis:

1. Grid search of hyper-parameters for RL models (results saved to **grid_search_results** folder)

```
from multiprocessing import Process
from models.wordle_base_15k import run_simulations as rl_base
from models.wordle_cluster_15k import run_simulations as rl_cluster
from models.wordle_cluster_2k import run_simulations as rl_cluster_2
import pandas as pd

# Our hyper-parameters to search on
learning_rates = [0.1, 0.01, 0.001]
exploration_rates = [0.5, 0.6, 0.7, 0.8, 0.9]
shrinkage_factors = [0.5, 0.6, 0.7, 0.8, 0.9]
num_of_clusters = [6, 7, 8, 9, 10]

# Set to True to rerun grid_search
run_grid_search = False

# Grid-search function for RL-base 15k
def func_1():
    df_1 = pd.DataFrame(columns=['learning_rate', 'exploration_rate',
    ↪  'shrinkage_factor', 'time_taken', 'average_guesses', 'win_rate'])

    for i, alpha in enumerate(learning_rates):
        for j, epsilon in enumerate(exploration_rates):
            for k, gamma in enumerate(shrinkage_factors):
                print(f'Running epoch {i}, {j}, {k}')
                time_taken, average_guesses, win_rate, guesses =
                ↪  rl_base(learning_rate=alpha, exploration_rate=epsilon,
                ↪  shrinkage_factor=gamma, num_simulations=100)
                df_1.loc[len(df_1)] = [alpha, epsilon, gamma, time_taken,
                ↪  average_guesses, win_rate]
```

```
    df_1.to_csv('grid_search_results/results_base.csv', index=False)
    df_1 = df_1.sort_values(by=['win_rate', 'average_guesses',
    ↪  'time_taken'], ascending=[False, True, True])
    print(df_1.iloc[0])

# Grid-search function for RL with clustering 15k
def func_2():
    df_2 = pd.DataFrame(columns=['learning_rate', 'exploration_rate',
    ↪  'shrinkage_factor', 'num_of_clusters', 'time_taken',
    ↪  'average_guesses', 'win_rate'])

    for i, alpha in enumerate(learning_rates):
        for j, epsilon in enumerate(exploration_rates):
            for k, gamma in enumerate(shrinkage_factors):
                for l, num_clusters in enumerate(num_of_clusters):
                    print(f'Running epoch {i}, {j}, {k}, {l}')
                    time_taken, average_guesses, win_rate, guesses =
                    ↪  rl_cluster(learning_rate=alpha,
                    ↪  exploration_rate=epsilon, shrinkage_factor=gamma,
                    ↪  number_of_cluster=num_clusters,
                    ↪  num_simulations=100)
                    df_2.loc[len(df_2)] = [alpha, epsilon, gamma,
                    ↪  num_clusters, time_taken, average_guesses,
                    ↪  win_rate]

    df_2.to_csv('grid_search_results/results_cluster.csv', index=False)
    df_2 = df_2.sort_values(by=['win_rate', 'average_guesses',
    ↪  'time_taken'], ascending=[False, True, True])
    print(df_2.iloc[0])

# Grid-search function for RL with clustering 2k
def func_3():
    df_3 = pd.DataFrame(columns=['learning_rate', 'exploration_rate',
    ↪  'shrinkage_factor', 'num_of_clusters', 'time_taken',
    ↪  'average_guesses', 'win_rate'])

    for i, alpha in enumerate(learning_rates):
        for j, epsilon in enumerate(exploration_rates):
```

```
            for k, gamma in enumerate(shrinkage_factors):
                for l,num_clusters in enumerate(num_of_clusters):
                    print(f'Running epoch {i}, {j}, {k}, {l}')
                    time_taken, average_guesses, win_rate, guesses =
                    ↪  rl_cluster_2(learning_rate=alpha,
                    ↪  exploration_rate=epsilon, shrinkage_factor=gamma,
                    ↪  number_of_cluster=num_clusters,
                    ↪  num_simulations=100)
                    df_3.loc[len(df_3)] = [alpha, epsilon, gamma,
                    ↪  num_clusters, time_taken, average_guesses,
                    ↪  win_rate]

    df_3.to_csv('grid_search_results/results_cluster_2.csv', index=False)
    df_3 = df_3.sort_values(by=['win_rate', 'average_guesses',
    ↪  'time_taken'], ascending=[False, True, True])
    print(df_3.iloc[0])

# Running the grid-search functions in cpu parallel
def run_cpu_tasks_in_parallel(tasks):
    running_tasks = [Process(target=task) for task in tasks]
    for running_task in running_tasks:
        running_task.start()
    for running_task in running_tasks:
        running_task.join()

if run_grid_search == True:
    run_cpu_tasks_in_parallel([func_1, func_2, func_3])

df_1 = pd.read_csv('grid_search_results/results_base.csv')
df_2 = pd.read_csv('grid_search_results/results_cluster.csv')
df_3 = pd.read_csv('grid_search_results/results_cluster_2.csv')

df_1 = df_1.sort_values(by=['win_rate', 'average_guesses', 'time_taken'],
↪  ascending=[False, True, True])
df_2 = df_2.sort_values(by=['win_rate', 'average_guesses', 'time_taken'],
↪  ascending=[False, True, True])
df_3 = df_3.sort_values(by=['win_rate', 'average_guesses', 'time_taken'],
↪  ascending=[False, True, True])
```

```
print(df_1.iloc[0])
print(df_2.iloc[0])
print(df_3.iloc[0])
```

2. Evaluation of performance of the 5 models. (results saved to **evaluation_results** folder)

```
import random
import numpy as np
import pandas as pd
import seaborn as sns
from tqdm import tqdm
from math import log10, floor
from scipy.stats import t
import matplotlib.pyplot as plt
from models.wordle_base_15k import run_simulations as rl_base
from models.wordle_cluster_15k import run_simulations as rl_cluster
from models.wordle_cluster_2k import run_simulations as rl_cluster_2
from models.wordle_greedy_search_15k import run_simulations as greed_search
from models.wordle_greedy_search_2k import run_simulations as
↪  greed_search_2

'''Running the simulations'''

# Running 10000 simulations for each of the 5 models and saving the results
run = False
if run:
    results = pd.DataFrame(columns=['time_taken', 'average_guesses',
    ↪  'win_rate'])

    for i in tqdm(range(5)):
        if i == 0:
            time_taken, average_guesses, win_rate, guesses =
            ↪  rl_base(learning_rate=0.1, exploration_rate=0.8,
            ↪  shrinkage_factor=0.8, num_simulations=10000)
            base_guesses = guesses
            results.loc[0] = [time_taken, average_guesses, win_rate]
```

```
        elif i == 1:
            time_taken, average_guesses, win_rate, guesses =
            ↪ rl_cluster(learning_rate=0.1, exploration_rate=0.5,
            ↪ number_of_cluster=6 ,shrinkage_factor=0.9,
            ↪ num_simulations=10000)
            cluster_guesses = guesses
            results.loc[1] = [time_taken, average_guesses, win_rate]
        elif i == 2:
            time_taken, average_guesses, win_rate, guesses =
            ↪ rl_cluster_2(learning_rate=0.001, exploration_rate=0.9,
            ↪ number_of_cluster=9, shrinkage_factor=0.9,
            ↪ num_simulations=10000)
