Mega Puzzle Solver

Vesaun Shrestha, Wei Jiang, Ky Wandishin, Luke Wu

March 2024

1 Semantle AI

1.1 Summary

We applied Word2Vec vectorization of words in order to perform linear algebra techniques to solve the Semantle puzzle in very few guesses. A very brief pseudocode is as follows:

- Create a list of words 'possible_words' that reads in a large set of all English words
- Make a random guess and receive Semantle's score for that day's guess

  Let G be the guess and S be the score for explanation purposes
- Based on that guess, find all words in 'possible_words' that have a semantic similarity of S to G and update 'possible_words' to be just that array
- Now pick a random element from the refined array and repeat step 3

  This algorithm will end once the length of 'possible_words' is 1 or you have a semantic score of 110% (not 100% due to chance error handling)

1.2 Explored Ideas

1. **K-Means Clustering**

   The purpose of considering this approach was to enhance the program's efficiency by utilizing the average semantic score of clusters for comparison instead of individual nodes. This strategy aimed to streamline the search process within the word vectors. The intended workflow involved identifying the target cluster, selecting a random node within it, and iteratively refining the selection by examining neighboring nodes.

   However, we had to abandon this concept due to the excessive computational demands associated with clustering 300 million words. Even when executed on a dedicated compute server, the process was taking over an hour to complete.

2. **Vector-Translation-Stepping**

   An original approach we were developing was to obtain two words and their scores and run a linear regression on those two words and nearby words of varying range. We could then plot a line of best fit that is in the direction of the closer word and translate that vector towards the best word. We would then choose a function that decides how far down the line we need to guess a word depending on the score we received. We would use a cosine similarity function to determine whether our vector is the same as the target since we would receive a score of 100. A combination of our limited knowledge of manipulating vectors in such a way as well as time limitations to figure out the math behind this algorithm forced us to abandon this approach.

1.3 Explaining the Math

Now, the pseudocode and algorithm is simple yet the math and implementation may be a bit more involved.

1. **Word2Vec:**

Word2Vec is a group of models that create word embeddings. The models are two-layer neural networks that create a vector space with a dimensionality of several hundreds typically. For our project, our Word2Vec created a vector space of approximately 300 dimensions where each word is assigned a unique vector in this space. It's important to vectorize words in order to run computations on them to calculate distances between two vectors. This will make more sense in a second.

2. **Cosine Similarity:**

   How does Semantle calculate its score? Well, there are two ways to compute the distance between two vectors. One is the well-known Euclidean distance where we simplify square root to the sum of the squared differences of the two vectors. However, upon plotting these distances with known distances of two test vectors, we found poor correlation. Now, the cosine similarity is more relevant for our purposes. Two non-zero vectors which are proportional will result in a similarity of 1 and orthogonal vectors give a score of 0. An implementation of this in Python looks like this:

```
1: from gensim.models import KeyedVectors
2: from gensim.test.utils import common_texts
3: from gensim.models import word2vec
4: from scipy import spatial
5: import numpy as np
6: import gensim.downloader

7: model = gensim.downloader.load('word2vec-google-news-300')

8: def square_rooted(x):
9:    return round(np.sqrt(sum([a*a for a in x])),3)

10: def cosine_similarity(x,y):
11:     numerator = sum(a*b for a,b in zip(x,y))
12:     denominator = square_rooted(x)*square_rooted(y)
13:     ans = round(numerator/float(denominator),4)
14:     return (ans * 100)
```

3. **Understanding Linearity and How to Imply It:**

   We have the math we need to solve Semantle laid out and our word2vec ready to vectorize words. However, how do we solve the puzzle using these tools? Well, observing our cosine similarity function,

   The cosine similarity between vectors $\mathbf{A}$ and $\mathbf{B}$ is given by:

$$\text{cosine\_similarity}(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

   Here,

$$\text{Dot product:} \quad \mathbf{A} \cdot \mathbf{B} = \sum_{i=1}^{n} A_i \cdot B_i$$

$$\text{Euclidean norm:} \quad \|\mathbf{A}\| = \sqrt{\sum_{i=1}^{n} A_i^2} \quad \text{and} \quad \|\mathbf{B}\| = \sqrt{\sum_{i=1}^{n} B_i^2}$$

   We notice that we are taking two nonzero linear vectors into our function and returning the $cos(\theta)$ between the two, hence the name of our method. By the properties of such linear vectors in a finite vector space, the dot product between A and B will be equal to the dot product between B and A since they are commutative.

$$A \cdot B = B \cdot A$$

   $S_C(A, B) = S_C(B, A)$, since we are dividing these dot products by the product of their magnitudes.

