Solving Connections: Thinking Like Wyna

Assignment by Kevin Wang,
with cheerleading from Prof. Zachary Dodds,
and support from all the Scripting For All '25 team

Available on Monday, July 21

You've probably heard of it: the notorious Connections puzzle, infamously crafted by puzzle creator Wyna Liu and released daily by the New York Times. In Connections, your goal is to form four groups of four items each, where each group shares something in common. There's always exactly one solution for each puzzle, and each group is more difficult than the previous. If you haven't played this game before, try solving today's puzzle! Chances are it's not that easy. This experience of frustration will form the backbone of this assignment. >:-)

In this assignment, we will harness the power of *embeddings* to create a Connections solver! Connections and embeddings hold a special affinity for each other: Connections tests the similarity between words in a variety of manners (including, but not limited to, homophony, polysemy, association, synonymy, etc.), and the vector space of embeddings *inherently* describes *semantic similarity*!

We'll see just how well we can utilize these embeddings to best Wyna's red herrings[1] and other assortments of tricks. We'll be forming our *quartets* by scoring words based on their *cosine similarity score*. Most importantly, we'll be judging these embeddings based on a variety of metrics (that we'll design, create, and implement!) and use various scoring tools to compare our solutions against Wyna's. We hope that this assignment is an exciting and fun journey!

Assignment Logistics

Starter Files

The starter files can be found here.
In this folder, you will find:

- `connections.py`: a Python file where we'll build our solver

[1] Something that misleads or distracts

- `graphing.py`: a Python file where we'll analyze and graph our solutions
- `openai_embeddings.csv`: a `CSV` file containing the embeddings of some of the most common words in the English language; this file also contains the embeddings of words in the first ~760 puzzles
- `openai_embeddings.pkl`: a pickled file containing a Python dictionary object of our embeddings as a bytestream—don't worry about this file, just know that we're using it behind the scenes to quickly retrieve these embeddings
- `puzzles.json`: a `JSON` file containing an archive of past Connections puzzles
- `util.py`: a Python file that provides helpful utility functions, including solution retrieval and embeddings lookup
- `investigation.txt`: where you'll answer some questions in Milestone 5

> ⚠️ *Note: If you haven't already, you will need to* `pip install` *the following libraries:*
> - `pandas`
> - `seaborn`
> - `scikit-learn`

Resources

Feel free to review the slides from this week.

Getting Help—Your Anchors and Lighthouses

We're here to help you if you run into any issues along the way. Swing by our (daily!) office hours, or drop a post over on the Ed forum. Swing by!

Submission

Please only submit the files that you edited. This means that you should submit the following files:

- `connections.py`
- `graphing.py`
- `investigation.txt`

As usual, upload these files to the assignment dropbox at the regular submission site. :)

⛵ Milestone 1 - Assembling our Ship: Retrieving Embeddings

> ⚠️ *Note: There are no deliverables in Milestone 1. The purpose of this Milestone is to get you acquainted with the structure of the solver and introduce you to some utility functions (and constants!) that will prove useful in later Milestones.*

As we know, the embedding of a word (think: a list with around a thousand numbers) captures the *meaning* of the word. Courtesy of OpenAI's API, we've already assembled embeddings of the words used in the first ~760 Connections puzzles and the top 100k most common words in the English language. You will be testing your solver on these 760 puzzles; you can look at all of the puzzles in `puzzles.json`—feel free to look through and pick one that you like![2]

You will need the embeddings of the words in the puzzle you would like to solve. For example, let's say that these are your words. Note that there are 4 x 4 = 16 words in each puzzle!

```
todays_words = ["brush", "dress",  "shave", "shower", \
                "nick",  "pocket", "tidy",  "smart",  \
                "key",   "touch",  "palm",  "sharp",  \
                "pinch", "mile",   "neat",  "birth"]
```

To get the embeddings of these 16 words, use the function `get_todays_embeddings`.

```
todays_embeddings = get_todays_embeddings(todays_words)
```

This will return a `dict`ionary mapping each word to its corresponding embedding. For example,

```
print(todays_embeddings["brush"])
```

will print out

[2] Unfortunately, we only offer support for the first ~760 puzzles in the Connections archive due to the already-set-in-stone CSV file. Feel free to use your own OpenAI API key to add your own embeddings if you would like to try out a puzzle from the future that contains words not currently stored in our file.

```
[-0.01645724  0.00556806 -0.01460918 ... -0.00528129 -0.02016928
  0.01229912]
```

These `len(todays_embeddings[“brush”])` = ~3000 numbers describe the *meaning* of `"brush"`. Notice that it's a ~3000-dimensional vector! Much more *expressive* than word2vec's embeddings (which only have ~300 dimensions)—around 10x more expressive! Wow!