            cluster_2_guesses = guesses
            results.loc[2] = [time_taken, average_guesses, win_rate]
        elif i == 3:
            time_taken, average_guesses, win_rate, guesses =
            ↪ greed_search(num_simulations=10000)
            greedy_search_guesses = guesses
            results.loc[3] = [time_taken, average_guesses, win_rate]
        elif i == 4:
            time_taken, average_guesses, win_rate, guesses =
            ↪ greed_search_2(num_simulations=10000)
            greedy_search_2_guesses = guesses
            results.loc[4] = [time_taken, average_guesses, win_rate]

    results.to_csv('evaluation_results/results.csv')
    np.save('evaluation_results/base_guesses.npy', base_guesses)
    np.save('evaluation_results/cluster_guesses.npy', cluster_guesses)
    np.save('evaluation_results/cluster_2_guesses.npy', cluster_2_guesses)
    np.save('evaluation_results/greedy_search_guesses.npy',
    ↪ greedy_search_guesses)
    np.save('evaluation_results/greedy_search_2_guesses.npy',
    ↪ greedy_search_2_guesses)

# Load the results
results = pd.read_csv('evaluation_results/results.csv')
base_guesses = np.load('evaluation_results/base_guesses.npy')
cluster_guesses = np.load('evaluation_results/cluster_guesses.npy')
```

```
cluster_2_guesses = np.load('evaluation_results/cluster_2_guesses.npy')
greedy_search_guesses =
↪  np.load('evaluation_results/greedy_search_guesses.npy')
greedy_search_2_guesses =
↪  np.load('evaluation_results/greedy_search_2_guesses.npy')

# Plotting the guesses over time to check for burn-in period and to check
↪  if the Q-learning reduces the guesses.
epochs = np.arange(10000)
fig, ax = plt.subplots(3, 2, figsize=(10, 10))
ax[0, 0].bar(epochs, base_guesses, color='blue')
ax[0, 0].set_title('RL Base 15k Words')
ax[1, 0].bar(epochs, cluster_guesses, color='blue')
ax[1, 0].set_title('RL with Cluster 15k Words')
ax[1, 1].bar(epochs, cluster_2_guesses, color='blue')
ax[1, 1].set_title('RL with Cluster 2k Words')
ax[2, 0].bar(epochs, greedy_search_guesses, color='blue')
ax[2, 0].set_title('Greedy Search 15k Words')
ax[2, 1].bar(epochs, greedy_search_2_guesses, color='blue')
ax[2, 1].set_title('Greedy Search 2k Words')
ax[0, 1].set_axis_off()
fig.supxlabel('Run Index')
fig.supylabel('Number of Guesses required to get goal word')

'''Batch-means approach on average guesses'''

# Batch-averaging calculations with burn-in period of 1000
def batch_calc(guesses:np.ndarray):
    guesses_mean = []
    for i in range(0, len(guesses), 1000):
        guesses_mean.append(np.mean(guesses[i:i+1000]))
    guesses_mean = np.array(guesses_mean)
    guesses_mean = guesses_mean[1:]
    return guesses_mean

# Round to nearest 3sf
def round_sig(x, sig=3):
    return round(x, sig-int(floor(log10(abs(x))))-1)
```

```
# Find the confidence interval for the batch-averaged guesses
def calculate_t_test_CI(values, confidence:int):
    dof = len(values) - 1
    conf = confidence
    t_crit = np.abs(t.ppf((1-confidence)/2,dof))
    mean = np.mean(values)
    variance = np.var(values)
    return (round((mean - t_crit * (variance / np.sqrt(dof))),3),
    ↪  round((mean + t_crit * (variance / np.sqrt(dof))),3))

# Do the batch-averaging calculations
base_avg_guesses = batch_calc(base_guesses)
cluster_avg_guesses = batch_calc(cluster_guesses)
cluster_2_avg_guesses = batch_calc(cluster_2_guesses)
greedy_search_avg_guesses = batch_calc(greedy_search_guesses)
greedy_search_2_avg_guesses = batch_calc(greedy_search_2_guesses)

# Plot the batch-averaged guesses
sns.lineplot(x=range(len(base_avg_guesses)), y=base_avg_guesses,
↪  marker='o', markersize=5, linewidth=1, label='RL base with 15k words')
sns.lineplot(x=range(len(cluster_avg_guesses)), y=cluster_avg_guesses,
↪  marker='o', markersize=5, linewidth=1, label='RL with cluster 15k
↪  words')
sns.lineplot(x=range(len(cluster_2_avg_guesses)), y=cluster_2_avg_guesses,
↪  marker='o', markersize=5, linewidth=1, label='RL with cluster 2k
↪  words')
sns.lineplot(x=range(len(greedy_search_avg_guesses)),
↪  y=greedy_search_avg_guesses, marker='o', markersize=5, linewidth=1,
↪  label='Greedy Search 15k words')
sns.lineplot(x=range(len(greedy_search_2_avg_guesses)),
↪  y=greedy_search_2_avg_guesses, marker='o', markersize=5, linewidth=1,
↪  label='Greedy Search 2k words')

for i in range(len(base_avg_guesses)):
    plt.text(i, base_avg_guesses[i], str(base_avg_guesses[i]),
    ↪  horizontalalignment='center', verticalalignment='bottom')
```

```
    plt.text(i, cluster_avg_guesses[i], str(cluster_avg_guesses[i]),
    ↪  horizontalalignment='center', verticalalignment='bottom')
    plt.text(i, cluster_2_avg_guesses[i], str(cluster_2_avg_guesses[i]),
    ↪  horizontalalignment='center', verticalalignment='bottom')
    plt.text(i, greedy_search_avg_guesses[i],
    ↪  str(greedy_search_avg_guesses[i]), horizontalalignment='center',
    ↪  verticalalignment='bottom')
    plt.text(i, greedy_search_2_avg_guesses[i],
    ↪  str(greedy_search_2_avg_guesses[i]), horizontalalignment='center',
    ↪  verticalalignment='bottom')

plt.title("Average guesses per 1000 simulations")
plt.xlabel("Batch number (1000 simulations per batch)")
plt.ylabel("Average guesses")
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

# Command-line printing of the confidence interval for the batch-averaged
↪  guesses, and the average guesses mean and variance
base_avg_guesses_mean = round_sig(np.mean(base_avg_guesses),3)
cluster_avg_guesses_mean = round_sig(np.mean(cluster_avg_guesses),3)
cluster_2_avg_guesses_mean = round_sig(np.mean(cluster_2_avg_guesses),3)
greedy_search_avg_guesses_mean =
↪  round_sig(np.mean(greedy_search_avg_guesses),3)
greedy_search_2_avg_guesses_mean =
↪  round_sig(np.mean(greedy_search_2_avg_guesses),3)

base_avg_guesses_variance = round_sig(np.var(base_avg_guesses),3)
cluster_avg_guesses_variance = round_sig(np.var(cluster_avg_guesses),3)
cluster_2_avg_guesses_variance = round_sig(np.var(cluster_2_avg_guesses),3)
greedy_search_avg_guesses_variance =
↪  round_sig(np.var(greedy_search_avg_guesses),3)
greedy_search_2_avg_guesses_variance =
↪  round_sig(np.var(greedy_search_2_avg_guesses),3)

base_avg_guesses_CI = calculate_t_test_CI(base_avg_guesses, 0.95)
cluster_avg_guesses_CI = calculate_t_test_CI(cluster_avg_guesses, 0.95)
cluster_2_avg_guesses_CI = calculate_t_test_CI(cluster_2_avg_guesses, 0.95)
```

```
greedy_search_avg_guesses_CI =
↪  calculate_t_test_CI(greedy_search_avg_guesses, 0.95)
greedy_search_2_avg_guesses_CI =
↪  calculate_t_test_CI(greedy_search_2_avg_guesses, 0.95)

print(f'Steady state average guesses for RL base with 15k words is
↪  {base_avg_guesses_mean}, its variance is {base_avg_guesses_variance}
↪  and 95% confidence interval of the average guess is
↪  {base_avg_guesses_CI}.')
