7 Information Theory and Wordle

Lab Objective: *Use the information theory concept of entropy to create an algorithm for playing the popular word game Wordle.*

Wordle

Wordle is a popular word game[1] where you have 6 guesses to guess a five-letter word. Every time a guess is made, you receive some information about how close your guess is to the correct answer. Letters in the guess that are in the correct location are colored green; letters that are present in the secret word but not in the correct location are colored yellow; and letters that aren't present in the secret word are colored gray. An example game is given in Figure **??**.

Figure 7.1: An example game of Wordle.

The secret word is chosen at random from a fixed list of 2309 words. While it is possible to only select guesses from these words, it is not necessarily the best strategy; in many situations, there is a word that can be guessed that gives more information about what the secret word is than guessing any possible secret word would. Additionally, there is a list of 12953 words that are allowed to be used as guesses; the guess we make cannot be any arbitrary string of 5 characters, but must always must be one of these words.

[1]Specifically, it went viral on the internet in 2022.

There are a few technicalities with how the guess is evaluated when there are duplicates of a letter. If the secret word were "speak" and a guess of "bevel" was made, the first `e` would be colored yellow and the second gray. Letters that are marked green take priority over this; if "bevel" is guessed and the secret word were "ashes" instead, the first `e` in the guess will be gray and the second green. With the same guess, if the secret word were "steel", then the first `e` would be yellow and the second green. So, if there are more of a given letter in the guess than in the secret word, only as many will be marked yellow or green in the guess as there are in the secret word.

Problem 1. Write a function that accepts the secret word and a guess, and returns the colors of the guess as an array. Label correct letters with the number 2, letters in the wrong location with 1, and incorrect letters with 0. For instance, with the secret word `"pages"` and the guess `"green"`, your function should return `array([1,0,0,2,0])`.

Hint: Find some way to keep track of which letters in the secret word have been matched to. Since strings are immutable, it may also be helpful in this case to cast the guess and secret word into arrays if you need to modify them.

Problem 2. In order to efficiently implement our strategy for Wordle, we need to know what the result of each guess is for every possible secret word. We only need to make this computation once for each pair, so we will do them all at once and store them in an array.

Load the lists of possible secret words and allowed guesses from `possible_words.txt` and `allowed_words.txt`. Write a function `get_all_guess_results()` that finds the result of making a guess for each pair of secret word and allowed guess. Store the results in a 3-dimensional numpy array, where the first axis corresponds to the guess, the second to the secret word, and the third to the letter.

This computation on the whole set of words will take several minutes at the very least, so test your function on a smaller subset of the word lists. Using the first three secret words and the first two possible guesses, your function should output the following:

```
>>> get_all_guess_results(possible_words[:3], allowed_words[:2])
array([[[2, 1, 0, 0, 0],
        [2, 1, 0, 1, 0],
        [2, 1, 0, 1, 0]],

       [[2, 1, 0, 0, 0],
        [2, 1, 0, 0, 0],
        [2, 1, 0, 0, 0]]])
```

Compute the array for the full word lists. Use `np.save` to save the array to a file, to avoid needing to recompute it.

Note

Three-dimensional numpy arrays behave similarly to two-dimensional ones, and can be accessed and sliced in the same way. The only difference is that indexing only a single axis will give a two-dimensional array. We illustrate this by showing how to access some useful subsets of the final array. Here, `all_guess_results` is the output of Problem 2, `i=1388` is the index of a given guess ("boxes"), and `j=1914` is the index of a given secret word ("steel").

```
# This 2-D array is the result of the i-th guess for every secret word
>>> all_guess_results[i,:]
array([[1, 0, 0, 0, 0],
       [1, 0, 0, 1, 1],
       [1, 0, 0, 1, 0],
       ...,
       [1, 0, 0, 1, 0],
       [0, 0, 0, 1, 1],
       [0, 2, 0, 0, 0]])
# This is equivalent to all_guess_results[i] and all_guess_results[i,:,:]

# This 2-D array is the result of every guess for the j-th secret word
>>> all_guess_results[:,j]
array([[0, 0, 0, 2, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0],
       ...,
       [0, 0, 0, 0, 0],
       [0, 0, 0, 2, 1],
       [0, 0, 0, 0, 0]])

# This 1-D array is the result of the i-th guess on the j-th secret word
>>> all_guess_results[i,j]
array([0, 0, 0, 2, 1])
```

Achtung!

We will use this array frequently in the next few problems and modify it in several ways; however, be sure to keep the original array, as it will still be needed. Remember that arrays are mutable, so do not do modifications in-place on the original array. If you accidentally modify the original array, reload it from your file with `np.load`.

Our objective is to create some strategy to play Wordle as effectively as possible. Simply choosing the word that is most likely to be the secret word is completely ineffective, as there is no reason to prefer any word over another, as long as both are consistent with the information we have. A much better strategy is to maximize the amount of information each of our guesses gives us, which we will quantify by using entropy.