If you want to try an arbitrary puzzle, you can use the `retrieve_puzzles` function. Each puzzle has a special, New York Times-certified, official ID associated with it. This function accepts a path to the `JSON` file where all the puzzles are stored and returns a `dict`ionary mapping each puzzle's ID to its answers. With this function, you can look up the answer to the above puzzle with its corresponding ID number 751. For example,

```
puzzles = retrieve_puzzles(PUZZLES_PATH)
print(puzzles[751])
```

will print out

```
[['nick', 'palm', 'pinch', 'pocket'], ['brush', 'dress', 'shave', 'shower'], ['neat', 'sharp',
'smart', 'tidy'], ['birth', 'key', 'mile', 'touch']]
```

where each list in this quartet is a group. Notice that it's a list of lists (LoL)!

If you want to convert this solution into a list of words that you can then feed into the solver to generate our solution, you can use the function `convert_to_words`—think of it as a reverser. For example,

```
puzzles = retrieve_puzzles(PUZZLES_PATH)
print(convert_to_words(puzzles[751]))
```

will print out

```
['nick', 'palm', 'pinch', 'pocket', 'brush', 'dress', 'shave', 'shower', 'neat', 'sharp',
'smart', 'tidy', 'birth', 'key', 'mile', 'touch']
```

All of these functions are in `util.py`, where you can look through them and study how they work, and there are even more functions that we provide that we hope will serve you well as you attempt to solve these puzzles. We highly recommend going through this file and playing around with some of the functions. And, of course, we provide pre-defined constants that we strongly suggest you use in the solver, as it will be helpful for you (and us!) to read through your code.

💰 Milestone 2 - Searching for Treasure: Implementing a *Greedy* Heuristic

Now that we have these embeddings at our disposal, how do we get them to do something cool?

A Prelude: Cosine Similarity

How do we judge how *semantically similar* two words are? For example, if I gave you the words "king" and "queen", how "close" or "similar" would they be? At first glance, it seems that they could be opposites—but they also have "royalness" associated with each of the words! How do we quantify this? Luckily, these embeddings (think: vectors with thousands of entries) are closer to each other in this vector space if they're semantically closer to each other. Somehow, some way, the embedding model has trained these vectors to produce this behavior.

So, a smaller angle between the two vectors means that the two words are semantically closer (at least according to the embedding model)! Great! We have a scoring system! But, it's kind of awkward for a smaller number to represent a better score, so we take the cosine of that number so that closer vectors now score close to a 1, and more antiparallel vectors score closer to a 0.

You can calculate the cosine similarity of two vectors using the `cosine_similarity` function from `util.py`. Take a look at the function to see how it works in more detail.

We have a way to score the similarity of two words—reframed, two vectors—now! Make no mistake: this is *incredibly* noteworthy and remarkable! We have a way to *quantify* semantic associations between words—we're quantifying the firing of neurons inside our brain when we associate, for example, "Gojo" or "Itadori" together.[3]

Armed with this scoring metric, let's start looking for some treasure! Unfortunately, we can't brute force every single possible quartet that could be out there and look for a solution—there are over 2 million combinations! Blind guessing isn't going to cut it. Instead, we'll sacrifice some accuracy for efficiency. Your task is to implement a *greedy heuristic algorithm*. It's a problem-solving approach where we make the *most optimal* choice at each stage, with the hope of attaining an accurate, or at least close-to-accurate, solution.