print(f'Steady state average guesses for RL with clustering & 15k words is
↪  {cluster_avg_guesses_mean}, its variance is
↪  {cluster_avg_guesses_variance} and 95% confidence interval of the
↪  average guess is {cluster_avg_guesses_CI}.')
print(f'Steady state average guesses for RL with clustering & 2k words is
↪  {cluster_2_avg_guesses_mean}, its variance is
↪  {cluster_2_avg_guesses_variance} and 95% confidence interval of the
↪  average guess is {cluster_2_avg_guesses_CI}.')
print(f'Steady state average guesses for Greedy search with 15k words is
↪  {greedy_search_avg_guesses_mean}, its variance is
↪  {greedy_search_avg_guesses_variance} and 95% confidence interval of the
↪  average guess is {greedy_search_avg_guesses_CI}.')
print(f'Steady state average guesses for Greedy search with 2k words is
↪  {greedy_search_2_avg_guesses_mean}, its variance is
↪  {greedy_search_2_avg_guesses_variance} and 95% confidence interval of
↪  the average guess is {greedy_search_2_avg_guesses_CI}.')

'''Batch-means approach on average win-rate'''
# Get the winrate for each run
def calculate_win_rate(batch_len:int, guesses):
    win_rate = (batch_len-np.sum(guesses>6))/batch_len*100
    return win_rate

# Batch-averaging calculations with burn-in period of 1000
def batch_calc(guesses:np.ndarray):
    win_rate = []
    for i in range(0, len(guesses), 1000):
        win_rate.append(calculate_win_rate(1000,guesses[i:i+1000]))
    win_rate = np.array(win_rate)
```

```
    win_rate = win_rate[1:]
    return win_rate

# Round to nearest 3sf
def round_sig(x, sig=3):
    return round(x, sig-int(floor(log10(abs(x))))-1)

# Find the confidence interval for the batch-averaged guesses
def calculate_t_test_CI(values, confidence:int):
    dof = len(values) - 1
    conf = confidence
    t_crit = np.abs(t.ppf((1-confidence)/2,dof))
    mean = np.mean(values)
    variance = np.var(values)
    return (round((mean - t_crit * (variance / np.sqrt(dof))),1),
    ↪  round((mean + t_crit * (variance / np.sqrt(dof))),1))

# Do the batch-averaging calculations
base_avg_win_rates = batch_calc(base_guesses)
cluster_avg_win_rates = batch_calc(cluster_guesses)
cluster_2_avg_win_rates = batch_calc(cluster_2_guesses)
greedy_search_avg_win_rates = batch_calc(greedy_search_guesses)
greedy_search_2_avg_win_rates = batch_calc(greedy_search_2_guesses)

# Plot the winrate for each batch
sns.lineplot(x=range(len(base_avg_win_rates)), y=base_avg_win_rates,
↪  marker='o', markersize=5, linewidth=1, label='RL base with 15k words')
sns.lineplot(x=range(len(cluster_avg_win_rates)), y=cluster_avg_win_rates,
↪  marker='o', markersize=5, linewidth=1, label='RL with clustering & 15k
↪  words')
sns.lineplot(x=range(len(cluster_2_avg_win_rates)),
↪  y=cluster_2_avg_win_rates, marker='o', markersize=5, linewidth=1,
↪  label='RL with clustering & 2k words')
sns.lineplot(x=range(len(greedy_search_avg_win_rates)),
↪  y=greedy_search_avg_win_rates, marker='o', markersize=5, linewidth=1,
↪  label='Greedy search with 15k words')
```

```
sns.lineplot(x=range(len(greedy_search_2_avg_win_rates)),
↪  y=greedy_search_2_avg_win_rates, marker='o', markersize=5, linewidth=1,
↪  label='Greedy search with 2k words')

for i in range(len(base_avg_win_rates)):
    plt.text(i, base_avg_win_rates[i], str(round(base_avg_win_rates[i],3)),
    ↪  horizontalalignment='center', verticalalignment='bottom')
    plt.text(i, cluster_avg_win_rates[i],
    ↪  str(round(cluster_avg_win_rates[i],3)),
    ↪  horizontalalignment='center', verticalalignment='bottom')
    plt.text(i, cluster_2_avg_win_rates[i],
    ↪  str(round(cluster_2_avg_win_rates[i],3)),
    ↪  horizontalalignment='center', verticalalignment='bottom')
    plt.text(i, greedy_search_avg_win_rates[i],
    ↪  str(round(greedy_search_avg_win_rates[i],3)),
    ↪  horizontalalignment='center', verticalalignment='bottom')
    plt.text(i, greedy_search_2_avg_win_rates[i],
    ↪  str(round(greedy_search_2_avg_win_rates[i],3)),
    ↪  horizontalalignment='center', verticalalignment='bottom')

plt.title("Average win-rate per 1000 simulations")
plt.xlabel("Batch number (1000 simulations per batch)")
plt.ylabel("Average win-rate")
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)

# Command-line printing of the confidence interval for the batch-averaged
↪  win-rate, and the average win-rate mean and variance
base_avg_win_rates_mean = round_sig(np.mean(base_avg_win_rates),3)
cluster_avg_win_rates_mean = round_sig(np.mean(cluster_avg_win_rates),3)
cluster_2_avg_win_rates_mean =
↪  round_sig(np.mean(cluster_2_avg_win_rates),3)
greedy_search_avg_win_rates_mean =
↪  round_sig(np.mean(greedy_search_avg_win_rates),3)
greedy_search_2_avg_win_rates_mean =
↪  round_sig(np.mean(greedy_search_2_avg_win_rates),3)

base_avg_win_rates_variance = round_sig(np.var(base_avg_win_rates),3)
cluster_avg_win_rates_variance = round_sig(np.var(cluster_avg_win_rates),3)
```

```
cluster_2_avg_win_rates_variance =
↪  round_sig(np.var(cluster_2_avg_win_rates),3)
greedy_search_avg_win_rates_variance =
↪  round_sig(np.var(greedy_search_avg_win_rates),3)
greedy_search_2_avg_win_rates_variance =
↪  round_sig(np.var(greedy_search_2_avg_win_rates),3)

base_avg_win_rates_CI = calculate_t_test_CI(base_avg_win_rates, 0.95)
cluster_avg_win_rates_CI = calculate_t_test_CI(cluster_avg_win_rates, 0.95)
cluster_2_avg_win_rates_CI = calculate_t_test_CI(cluster_2_avg_win_rates,
↪  0.95)
greedy_search_avg_win_rates_CI =
↪  calculate_t_test_CI(greedy_search_avg_win_rates, 0.95)
greedy_search_2_avg_win_rates_CI =
↪  calculate_t_test_CI(greedy_search_2_avg_win_rates, 0.95)

print(f'Steady state average guesses for RL base with 15k words is
↪  {base_avg_win_rates_mean}, its variance is
↪  {base_avg_win_rates_variance} and 95% confidence interval of the
↪  average guess is {base_avg_win_rates_CI}.')