Information and Entropy

Entropy is the expected amount of information we would gain by knowing the result of a variable. A natural way to define the information of an event A is as $-\log_2 P(A)$.[2] The entropy of a random variable X, which we denote $H(X)$, is then defined as

$$H(X) = \mathbb{E}\left[-\log_2 P(X = x)\right] = -\sum_x P(X = x)\log_2 P(X = x),$$

where the sum version comes from the Law of the Unconscious Statistician. A loose interpretation is that if a random variable has lower entropy, then we know more about what its value will be even if we haven't observed it yet, and observing it usually will give little information. At one extreme, if a discrete random variable has zero entropy, then it is in fact necessarily constant.On the other hand, if a random variable has higher entropy, then we know less about its result and observing it typically will give more information.

For Wordle, since we don't know the secret word, it is reasonable to consider it as a random variable; this gives the secret word a value of entropy, which can be used to choose a guess that is likely to give more information. Denote the secret word as W and the result of making a guess g as $R(g)$; since we don't know the secret word, this is also a random variable. There are two approaches we can take to make a strategy out of this. First, we can think of trying to minimize the entropy of the variable $W|R(g)$. This essentially is trying to find the guess that will on average minimize how much we don't know about the secret word after we know the result of the guess. Second, we can think of trying to maximize the entropy of the variable $R(g)$. This amounts to finding which guess is expected to give the most information.

These two approaches are in fact equivalent, as

$$H(W|R(g)) = H((W, R(g))) - H(R(g)) = H(W) - H(R(g)),$$

where $H((W, R(g)))$ denotes the entropy of the joint random variable $(W, R(g))$ (*not* the cross entropy). To see this equality, note that for random variables X, Y we have

$$-\log_2 P(X|Y) = -\log_2 \frac{P(X,Y)}{P(Y)} = -\log_2 P(X,Y) + \log_2 P(Y);$$

taking the expectation of both sides implies that

$$H(X|Y) = H((X,Y)) - H(Y).$$

Additionally, the value of $R(g)$ is completely determined by W, so $H((W, R(g))) = H(W)$.

The result of all of this is that minimizing the entropy of $W|R(g)$ is equivalent to maximizing the entropy of $R(g)$, which we will use to select a good guess. Since the entropy of $R(g)$ is more straightforward to calculate, this is the approach we take for the remainder of the lab.

We now seek to calculate the entropy of $R(g)$, the result of the guess, for each guess g we can make. This is given by

$$\begin{aligned} H(R(g)) &= -\sum_r P(R(g) = r)\log_2 P(R(g) = r) \\ &= -\sum_r P(R(g, W) = r)\log_2 P(R(g, W) = r). \end{aligned}$$

[2] This choice of definition has a number of desirable properties for information: information is non-negative, the information of two independent events is the sum of their individual informations, and information is a continuous function of the probability of an event. In fact, it can be shown that such a function is the negative logarithm of the probability of an event, for some logarithm base; refer to the Volume 3 textbook for more details. The base-2 logarithm is commonly used because it can be thought of as representing the number of bits needed to encode the information.

Since we assumed a uniform distribution over the set of possible secret words, the probability $P(R(g, W) = r)$ can be calculated simply as the proportion of secret words that yield the same result r given the same guess g. We already computed the result of making each acceptable guess with each possible secret word in Problem 2. So, to find the entropy of a guess, we need only to compute the probability of each unique guess result, and then apply the equation above. This sum will need to be evaluated for each individual guess that we can make.

As an example, suppose that we know the secret word is one of the words "boney", "disco", "marsh", "stock", or "visor", and we are evaluating the guess "boxes". The result of this guess for each of these words is as follows:

Word	Guess result
boney	(2,2,0,2,0)
disco	(0,1,0,0,1)
marsh	(0,0,0,0,1)
stock	(0,1,0,0,1)
visor	(0,1,0,0,1)

There are three distinct possible results: (2,2,0,2,0), with probability $\frac{1}{5}$; (0,1,0,0,1), with probability $\frac{3}{5}$; and (0,0,0,0,1), with probability $\frac{1}{5}$. Using the above formula gives the entropy of this guess as

$$-\frac{1}{5}\log_2\frac{1}{5} - \frac{3}{5}\log_2\frac{3}{5} - \frac{1}{5}\log_2\frac{1}{5} \approx 1.3710$$

Problem 3. Write a function that accepts the multidimensional array created in Problem 2 and calculates the entropy of each guess. Return the guess with the highest entropy. Also return the index of this guess in the list of allowed guesses, to avoid needing to find it later.