In this assignment, the greedy heuristic will look something like this. Once again, let's say that these are your 16 words:

```
todays_words = ["brush", "dress",  "shave", "shower", \
                "nick",  "pocket", "tidy",  "smart",  \
                "key",   "touch",  "palm",  "sharp",  \
                "pinch", "mile",   "neat",  "birth"]
```

We'll have to have a starting point for this algorithm—let's start with `"brush"`, for example. This is what we'll colloquially call our *source*. At this step, we're going to compute all of the cosine similarities between `"brush"` and every other word. Then, we're going to take the word that has the highest cosine similarity and add it to our *group*, a list of four words. It turns out that the closest word to `"brush”` is `"shave”`! So, we add `"shave”` into this group with `"brush”`. Then, we find the word that has the highest cosine similarity score with `"brush”` *and* `"shave”`; put another way, we average the embeddings already in the group and compute the cosine similarities of the remaining words with this new "centroid" embedding, and we add the word that resulted in the highest cosine similarity score to the group. We continue this process until we have four words in a group. Note that our source word when forming a group will always be the first word in the list (for now—we'll revise this later once we can numerically compare quartets; see the next Milestone for more details on how scoring will work). After forming a group, continue this process until you have *four groups*. These four groups form a *quartet*. We will be returning this quartet as our "best solution".

[3] Haven't watched *Jujutsu Kaisen* yet? You're missing out…

Implement this algorithm in the function `find_best_quartet`. A gentle reminder that decomposition is your friend! Don't compile everything into a single function. :)

💡 Hint: Remember that lists are mutable! This leads to consequences such as this one:

```
>>> scholars = ["Walter Benjamin", "Edward Said",  "Stuart Hall"]
>>>
>>> thinkers = scholars
>>> scholars.remove("Walter Benjamin")
>>>
>>> print(thinkers)
["Edward Said",  "Stuart Hall"]
```

Notice that `"Walter Benjamin"` is missing!

To "truly copy" (i.e. *shallow copy*) a list, you can use the following (funky) syntax:

```
>>> thinkers = scholars[:]  # shallow copies `scholars`!
```

Replacing the `thinkers = scholars` line with the line above, the `print` statement will now print out:

```
>>> print(thinkers)
["Walter Benjamin", "Edward Said",  "Stuart Hall"]
```

🎯 Milestone 3 - Alas, Treasure? Scoring Solutions…

It's time to judge our solution! We'll be judging our solution based on three different metrics (the heart and soul of embeddings!):

- The ***internal score*** measures how cohesive a group of four words is. Loop through each word and calculate all the pairwise cosine similarities. Return a weighted sum of the average of these similarity scores and the minimum of the similarities (to penalize having a weak link). Put your code inside `calculate_group_internal_score`.

- The ***external score*** measures how different the words in a group are from words in other groups. Return the *negative* of the mean of all the pairwise cosine similarities of words in the group versus the words in the other groups. Put your code inside `calculate_group_external_score`.

- The ***word ambiguity score*** measures how *semantically ambiguous* a single word is in the context of a set of words. Return the maximum of all the pairwise cosine similarities between this word and the set of words. This means that if this word is highly similar to another word, it must be *semantically ambiguous* or have several different meanings attached to it. Put your code inside `calculate_word_ambiguity`.

Finally, calculate the overall or total score of this quartet in `calculate_total_score` by taking a *weighted sum* of all three scores. Note that for the word ambiguity penalty, since `calculate_word_ambiguity` only looks at a single word, take the average of all of the word ambiguity scores and make it negative—this will be your overall word ambiguity penalty. See the function signature and return statement of `calculate_total_score` for some more insight.

🗺️ Milestone 4 - Looking For More Treasure: Refining Our Search Algorithm

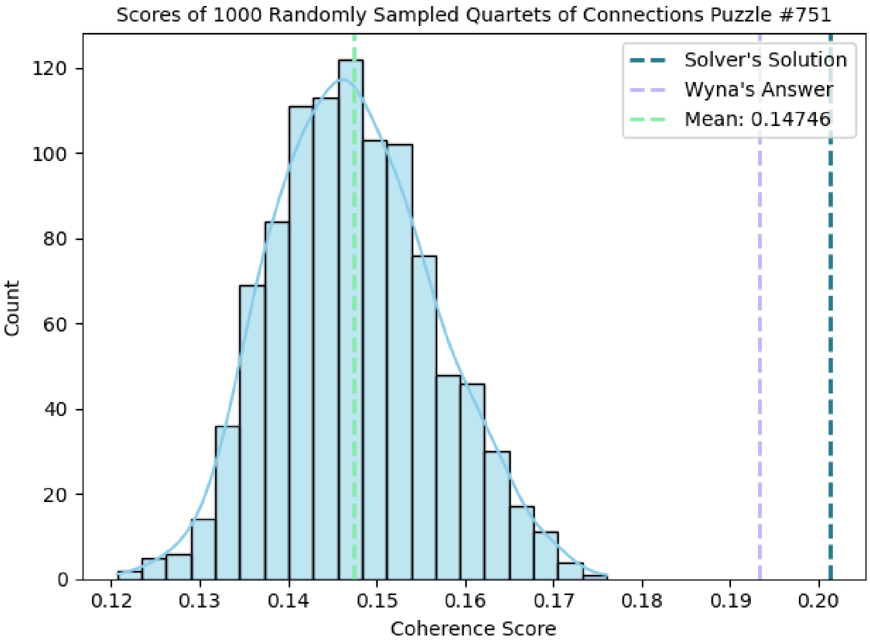
A limitation of our current search algorithm is that it relies entirely on the starting node—the word we begin with. Refine our algorithm by looping through all words in the puzzle, using each as a starting point in the search, and return the quartet that attains the best score from `calculate_total_score` in the previous Milestone. You'll only need to add a few lines to `find_best_quartet`.