print(f'Steady state average guesses for RL with clustering & 15k words is
↪  {cluster_avg_win_rates_mean}, its variance is
↪  {cluster_avg_win_rates_variance} and 95% confidence interval of the
↪  average guess is {cluster_avg_win_rates_CI}.')
print(f'Steady state average guesses for RL with clustering & 2k words is
↪  {cluster_2_avg_win_rates_mean}, its variance is
↪  {cluster_2_avg_win_rates_variance} and 95% confidence interval of the
↪  average guess is {cluster_2_avg_win_rates_CI}.')
print(f'Steady state average guesses for Greedy search with 15k words is
↪  {greedy_search_avg_win_rates_mean}, its variance is
↪  {greedy_search_avg_win_rates_variance} and 95% confidence interval of
↪  the average guess is {greedy_search_avg_win_rates_CI}.')
print(f'Steady state average guesses for Greedy search with 2k words is
↪  {greedy_search_2_avg_win_rates_mean}, its variance is
↪  {greedy_search_2_avg_win_rates_variance} and 95% confidence interval of
↪  the average guess is {greedy_search_2_avg_win_rates_CI}.')

'''To see the time taken over the 10000 runs'''
```

```
results['time_taken']
```

3. Future work - Imbalanced training

```
goal_words = []
with open('models/goal_words.txt', 'r') as file:
    for word in file:
        goal_words.append(word.strip('\n').upper())

# Simulate our random selection of goal word for the 10000 runs
words = {}
for i in range(10000):
    word = goal_words[int(np.random.uniform(0,2308))]
    if word in words:
        words[word] += 1
    else:
        words[word] = 1

# Do a count of the number of times each word was selected and plot
df = pd.DataFrame(words.items(), columns=['word', 'count'])
df.sort_values(by='count', ascending=False, inplace=True)
df = df['count'].value_counts()

df.plot(kind='bar')
plt.title("Distribution of goal words used across 10000 runs")
plt.xlabel("Number of times used as goal words")
plt.ylabel("Count of goal words")
```

8.1.3 Graphical User Interfaces (GUIs)

Within the root directory, there exist two python files of our GUIs:

- **"wordle_performance_kivy.py"**

```
from pyexpat import model
from kivy.app import App
from kivy.uix.label import Label
from kivy.uix.togglebutton import ToggleButton
```

```
from kivy.uix.button import Button
from kivy.uix.gridlayout import GridLayout
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.slider import Slider
from kivy.uix.popup import Popup
from kivy.metrics import *
from models.wordle_base_15k import run_simulations as rl_base
from models.wordle_cluster_2k import run_simulations as rl_cluster_1
from models.wordle_cluster_15k import run_simulations as rl_cluster_2
from models.wordle_greedy_search_2k import run_simulations as rl_greedy_1
from models.wordle_greedy_search_15k import run_simulations as rl_greedy_2
import numpy as np
import matplotlib.pyplot as plt
from GUI_files.complexRadar import ComplexRadar # Code taken online
from kivy.garden.matplotlib import FigureCanvasKivyAgg

class MainApp(App):

    def build(self):
        # initialize some variables
        self.state = 0
        self.current_words_on_display = []
        self.currently_displayed = {
                                    "RL Base": [],
                                    "RL Cluster 2k": [],
                                    "RL Cluster 15k": [] ,
                                    "Greedy Search 2k": [],
                                    "Greedy Search 15k": []
                                    }
        self.state_dict = {
            1 : "RL Base",
            2 : "RL Cluster 2k",
            3 : "RL Cluster 15k",
            4 : "Greedy Search 2k",
            5 : "Greedy Search 15k"
        }
        self.max_time = 0
        # best parameters from grid search
```

```
# [learning_rate, exploration_rate, shrinkage_factor, num_clusters]
self.best_params = {
    1 : [0.1, 0.8, 0.8, 10],
    2 : [0.1, 0.9, 0.5, 6],
    3 : [0.001, 0.9, 0.9, 9],
    4 : [0.9, 0.9, 0.9, 10],
    5 : [0.9, 0.9, 0.9, 10]
}


# main widget (root)
# self.root = GridLayout(cols=4, rows=3)
# self.root.add_widget(BoxLayout(orientation='vertical'))
self.root = BoxLayout(orientation='vertical')

# header row to welcome the player
header_row = BoxLayout(orientation = "vertical")
intro_text = Label(text='Welcome to [color=3333ff]Wordle[/color]
↪  solver results page',markup=True,font_size='20sp')
header_row.add_widget(intro_text)
self.root.add_widget(header_row)

# buttons row which contains the togglebuttons indicating the
↪  selected mode
button_row = BoxLayout(orientation = "horizontal", spacing=2)
btn1 = ToggleButton(text='RL Base model', group='mode')
btn1.id = 1
btn1.bind(on_press = self.select_mode)

btn2 = ToggleButton(text='RL Clustering 2k', group='mode')
btn2.id = 2
btn2.bind(on_press= self.select_mode)

btn3 = ToggleButton(text='RL Clustering 15k', group='mode')
btn3.id = 3
btn3.bind(on_press = self.select_mode)

btn4 = ToggleButton(text='Greedy Search 2k', group='mode')
```

```
btn4.id = 4
btn4.bind(on_press = self.select_mode)

btn5 = ToggleButton(text='Greedy Search 15k', group='mode')
btn5.id = 5
btn5.bind(on_press = self.select_mode)

button_row.add_widget(btn1)
button_row.add_widget(btn2)
button_row.add_widget(btn3)
button_row.add_widget(btn4)
button_row.add_widget(btn5)
self.root.add_widget(button_row)

# row to select parameters
parameter_row = GridLayout(rows=3, cols=2)

# Cell 0,0 : learning rate
self.box1 = BoxLayout(orientation="horizontal")
self.learning_rate = 0.9
self.learning_rate_text = Label(text=f'learning rate :
↪  {self.learning_rate}')
self.box1.add_widget(self.learning_rate_text)
self.learning_rate_slider =
↪  Slider(min=0.001,max=0.1,value=0.9,step=0.001)
self.learning_rate_slider.bind(
    value=self.update_learning_rate_value)
self.box1.add_widget(self.learning_rate_slider)
parameter_row.add_widget(self.box1)

# Cell 0,1 : exploration rate
self.box2 = BoxLayout(orientation="horizontal")
self.exploration_rate = 0.9