   Now, if we take any random guess, say "apple", and get a score of 29.06 - both completely random and do not matter for our algorithm - then the target word has a similarity of 29.06 to "apple".

4. **Implementation:**

Finally, we want to implement a Python function that takes in the word, semantic score Semantle returned for that day's puzzle, and a list of possible words to consider. Initially, our possible words should be the entire English language - you need at least all the words that Semantle uses which is actually smaller than just an ordinary English dictionary but using a bigger set won't hurt you (other than computing).

Now, assuming we use the similarity functions we defined above as well as load a list of possible words that contains the English language, we can implement this loop:

```
1:while similarity != 110 and len(possible_words) != 0:
2:       current_word = np.random.choice(list(possible_words.keys()))
3:        similarity, possible_words = print_similarity_and_possible_words
(current_word, semantle_game, possible_words)
```

Let G be our guess, S be the score Semantle returned to us and P be our list of words. We will refine P from being a large list of all English words to just words that have a cosine similarity of S to G. Applying our properties of linearity mentioned in 1.3.3, our target word **must** be in this newly refined array. We can repeat this process where G_2 will be a random element from P and find words within P that have S to G_2.

After implementing this, the iterations of the loop and state of intelligence during that could look like this depending on your output:

Goal word: intron

Current word: apple
Similarity score for apple: 8.13%
Possible words: ['happen', 'west', 'surrounding', ...]

Current word: waterjet
Similarity score for waterjet: 12.22%
Possible words: ['intron']

Current word: intron
Similarity score for intron: 110%
Possible words: []

With the length of possible words being 0 as well as a similarity score of 110%, we can terminate the program.

1.4 Future goals and improvements:

Using a clustering algorithm to group similar words or preprocess with cosine similarity matrices and clustering these. Allows to check cluster to cluster distance to give a ballpark of nodes to check instead of checking every node of a large dataset.

2 New York Times Connections

Prior to Semantle, our aim was to develop an AI system capable of automatically categorizing and solving the daily NYTimes game known as Connections. This task presented significant challenges due to the abstract nature of the word categories in Connections, which heavily rely on the player's familiarity with pop culture or humor. Extracting such nuanced connections from word vectors alone proved to be a complex endeavor without the assistance of a Language Model (LLM) and extensive training datasets. Given that Connections is a relatively new game with limited word data available, our resources for training were insufficient to uncover these intricate connections (haha).

2.1 Explored Idea

1. **Word2Vec**

   In order to leverage pretrained models and existing training data for word connections, we opted to refine the pretrained GloVe 6B model using the Connections dataset.

   For the refinement process, we included each connection category as a sentence. Subsequently, we constructed the vocabulary of our new model with these sentences and proceeded with training.

   Following the training phase, we utilize the model to extract the word vectors from an unseen connections game. Subsequently, we apply k-means clustering on the cosine distances of the nodes to effectively group similar words into clusters, thereby defining the categories to which our words belong.

   The prototype code can be found within the github repository: https://github.com/kylewandishin/mega-problem-solver-backend/tree/main/connections