🧈 Milestone 5 - Fool's Gold: Comparing Our Solutions Against Wyna's Answer

In this milestone, you'll be making two graphs: a histogram and a heatmap.

Make a histogram of the scores of randomly formed quartets from your selected puzzle. Draw vertical lines where Wyna's solution would fall and where your solution would fall. Put your code inside `graph_scores`. We've already provided the function `generate_random_quartet` for you; remember to use it—helper functions are your best friend!

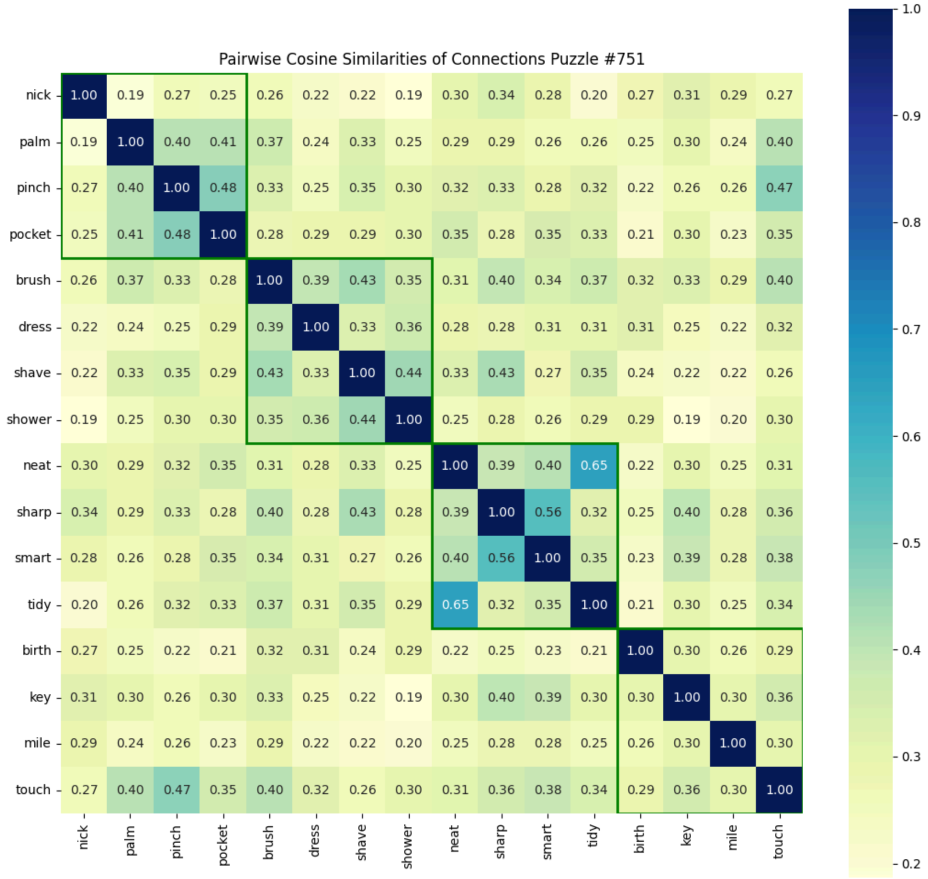
Your histogram should look something like this:



Next, make a heatmap of your favorite puzzle inside `graph_pairwise_similarities`. The result should be a 16x16 grid with each square cell denoting the cosine similarity score between the two words.