self.exploration_rate_text = Label(text=f'exploration rate :
↪  {self.exploration_rate}')
self.box2.add_widget(self.exploration_rate_text)
self.exploration_rate_slider =
↪  Slider(min=0.5,max=0.9,value=0.9,step=0.1)
```

```
self.exploration_rate_slider.bind(
    value=self.update_exploration_rate_value)
self.box2.add_widget(self.exploration_rate_slider)
parameter_row.add_widget(self.box2)

# Cell 1,0 : shrinkage factor
self.box3 = BoxLayout(orientation="horizontal")
self.shrinkage_factor = 0.9
self.shrinkage_factor_text = Label(text=f'shrinkage factor :
↪  {self.shrinkage_factor}')
self.box3.add_widget(self.shrinkage_factor_text)
self.shrinkage_factor_slider =
↪  Slider(min=0.5,max=0.9,value=0.9,step=0.1)
self.shrinkage_factor_slider.bind(
    value=self.update_shrinkage_factor_value)
self.box3.add_widget(self.shrinkage_factor_slider)
parameter_row.add_widget(self.box3)

# Cell 1,1 : num clusters
self.box4 = BoxLayout(orientation="horizontal")
self.num_clusters = 10
self.num_clusters_text = Label(text=f'num clusters :
↪  {self.num_clusters}')
self.box4.add_widget(self.num_clusters_text)
self.num_clusters_slider = Slider(min=6,max=10,value=10,step=1)
self.num_clusters_slider.bind(value=self.update_num_clusters_value)
self.box4.add_widget(self.num_clusters_slider)
parameter_row.add_widget(self.box4)

# num_sims
self.box5 = BoxLayout(orientation="horizontal")
self.num_sims = 10
self.num_sims_text = Label(text=f'num sims : {self.num_sims}')
self.box5.add_widget(self.num_sims_text)
self.num_sims_slider = Slider(min=1,max=50,value=10,step=1)
self.num_sims_slider.bind(value=self.update_num_sims_value)
self.box5.add_widget(self.num_sims_slider)
parameter_row.add_widget(self.box5)
```

```
        # run sims btn
        run_sim_row = BoxLayout(orientation='vertical')
        run_btn = Button(text = "Run model")
        run_btn.bind(on_press = self.generate_graph)
        run_btn.pos_hint = {'center_y':0.5, 'center_x':0.5}
        # run_btn.size_hint = 0.3,0.3
        run_sim_row.add_widget(run_btn)
        parameter_row.add_widget(run_sim_row)
        self.root.add_widget(parameter_row)

        return self.root

    def select_mode(self, instance):
        if instance.state == "normal":
            self.state = 0
            self.learning_rate_slider.disabled = False
            self.exploration_rate_slider.disabled = False
            self.shrinkage_factor_slider.disabled = False
            self.num_clusters_slider.disabled = False
        else:
            self.state = instance.id
            # print(self.state)

            # learning_rate defaults
            self.learning_rate = self.best_params[self.state][0]
            self.learning_rate_text.text = f"learning rate :
            ↪  {round(self.learning_rate,3)}"
            self.learning_rate_slider.value = self.learning_rate

            # exploration_rate defaults
            self.exploration_rate = self.best_params[self.state][1]
            self.exploration_rate_text.text = f"exploration rate :
            ↪  {round(self.exploration_rate,2)}"
            self.exploration_rate_slider.value = self.exploration_rate

            # shrinkage_factor defaults
            self.shrinkage_factor = self.best_params[self.state][2]
```

```
            self.shrinkage_factor_text.text = f"shrinkage factor :
            ↪  {round(self.shrinkage_factor,2)}"
            self.shrinkage_factor_slider.value = self.shrinkage_factor

            # num_clusters defaults
            self.num_clusters = self.best_params[self.state][3]
            self.num_clusters_text.text = f"num clusters :
            ↪  {round(self.num_clusters,2)}"
            self.num_clusters_slider.value = self.num_clusters

            if self.state == 4 or self.state == 5:
                self.learning_rate_slider.disabled = True
                self.exploration_rate_slider.disabled = True
                self.shrinkage_factor_slider.disabled = True
                self.num_clusters_slider.disabled = True
            else:
                self.learning_rate_slider.disabled = False
                self.exploration_rate_slider.disabled = False
                self.shrinkage_factor_slider.disabled = False
                self.num_clusters_slider.disabled = False




def update_learning_rate_value(self,instance,value):
    self.learning_rate_text.text = f"learning rate : {round(value,3)}"
    self.learning_rate = value

def update_exploration_rate_value(self,instance,value):
    self.exploration_rate_text.text = f"exploration rate :
    ↪  {round(value,2)}"
    self.exploration_rate = value

def update_shrinkage_factor_value(self,instance,value):
    self.shrinkage_factor_text.text = f"shrinkage factor :
    ↪  {round(value,2)}"
    self.shrinkage_factor = value
```

```
def update_num_clusters_value(self,instance,value):
    self.num_clusters_text.text = f"num clusters : {round(value)}"
    self.num_clusters = value

def update_num_sims_value(self,instance,value):
    self.num_sims_text.text = f"num sims : {round(value)}"
    self.num_sims = value

def generate_graph(self,instance):

    categories = ['Time Taken', 'Average guesses', 'Win Rate']
    current_model = None
    popup_content = BoxLayout(orientation = "vertical")

    if self.state == 0:
        warning_text = Label(text="No models selected")
        popup_content.add_widget(warning_text)

    else:
        # graphs
        popup_content.clear_widgets()
        if self.state == 1:
            current_model = "RL Base"
            time_taken, average_guesses, win_rate,guesses =
            ↪  rl_base(self.learning_rate,
                self.exploration_rate,
                self.shrinkage_factor,
                self.num_sims)
            epochs = np.arange(self.num_sims)
        elif self.state == 2:
            current_model = "RL Cluster 2k"
            time_taken, average_guesses, win_rate,guesses =
            ↪  rl_cluster_1(self.learning_rate,
