LX496 Final Project: AI Codenames

Timothy Evdokimov

December 2022

1 Code

The code for this project can be found here

https://github.com/Timevdo/codenames

This git repository should be updated if I make any changes to AI Codenames, this paper was written as of commit `0b567c4`. To run the code locally, follow the instructions in the `README` – note that the word vector file is $\sim$ 6GB and may take a while to download.

2 Background

2.1 Introduction

The purpose of this project is to solve a time-honored class of computer science problem – teaching a computer how to play a game previously only played by humans; although in this case with computational linguistics methods. Codenames (created by Vladimír Chvátil) is a great candidate for such a game, since its mechanism is very simple and also deeply rooted in linguistics. The underlying mechanism is one player coming up with a clue word that is semantically similar to a set of other words, such that another play can guess which out of a set of random words the clue refers to. This is a problem that requires understanding both how humans semantically associate words, understanding the strategy of the game, and being able to compute guesses for arbitrary inputs in real time, all of which make for a very interesting computational problem.

2.2 Rules of Codenames

The game requires four players, in two teams of at least two – a red team and a blue team. The board consists of 25 words in a 5×5 grid, each of which is assigned to belong to the red team, the blue team, or be neutral. Only one person from each team knows the affiliation of each word, that person is called the codemaster. On each turn, the codemaster from one of the team says a clue

in the form of a word and a number, referring to that number of words on the board (e.g. "Tennis, 2" referring to "Racket" and "Swing"). The codemaster's teammates then attempt to guess the words that belong to their team based on this clue. Each word they guess is revealed as red, blue, or neutral. If a team guesses a word that does not belong to them, their turn ends. Teams alternate giving clues and guessing from them until one team has revealed all their words, that team is considered the winner.

Example of a codenames board

3 Word Vectorization

The underlying mechanism of how the computer assigns a semantic "closeness" to a word is via word vectorization, a reasonably common model where each word or morpheme is assigned a point is some high-dimensional space, and the distance between two such points represents the semantic closeness between the two words. The vectors themselves are generated programmatically by scanning a large corpus of text and generating vectors based on which words appear near one another in context.

There are three popular algorithms for generating such vectors – word2vec, fastText, and GLoVE – most of which come with a large, pre-generated set of word vectors for ease of use. I wound up using fastText for several reasons, the main one being that it has the very useful feature of being able to ascribe a vector to any word, even one it has never encountered before or even utter gibberish by using vectors of known sub-strings. However, if word2vec is given a word it has never encountered before, it will simply fail to produce a vector, so using fastText saves a considerable amount of error handling. This feature

also allows more diverse and creatives guesses from the user, although the clue-generation function is still limited to words fastText has encountered before, although it's vocabulary is considerable because of the sub-string feature. I used a standard set of pre-generated vectors for this project, which produced good results and contained some 500,000 word-to-vector mappings.

The other method of measuring the semantic similarity of two words is using Wu-Palmer similarity, a metric that uses the WordNet database. Despite the fact that the original idea for this project ironically came from an attempt to vindicate WordNet, this method turned out to be fairly useless. Wu-Palmer similarity is highly deterministic, it is computed using the formula

$$\text{Wu-Palmer}(w_1, w_2) = 2 \cdot \frac{\text{depth}(\text{most recent common ancestor of } (w_1, w_2))}{\text{depth}(w_1) + \text{depth}(w_2)}$$

Based on the fact that WordNet is organized as a giant tree, with each word having a set of hyponyms (more specific) below it and hypernyms (less specific) above it. This method proved ineffective and measuring semantic similarities for any words that are not hyper/hyponyms of one another, which makes it unsuitable for a task which requires measuring how humans freely associate words. See table below comparing different semantic models.

Word Pair	Word2Vec	FastText	Wu-Palmer
apple, orange	0.392	0.503	0.783
snowstorm, cold	0.36	0.401	0.095
theft, criminal	0.37	0.414	0.105
unrelated, word	0.02	0.106	0.222

Note that the word pairs (snowstorm, cold) and (theft, criminal) have a low order of similarity by Wu-Palmer, although they are commonly associated in natural language semantics. (apple, orange) and (unrelated, word) are included as a control.

4 Guessing Algorithm

4.1 Deterministic Algorithm

The algorithm to guess words based on a clue – that is, given an input board of 25 words of unknown affiliation, a clue, and a number n, output the n words on the board that the clue is referring to – is relatively straightforward. It is as follows:

1. Find the FastText similarity between each word on the board and the clue
2. Sort the words on the board in descending order of similarity to clue
3. Return the first n words of that order

Or in code

```python
#implemented on line 70 of algs.py
def ft_guess_from_clue(board, clue, n):
    guesses = []
    for i, option in enumerate(board[0]):
        #if the word has already been guessed, skip it
        if board[3][i] == "True":
            continue

        guesses += [(option, ft_model.similarity(clue, option))]

    guesses = sorted(guesses, key= lambda g : g[1], reverse=True)
    return guesses[0:int(n)]
```

It's pretty straightforward why this works in theory, and it works pretty well in practice, understanding human guesses reasonably well. If you want to try it yourself, uncomment `#player_cluegiver(board)` on line 192 of `codenames.py` in the source code. This will allow the user to enter clues for a random board and see how the algorithm reacts to them.