In order to simplify determining the numbers of each unique guess result, we can first condense the result of each guess into a single number. A simple way to do this is interpreting the five numbers of the result as a ternary (base 3) number. For instance, we can convert the array `[1,0,2,2,1]` to the number $1 \cdot 3^0 + 0 \cdot 3^1 + 2 \cdot 3^2 + 2 \cdot 3^3 + 1 \cdot 3^4 = 154$. This step is also simple to vectorize, which makes it a relatively quick operation. Applying this to the whole array from Problem 2 will result in a 2-dimenional array, whose axes correspond to the possible secret words and the allowed guesses. Be sure not to overwrite the original array.

Hint: `np.unique` with the argument `return_counts=True` will return an array with the number of occurrences of each of the different values in a one-dimensional array. By looping over the allowed guesses, you can use this function to compute the entropy quickly. Applying this function directly to multidimensional arrays results in different behavior, however.

After we make a guess, we would like to compute the effect of knowing the result on the probability distribution of the secret word. Bayes' Rule gives

$$P(W = w|R(g) = r) = \frac{P(R(g) = r|W = w)P(W = w)}{P(R(g) = r)}.$$

First, we look at the term $P(R(g) = r|W = w)$. If we know the secret word W, then for any guess g, the result $R(g)$ is uniquely determined. Thus, this probability is either 0 or 1, depending on whether the guess result we observed is the result that would be seen if w is the secret word. For instance, with the secret word $w =$ "steel" and the guess $g =$ "boxes", the only value of r for which the probability is not zero is $r = [0, 0, 0, 2, 1]$. This is precisely the result of making that guess for that word.

Now, also note that $P(W = w)$ is a constant, and $P(R(g) = r)$ is constant for all secret words that have $P(R(g) = r|W = w) \neq 0$, since these all have the same value of $R(g)$. So, the posterior distribution is just a uniform distribution over the set of words that give the same result for our guess as we observed. Finding the optimal next guess to make is then equivalent to repeating the same process as before with a smaller initial list of possible secret words.

Problem 4. Create a function that filters the list of possible words after making a guess. Since we already computed the result of all guesses for all possible words in Problem 2, we will use this array instead of recomputing the results. Accept this array, the list of possible words, a guessed word's index in the list of allowed guesses, and the guess's result. Return a filtered list of possible words that are still possible after knowing the result of a guess. Also return a filtered version of the array of all guess results that only contains the results for the secret words still possible after making the guess. This smaller array will be used to compute the entropies for the following guess.

Hint: The most efficient way to do this problem is with *boolean masking*. If `A` is any numpy array and `mask` is a 1-D array of True/False values, then `A[mask]` will return the portion of `A` where `mask` is true. This can be used even if `A` is multidimensional, and on dimensions other than the first; for instance, `A[:,mask]` will use the mask for the second dimension of the array.

Note

Note that, while we filter down the list of possible secret words, we do not do anything similar for the list of allowed guesses. The reason for this is that, as the game goes on and we make more guesses, the list of words that could still be the secret word shrinks, while the list of words we are allowed to guess stays the same. In general, it is beneficial to allow ourselves to guess words that cannot be the secret word, because in some cases we will get more information that way.

Before we assemble our algorithm for playing Wordle, we would like a benchmark. A simple strategy to compare to is to select an allowed guess at random until we know the secret word.

Problem 5. The file `wordle.py` contains a class called `WordleGame` object that can be used to simulate games of Wordle.[a] Instantiate one of these, use the `start_game()` function to start a game, and use the `make_guess()` function to make a guess.

Write a function that accepts a `WordleGame` and starts and plays a game using the strategy of randomly selecting words. At each step:

- If we know the word, guess it; otherwise, choose a guess at random from the list of allowed guesses.
- Filter the list of possible words to only those that are still possible; this allows us to determine if we know what the secret word is
- Repeat until the secret word has been guessed

Return the number of guesses needed to guess the secret word. To visualize this algorithm, pass the argument `display=True`, and the `WordleGame` will print out each word as it is guessed.

[a] This class uses the `colorama` package to format terminal output. This package is included with the Anaconda distribution of python, but can easily be installed with `pip` if needed.

Problem 6. Write a function that accepts a `WordleGame` object and starts and plays a game using the strategy of maximizing the entropy of each guess. At each step:

- If we know the secret word, guess that word
- Otherwise, compute the entropies, and make the guess that has the highest entropy
- Filter the possible words to only those that are possible after the guess
- Repeat until the secret word has been guessed

Return the number of guesses needed to guess the secret word.

Problem 7. Write a function that accepts an integer n and simulates that many games of Wordle using each of the above algorithms. Return the average number of guesses each required to find the secret word. Compare their performance; the approach using the entropy should require about half as many guesses on average.

The `WordleGame` object also has a version you can play in the terminal, which can be started using the `play_game_interactive()` method. You can use this to also compare your own performance to that of your algorithm.