                self.exploration_rate,
                self.shrinkage_factor,
                self.num_sims,
                self.num_clusters)
            epochs = np.arange(self.num_sims)
```

```
elif self.state == 3:
    current_model = "RL Cluster 15k"
    time_taken, average_guesses, win_rate,guesses =
    ↪  rl_cluster_2(self.learning_rate,
        self.exploration_rate,
        self.shrinkage_factor,
        self.num_sims,
        self.num_clusters)
    epochs = np.arange(self.num_sims)
elif self.state == 4:
    current_model = "Greedy Search 2k"
    time_taken, average_guesses, win_rate,guesses =
    ↪  rl_greedy_1(self.num_sims)
    epochs = np.arange(self.num_sims)
elif self.state == 5:
    current_model = "Greedy Search 15k"
    time_taken, average_guesses, win_rate,guesses =
    ↪  rl_greedy_2(self.num_sims)
    epochs = np.arange(self.num_sims)

else:
    pass

plt.figure(0) # First plot of epochs vs guesses
plt.bar(epochs,guesses,alpha=0.5,
↪  label=self.state_dict[self.state])
# plt.legend(bbox_to_anchor=(1.04,0.5), loc="upper left")
plt.legend(loc="upper right")
plt.title("Epochs vs Guesses")
plt.xlabel("Epochs")
plt.ylabel("Guesses")
popup_content.add_widget(FigureCanvasKivyAgg(plt.gcf()))


# Used to create a radar chart for all other metrics
fig1 = plt.figure(1)
#define max scale range for each axes
max_time = time_taken + 1.0
```

```
            if max_time > self.max_time:
                self.max_time = max_time
            category_range = [(0,self.max_time),(1,15),(0,100)]
            radar = ComplexRadar(fig1,categories,category_range)

            metrics = [time_taken,average_guesses,win_rate]

            # To plot multiple radar charts
            for key in self.currently_displayed.keys():
                if key == current_model:
                    self.currently_displayed[current_model] = metrics
                    if self.currently_displayed[current_model] == []:
                        self.currently_displayed[current_model] = metrics
                        radar.plot(self.currently_displayed[current_model])
                        ↪  # this plot will add multiple plots to the
                        ↪  graph
                        radar.fill(metrics,alpha=0.2)
                if self.currently_displayed[key] != []:
                    radar.plot(self.currently_displayed[key])
                    radar.fill(self.currently_displayed[key],alpha=0.2)
                print(key,self.currently_displayed[key])
            popup_content.add_widget(FigureCanvasKivyAgg(plt.gcf()))


        close_btn = Button(text='Close me!', size_hint=(0.3,0.2))
        close_btn.pos_hint = {"center_x":0.5, "center_y":0.5}
        popup_content.add_widget(close_btn)

        popup = Popup(title="Results", content=popup_content,
        ↪  auto_dismiss=False)
        close_btn.bind(on_press=popup.dismiss)

        # open the popup
        popup.open()

if __name__ == "__main__":
    MainApp().run()
```

- **“wordle_solver_pygame.py”**

```
import re
import sys
import random
import pygame
import numpy as np
from datetime import date
from leven import levenshtein
from sklearn.cluster import AgglomerativeClustering

'''Our AI wordle algorithm similar to model/wordle_cluster_2k.py but
↪  modified slightly.'''

words = []
with open('models/goal_words.txt', 'r') as file:
    for word in file:
        words.append(word.strip('\n').upper())

reference_goal = words.index('BEADY')
date_diff = (date.today() - date(2022,6,25)).days
CORRECT_WORD = words[reference_goal + date_diff].lower()

class Clustering():
    def __init__(self, number_of_clusters:int):
        self.number_of_clusters = number_of_clusters

    def get_dist_matrix(self, corpus:list):
        n = len(corpus)
        distance_matrix = np.zeros((n, n))
        for i in range(n):
            for j in range(i, n):
                distance_matrix[i, j] = levenshtein(corpus[i], corpus[j])
                distance_matrix[j, i] = distance_matrix[i, j]
        return distance_matrix

    def get_indexes_of_cluster(self, cluster_number:int, clusters:list):
        indexes = []
```

```
            for index, number in enumerate(clusters):
                if cluster_number == number:
                    indexes.append(index)
            return indexes

        def get_chosen_word(self, indexes:list, corpus:list):
            chosen_word_index = random.choice(indexes)
            return corpus[chosen_word_index]

        def get_clusters(self, corpus:list):
            distance_matrix = self.get_dist_matrix(corpus)
            clusters = AgglomerativeClustering(
                n_clusters=self.number_of_clusters,
                affinity='precomputed',
                linkage='average').fit_predict(distance_matrix)
            return clusters

    class Wordle():
        def __init__(self, initial_word='CRANE'):
            self.current_word = initial_word
            self.current_state = None
            self.goal_word = CORRECT_WORD.upper()
            self.reached_goal = False

        def get_state(self):
            return self.current_state

        def get_curr_word(self):
            return self.current_word

        def get_goal(self):
            return self.goal_word

        def make_action(self, action, state):
            current_score = eval.get_score(self.current_word, self.goal_word)

            self.current_word = action
            self.current_state = state
```

```
        new_score = eval.get_score(self.current_word, self.goal_word)
        reward = eval.get_reward(current_score, new_score)

        if self.current_word == self.goal_word:
            return reward, True
        return reward, False

class eval():
    def __init__(self):
        pass

    def get_score(word_1:str , word_2:str):
        scoring = {'green': 0, 'yellow': 0, 'black': 0}
        for i in range(5):
            if word_1[i] == word_2[i]:
                scoring['green'] += 1