4.2 Probabilistic Algorithm

However, the deterministic guessing algorithm is not a good model for how humans actually guess words in a real game of codenames. I attempted to make a more probabilistic model by mapping each similarity score to a probability. However, since any given similarity score is usually on the range [0.3, 0.6], simply normalizing the probability scores such that they add up to 1 and using that as a probably map is ineffective, since if one words has a similarity to the clue of 0.5 and all the rest have 0.2, the algorithm will get it wrong 50% of the time when the answer should be obvious. After some trial and error, I found the most effective algorithm is to take only the top 20% of guesses, apply the sigmoid function to each of them (in order to map low-probability guesses to 0 and high-probability guesses to 1), and normalize the resulting values so that they sum to 1, where the sigmoid function is

$$\sigma(x) = \frac{1}{1 + e^{-(a(x-b))}}$$

With coefficients $a = 20, b = 0.3$, also determined by trial and error.

1. Find the FastText similarity between each word on the board and the clue
2. Keep the top 20% of the guesses, discard the bottom 80%
3. Take the sigmoid function of each similarity score, defined above
4. normalize the each resulting value such that the values add up to 1
5. pick n words at random, without replacement, based on the probabilities computed in step 4

Or in code

```
def sigmoid(x, a=1, b=0):
    x_a = a*(x - b)
    return 1/(1 + np.exp(-(x_a)))

#defined on line 162 of algs.py
def ft_guess_from_clue_prob(board, clue, n):
    guesses = []
    for i, option in enumerate(board[0]):
        if board[3][i] == "True":
            guesses += [0]
        else:
            guesses += [ft_model.similarity(clue, option)]
    guesses = np.array(guesses)

    #make a stochastic vector
    q5_guesses = np.quantile(guesses, 0.8)
    filtered_guesses = np.array([g if g > q5_guesses else 0 for g in guesses])
    sigm_guesses = sigmoid(filtered_guesses, a=20, b=0.3)
    stoch_guesses =  sigm_guesses / np.linalg.norm(sigm_guesses, ord=1)

    #choose based on weights
    guess = np.random.choice(board[0], n, p=stoch_guesses, replace=False)
    return guess
```

This function is used by the AI opponent when playing against the computer, as it is fairer (i.e. feels less like the computer is cheating) than the deterministic model. Since the code master and the guessers have the exact same understanding of how words are similar to one another when played by the computer – a quality which a team of multiple humans do not share – it is unreasonably good at guessing its own clues. This probabilistic guessing function better models how humans guess, and levels the field when playing against the computer while keeping it somewhat competent. This is the function the computer uses to guess words when a human is playing against it.

5 Clue-Generating Algorithm

The main problem to solve with the codenames AI is an algorithm to generate clues, that is, given a 25-word board, generate the word-number pair that will lead a *human* player to guess the most correct words.

There were two attempts made based on roughly the same principle, guessing words based on their centroid – the average vector representing all the words the computer is trying to generate a clue for. The intuitive way to do this is to take the closest word to the centroid, but I found empirically that the results of this are not that great, and it has the problem that it is very difficult to

avoid generating a misleading guess - one that may cause the guessers to guess a word belonging to their opponent. Attempting to solve this issue by giving opposing words a negative weight when calculating the centroid (i.e. making the centroid a point near all the target words and far from the opposing words) was empirically found to produce nonsensical guesses.

What I found to be the best approach is an adverserial algorithm. First, generate a list of $k = 500$ candidate words – the 500 nearest words to the centroid of the target words. Then, for each candidate word, try running it through the deterministic guessing algorithm, and return the best clue. The best clue is defined as the clue that the computer used to guess the most words without guessing any words incorrectly. Although it is imperfect, the deterministic guessing algorithm is a much better model of how well a human will guess with a given clue than the distance in the semantic space alone. The code is as follows:

```
clues = [] #defined on line 109 of algs.py
iter = 0
N = min(len(good_words) + 1, 6) #don't try to guess more than 6 words

best_match = ft_model.most_similar(positive=good_words, k=500)
for b in best_match:
    if is_valid_clue(b[0], board, past_clues):
        clues.append(b[0])

best = ("NO CLUE", 0, [])
for j, c in enumerate(clues):
    for i in range(1, len(good_words)):
        guess = ft_guess_from_clue(board, c, i)

        correct = True
        new_weights = []
        for g in guess:
            new_weights.append(g[1])
            if g[0] not in good_words:
                correct = False
        old_weights = []
        for g in best[2]:
            old_weights.append(g[1])

        if (correct and i > best[1]) or
        (correct and i==best[1] and mean(new_weights)>mean(old_weights)) or
        (correct and i>best[1]-1 and mean(new_weights)-mean(old_weights)>0.2):
            best = (c, i, guess)

if best[1] == 0:
    best = (clues[0], 1, [])
return best
```