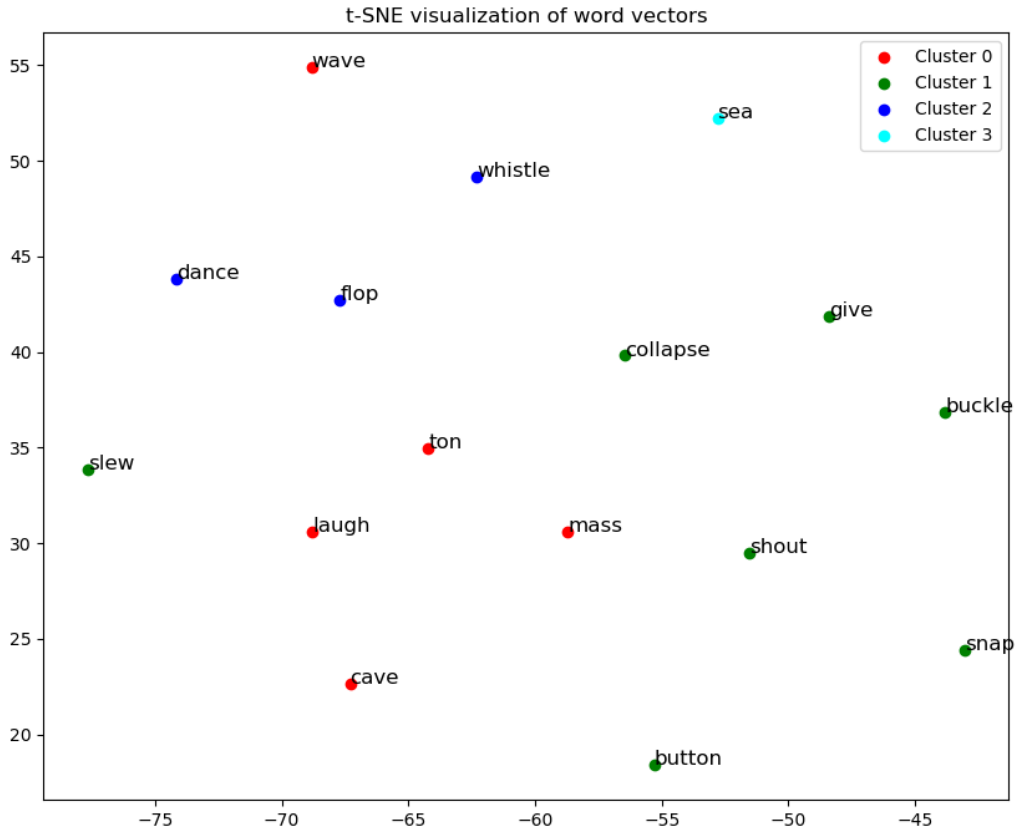

Figure 1: Visualization of word clusters after applying t-SNE and k-means clustering.

After running 400 trials, it is evident that the algorithm mostly gets two words correct per category with more being one away than 3 away. With a more sophisticated algorithm and more data, we could improve this to be more accurate.

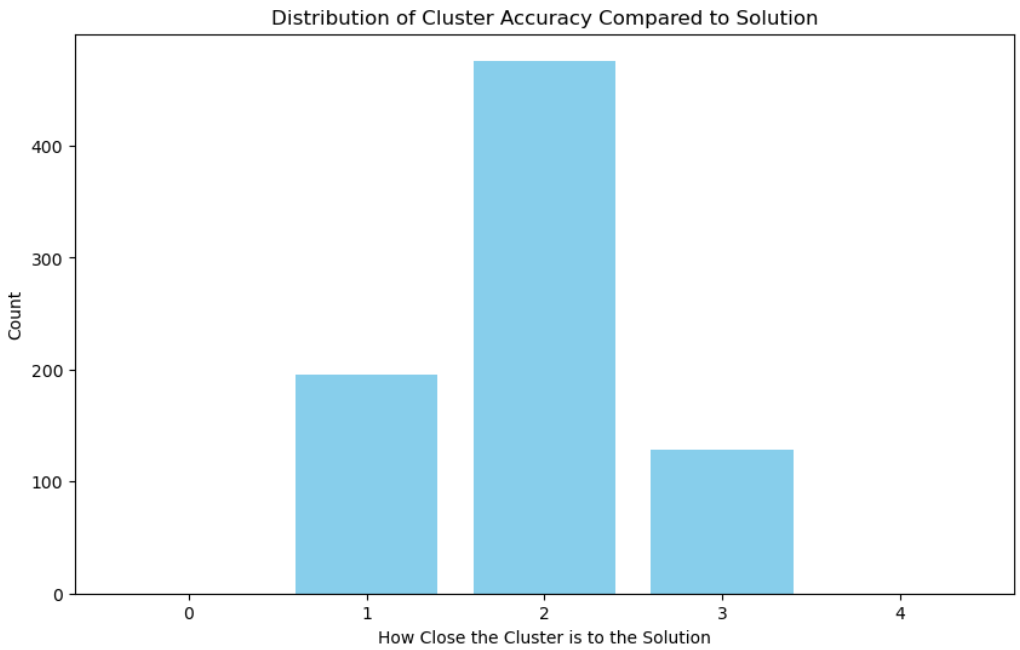

Figure 2: Distribution of accuracy after running 400 trials. The x axis represents how many words away the algorithm was from the correct answer. 1 means one word in the cluster was not in the solution