            elif word_1[i] in word_2:
                scoring['yellow'] += 1
            else:
                scoring['black'] += 1
        return scoring

    def get_reward(new_scoring:dict, previous_score:dict):
        reward = 0
        reward += (new_scoring['green'] - previous_score['green'])*10
        reward += (new_scoring['yellow'] - previous_score['yellow'])*5
        reward -= (new_scoring['black'] - previous_score['black'])*1
        return reward

    def filter(filter_word:str, goal_word:str, corpus:list):
        black_letters = []
        yellow_letters = {}
        green_letters = {}

        for i in range(5):
            if filter_word[i] != goal_word[i] and filter_word[i] not in
            ↪  goal_word:
                black_letters.append(filter_word[i])
```

```
            elif filter_word[i] == goal_word[i]:
                green_letters[filter_word[i]] = i
            elif filter_word[i] != goal_word[i] and filter_word[i] in
            ↪  goal_word:
                yellow_letters[filter_word[i]] = i

        if len(black_letters) != 0:
            strings_to_remove = "[{}]".format("".join(black_letters))
            corpus = [word for word in corpus if (
                re.sub(strings_to_remove, '', word) == word or word ==
                ↪  filter_word)]

        if len(green_letters) != 0:
            for key, value in green_letters.items():
                corpus = [word for word in corpus if (
                    word[value] == key or word == filter_word)]

        if len(yellow_letters) != 0:
            for key, value in yellow_letters.items():
                corpus = [word for word in corpus if (
                    word[value] != key or word == filter_word)]

        if len(yellow_letters) != 0:
            for yellow_letter in yellow_letters.keys():
                corpus = [word for word in corpus if (
                    yellow_letter in word or word == filter_word)]

        if filter_word in corpus:
            corpus.remove(filter_word)

        return corpus

def reinforcement_learning(learning_rate: int,
                           exploration_rate: int,
                           shrinkage_factor: int,
                           number_of_cluster: int):

    epsilon = exploration_rate
```

```
alpha = learning_rate
gamma = shrinkage_factor

wordle = Wordle()
done = False
steps = 1

goal_word = wordle.get_goal()
if goal_word == 'CRANE':
    return 1, ['CRANE']

curr_corpus = words.copy()

# MODIFICATION here, to initialize the trained Q-table
q_table = np.load('models/Q_table.npy')

clust = Clustering(number_of_cluster)
distance_matrix = clust.get_dist_matrix(words)
cluster_results = clust.get_clusters(words)

wordle.current_state = cluster_results[curr_corpus.index(
    wordle.get_curr_word())]

visited_words = []
while not done:
    state = wordle.get_state()
    word_to_filter_on = wordle.get_curr_word()
    visited_words.append(word_to_filter_on)

    prev_corpus = curr_corpus.copy()
    curr_corpus = eval.filter(word_to_filter_on, goal_word,
    ↪  curr_corpus)

    indices_removed = []
    for i, word in enumerate(prev_corpus):
        if word not in curr_corpus:
            indices_removed.append(i)
```

```
distance_matrix = np.delete(distance_matrix, indices_removed,
↪  axis=0)
distance_matrix = np.delete(distance_matrix, indices_removed,
↪  axis=1)
cluster_results = np.delete(cluster_results, indices_removed,
↪  axis=0)

epsilon = epsilon / (steps ** 2)
if random.uniform(0, 1) < epsilon:
    list_of_states_to_explore = list(set(cluster_results))
    if len(list_of_states_to_explore) != 1:
        if state in list_of_states_to_explore:
            list_of_states_to_explore.remove(state)
    action_index = random.choice(list_of_states_to_explore)

else:
    if np.all(q_table[state][i] == q_table[state][0] for i in
    ↪  range(len(curr_corpus))):
        list_of_states_to_explore = list(set(cluster_results))
        if len(list_of_states_to_explore) != 1:
            if state in list_of_states_to_explore:
                list_of_states_to_explore.remove(state)
        action_index = random.choice(list_of_states_to_explore)
    else:
        action_index = np.argmax(q_table[state])

chosen_word =
↪  clust.get_chosen_word(clust.get_indexes_of_cluster(action_index,
↪  cluster_results), curr_corpus)

reward, done = wordle.make_action(chosen_word, action_index)
new_state_max = np.max(q_table[action_index])

q_table[state, action_index] = (1 - alpha)*q_table[state,
↪  action_index] + alpha*(
    reward + gamma*new_state_max - q_table[state, action_index])

steps = steps + 1
```

```
        if steps >= len(words):
            break

    visited_words.append(goal_word)
    return visited_words

'''Pygame Environment, referenced
↪  https://github.com/baraltech/Wordle-PyGame for the UI layout,
adjusted the cases and code to our AI bot solver case.'''

WORDS = [word.lower() for word in words]

pygame.init()

# Constants
WIDTH, HEIGHT = 633, 900
SCREEN = pygame.display.set_mode((WIDTH, HEIGHT),pygame.SCALED)
BACKGROUND = pygame.image.load("GUI_files/assets/starting_tiles.png")
BACKGROUND_RECT = BACKGROUND.get_rect(center=(317, 300))
SCREEN.fill("white")
SCREEN.blit(BACKGROUND, BACKGROUND_RECT)

ICON = pygame.image.load("GUI_files/assets/Icon.png")
pygame.display.set_icon(ICON)
pygame.display.set_caption("Wordle AI Bot Solver")

GREEN = "#6aaa64"
YELLOW = "#c9b458"
GREY = "#787c7e"
OUTLINE = "#d3d6da"
FILLED_OUTLINE = "#878a8c"

ALPHABET = ["QWERTYUIOP", "ASDFGHJKL", "ZXCVBNM"]
GUESSED_LETTER_FONT = pygame.font.Font("GUI_files/assets/FreeSansBold.otf",
↪  50)
AVAILABLE_LETTER_FONT =
↪  pygame.font.Font("GUI_files/assets/FreeSansBold.otf", 25)
```

```
pygame.display.update()

LETTER_X_SPACING = 85
LETTER_Y_SPACING = 12
LETTER_SIZE = 75

# Global variables
guesses_count = 0

# Guesses is a 2D list that will store guesses. A guess will be a list of
↪  letters.