> 💡 Hint: You'll need the `heatmap` function from the `seaborn` library with the optional argument `square` set to `True`.

Your heatmap should look something like this:



Pairwise Cosine Similarities of Connections Puzzle #751

	nick	palm	pinch	pocket	brush	dress	shave	shower	neat	sharp	smart	tidy	birth	key	mile	touch
nick	1.00	0.19	0.27	0.25	0.26	0.22	0.22	0.19	0.30	0.34	0.28	0.20	0.27	0.31	0.29	0.27
palm	0.19	1.00	0.40	0.41	0.37	0.24	0.33	0.25	0.29	0.29	0.26	0.26	0.25	0.30	0.24	0.40
pinch	0.27	0.40	1.00	0.48	0.33	0.25	0.35	0.30	0.32	0.33	0.28	0.32	0.22	0.26	0.26	0.47
pocket	0.25	0.41	0.48	1.00	0.28	0.29	0.29	0.30	0.35	0.28	0.35	0.33	0.21	0.30	0.23	0.35
brush	0.26	0.37	0.33	0.28	1.00	0.39	0.43	0.35	0.31	0.40	0.34	0.37	0.32	0.33	0.29	0.40
dress	0.22	0.24	0.25	0.29	0.39	1.00	0.33	0.36	0.28	0.28	0.31	0.31	0.31	0.25	0.22	0.32
shave	0.22	0.33	0.35	0.29	0.43	0.33	1.00	0.44	0.33	0.43	0.27	0.35	0.24	0.22	0.22	0.26
shower	0.19	0.25	0.30	0.30	0.35	0.36	0.44	1.00	0.25	0.28	0.26	0.29	0.29	0.19	0.20	0.30
neat	0.30	0.29	0.32	0.35	0.31	0.28	0.33	0.25	1.00	0.39	0.40	0.65	0.22	0.30	0.25	0.31
sharp	0.34	0.29	0.33	0.28	0.40	0.28	0.43	0.28	0.39	1.00	0.56	0.32	0.25	0.40	0.28	0.36
smart	0.28	0.26	0.28	0.35	0.34	0.31	0.27	0.26	0.40	0.56	1.00	0.35	0.23	0.39	0.28	0.38
tidy	0.20	0.26	0.32	0.33	0.37	0.31	0.35	0.29	0.65	0.32	0.35	1.00	0.21	0.30	0.25	0.34
birth	0.27	0.25	0.22	0.21	0.32	0.31	0.24	0.29	0.22	0.25	0.23	0.21	1.00	0.30	0.26	0.29
key	0.31	0.30	0.26	0.30	0.33	0.25	0.22	0.19	0.30	0.40	0.39	0.30	0.30	1.00	0.30	0.36
mile	0.29	0.24	0.26	0.23	0.29	0.22	0.22	0.20	0.25	0.28	0.28	0.25	0.26	0.30	1.00	0.30
touch	0.27	0.40	0.47	0.35	0.40	0.32	0.26	0.30	0.31	0.36	0.38	0.34	0.29	0.36	0.30	1.00

🏝️ Milestone 6 - Being One With The Island: Answering Your Questions About The Solver

Now that you've traversed the island, it's time to get to know it better! Answer the following questions in `investigation.txt`:

1. Why does our scoring system work? What does it account for, and what doesn't it account for? What are some other scoring metrics you can think of? Answer in around 3-4 sentences.

2. Look at the histogram that you made in Milestone 5. What can you say about the distribution of these randomly sampled quartets? Why does our solution sometimes attain a higher score than Wyna's solution when our solution is incorrect? Answer in around 4 sentences, and more thoughts are welcome if you so desire. :)

3. In your pairwise similarity cosine graph made in Milestone 4, can you spot any red herrings for your chosen puzzle? What are some unique quirks about this graph

that you may notice? What do you see, or what don’t you see? Answer in around 5 sentences, and, of course, more thoughts are always welcome.

Now, come up with two questions about the solver that you would like to investigate and tweak in the solver. Perhaps this is how the solver scores solutions, how we search for a solution, or examining the overall accuracy of the solver—the possibilities are endless!

Jot down your two questions in `investigation.txt`, and explain how you answered these questions and what your results after tweaking reveal about the solver.

Good luck with HW8, everyone! 💫