There are a few things I'd like to point about this particular implementation, firstly that the computer is limited to generating clues referring to 6 words or fewer. This is for two reasons, one is to save on compute time, and the other is because it was generally found that a clue with more than 6 words tied to it are so vague as to be meaningless.

Secondly, there is the very long `if` statement determining whether or not to pick a word as the new best option. The first clause, `(correct and i > best[1])`, will select a clue if it will lead the computer to pick all correct words, and picks more correct words than the best known clue. The boolean `correct` is `true` if and only if the clue led the computer to pick only words belonging to its team. The second clause is a tie-breaker,

```
correct and i==best[1] and mean(new_weights)>mean(old_weights)
```

That will choose the clue if it picks the same number of correct words as the best known clue, but has a higher average similarity to the being referred to. Finally, the clause

```
correct and i>best[1]-1 and mean(new_weights)-mean(old_weights)>0.2
```

is somewhat experimental, it tells the computer that it's acceptable to reduce number of words referred to by the best known clue by 1 if the new clue is much closer in similarity to its referred words. I did not keep track if this clause is ever invoked, so it may very well be redundant, but the idea is that a clue that is much easier to guess is worth sacrificing 1 word worth of expected guesses. The cutoff value of a difference in score of 0.2 was picked arbitrarily.

6 Results

6.1 General Results

In general, I found that the computer plays codenames reasonably well. I tried extensively playing against the computer, as well as an experiment meant to determine how well it plays against a full team of humans. It's sense of semantic understanding is still a little off compared to a human's, but I suspect this is a fundamental limitation of the word vectorization algorithm.

The main problem I observed that was with the codenames implementation specifically was that the algorithm was unreasonably good at guessing it's own clues, often guessing 3-5 words at a time when one or more words had no relation to the clue. This is because unlike two human players, who have subtly different associations between words, the computer has exactly one sense of how words relate to one another, the values in the word vectorization model. This might be solved by two different sets of word vectors used for guessing and for generator clues, but I have not had a chance to try and see if it would work. Overall, I would say the project was successful, since the computer program can play codenames pretty well, if not appearing to cheat a little and often losing in

direct competition to a human. The fact that a computer could compute with a human at all in a mainly semantic game is a recent development, since the required compute power as well as the underlying linguistic research only became available in the past 5-10 years.

6.2 Social Experiment

One thing I wanted to test about the model was how well it fares compared to a human at the most difficult task, thinking of good clues that other *humans can understand*. To test this, I performed a test to see how well the AI actually performs in an actual game. With the help of volunteers from the Boston University Board Games Club, I had the AI play 5 codenames matches in which all participants were human except for one AI codemaster. The AI fared somewhat badly, losing 4 out of 5 games it played. However, it would play consistently badly, with the computer-aided team losing consistently by 1-3 unrevealed words. It also managed to win on one occassion, generating some genuinley effective clues in the process. The main complaints I got from the volunteers was that clues generated are often cryptic (i.e. make sense, but only after you know the word they're referring to) or very overspecific. Overall, while the AI was not able to consistently outcompete a human, it played enough like a human (albeit a somewhat incompetent one) not to disrupt the flow of the game.

7 Next Steps

There are some rules of codenames which I omitted to simplify the game for engineering purposes. One was the so-called "death card", where one of the words is designated the death card, and any team which guesses it loses on the spot. The other is the ability to make a "zero-guess", where a codemaster may give a clue referring to 0 words, which is interpreted as referring to all unrevealed words on the board except those related to the clue. Adding these would be primarily an engineering challenge, but it may be a good follow-up project to add these rules.

The other far more interesting idea of what to do next with the project is to train multiple versions of the word vectorization model, each based on a different corpus of text. For instance, one could have a cluegiver with word vectors based on the vocabulary of Shakespeare being interpreted by a player with word vectors based on the vocabulary of P.G. Wodehouse. Other good corpus candidates for this are the works of Lewis Carroll, J.R.R. Tolkien, and Kurt Vonnegut. I did not have time to actually implement this before the deadline of the project, but I hopefully I will give it a try in the near future. If I do, the one thing I'm not sure is how to go about sharing the word vectors, as the word vector files tend to gigabytes big, well beyond what can be committed to a git repository. I expect I'll have to use google drive or some other cloud storage mechanism with a download script of some kind.