# The list will be iterated through and each letter in each guess will be
↪  drawn on the screen.
guesses = [[]] * 6

current_guess = []
current_guess_string = ""
current_letter_bg_x = 110

# Indicators is a list storing all the Indicator object. An indicator is
↪  that button thing with all the letters you see.
indicators = []

# List of words from our AI solver and their letters to display
visited_words = reinforcement_learning(learning_rate=0.1,
↪  exploration_rate=0.9, shrinkage_factor=0.9, number_of_cluster=10)

letters = []
for word in visited_words:
    word_letters = list(word)
    for letter in word_letters:
        letters.append(letter)

# Keep track of the number of presses of `Enter` and the upperbound
max_presses = len(letters)
presses = 0

game_result = ""
```

```
class Letter:
    def __init__(self, text, bg_position):
        # Initializes all the variables, including text, color, position,
        ↪  size, etc.
        self.bg_color = "white"
        self.text_color = "black"
        self.bg_position = bg_position
        self.bg_x = bg_position[0]
        self.bg_y = bg_position[1]
        self.bg_rect = (bg_position[0], self.bg_y, LETTER_SIZE,
        ↪  LETTER_SIZE)
        self.text = text
        self.text_position = (self.bg_x+36, self.bg_position[1]+34)
        self.text_surface = GUESSED_LETTER_FONT.render(self.text, True,
        ↪  self.text_color)
        self.text_rect =
        ↪  self.text_surface.get_rect(center=self.text_position)

    def draw(self):
        # Puts the letter and text on the screen at the desired positions.
        pygame.draw.rect(SCREEN, self.bg_color, self.bg_rect)
        if self.bg_color == "white":
            pygame.draw.rect(SCREEN, FILLED_OUTLINE, self.bg_rect, 3)
        self.text_surface = GUESSED_LETTER_FONT.render(self.text, True,
        ↪  self.text_color)
        SCREEN.blit(self.text_surface, self.text_rect)
        pygame.display.update()

class Indicator:
    def __init__(self, x, y, letter):
        # Initializes variables such as color, size, position, and letter.
        self.x = x
        self.y = y
        self.text = letter
        self.rect = (self.x, self.y, 57, 75)
        self.bg_color = OUTLINE
```

```
    def draw(self):
        # Puts the indicator and its text on the screen at the desired
        ↪ position.
        pygame.draw.rect(SCREEN, self.bg_color, self.rect)
        self.text_surface = AVAILABLE_LETTER_FONT.render(self.text, True,
        ↪ "white")
        self.text_rect = self.text_surface.get_rect(center=(self.x+27,
        ↪ self.y+30))
        SCREEN.blit(self.text_surface, self.text_rect)
        pygame.display.update()

# Drawing the indicators on the screen.
indicator_x, indicator_y = 20, 600

for i in range(3):
    for letter in ALPHABET[i]:
        new_indicator = Indicator(indicator_x, indicator_y, letter)
        indicators.append(new_indicator)
        new_indicator.draw()
        indicator_x += 60
    indicator_y += 100
    if i == 0:
        indicator_x = 50
    elif i == 1:
        indicator_x = 105

def check_guess(guess_to_check):
    # Goes through each letter and checks if it should be green, yellow, or
    ↪ grey.
    global current_guess, current_guess_string, guesses_count,
    ↪ current_letter_bg_x, game_result
    game_decided = False
    for i in range(5):
        lowercase_letter = guess_to_check[i].text.lower()
        if lowercase_letter in CORRECT_WORD:
            if lowercase_letter == CORRECT_WORD[i]:
                guess_to_check[i].bg_color = GREEN
                for indicator in indicators:
```

```
                    if indicator.text == lowercase_letter.upper():
                        indicator.bg_color = GREEN
                        indicator.draw()
                guess_to_check[i].text_color = "white"
                if not game_decided:
                    game_result = "W"
            else:
                guess_to_check[i].bg_color = YELLOW
                for indicator in indicators:
                    if indicator.text == lowercase_letter.upper():
                        indicator.bg_color = YELLOW
                        indicator.draw()
                guess_to_check[i].text_color = "white"
                game_result = ""
                game_decided = True
        else:
            guess_to_check[i].bg_color = GREY
            for indicator in indicators:
                if indicator.text == lowercase_letter.upper():
                    indicator.bg_color = GREY
                    indicator.draw()
            guess_to_check[i].text_color = "white"
            game_result = ""
            game_decided = True
        guess_to_check[i].draw()
        pygame.display.update()

    guesses_count += 1
    current_guess = []
    current_guess_string = ""
    current_letter_bg_x = 110

    if guesses_count == 6 and game_result == "":
        game_result = "L"

def play_again():
    # Puts the play again text on the screen.
    pygame.draw.rect(SCREEN, "white", (10, 600, 1000, 600))
```

```
    play_again_font = pygame.font.Font("GUI_files/assets/FreeSansBold.otf",
    ↪  40)
    play_again_text = play_again_font.render("Press ESC to rerun!", True,
    ↪  "black")
    play_again_rect = play_again_text.get_rect(center=(WIDTH/2, 700))
    word_was_text = play_again_font.render(f"Today's wordle is
    ↪  {CORRECT_WORD.upper()}!", True, "black")
    word_was_rect = word_was_text.get_rect(center=(WIDTH/2, 650))
    SCREEN.blit(word_was_text, word_was_rect)
    SCREEN.blit(play_again_text, play_again_rect)
    pygame.display.update()

def reset():
    # Resets some global variables to their default states.
    global guesses_count, CORRECT_WORD, guesses, current_guess,
    ↪  current_guess_string, game_result, presses, letters, max_presses,
    ↪  visited_words
    SCREEN.fill("white")
    SCREEN.blit(BACKGROUND, BACKGROUND_RECT)
    guesses_count = 0
    CORRECT_WORD = CORRECT_WORD
    guesses = [[]] * 6
    current_guess = []
    current_guess_string = ""
    game_result = ""
    visited_words = reinforcement_learning(learning_rate=0.001,
    ↪  exploration_rate=0.9, shrinkage_factor=0.9, number_of_cluster=9)
    presses = 0
    letters = []
    for word in visited_words:
        word_letters = list(word)
        for letter in word_letters:
            letters.append(letter)
    max_presses = len(letters)

    pygame.display.update()
    for indicator in indicators:
        indicator.bg_color = OUTLINE
```

```
        indicator.draw()

def create_new_letter():
    # Creates a new letter and adds it to the guess.
    global current_guess_string, current_letter_bg_x
    current_guess_string += key_pressed
    new_letter = Letter(key_pressed, (current_letter_bg_x,
    ↪  guesses_count*100+LETTER_Y_SPACING))
    current_letter_bg_x += LETTER_X_SPACING
    guesses[guesses_count].append(new_letter)
    current_guess.append(new_letter)
    for guess in guesses:
        for letter in guess:
            letter.draw()

while True:
    if game_result != "":
        play_again()
    for event in pygame.event.get():
        # If quit, then exit the game e.g. alt-f4
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
        # If the user pressed a key
        if event.type == pygame.KEYDOWN:
            # If user pressed escape, reset the game
            if event.key == pygame.K_ESCAPE:
                if game_result != "":
                    reset()
            # If user pressed enter button, check the guess
            elif event.key == pygame.K_RETURN:
                if presses < max_presses and len(current_guess_string) < 5:
                        key_pressed = str(letters.pop(0))
                        presses += 1
                        create_new_letter()
                if len(current_guess_string) == 5 and
                ↪  current_guess_string.lower() in WORDS:
                    check_guess(current_guess)
```