AUTOMATING CODENAMES SPYMASTERS: Ensemble Similarity with Word and Text Embeddings

Luca Borletti
Carnegie Mellon University
lucaborletti@cmu.edu

Jack Wei
Carnegie Mellon University
jackwei@andrew.cmu.edu

Abstract

Codenames is a word-based board game with simple rules, but requires a deep understanding of language. One of the primary strategies of Codenames is understanding the multiple possible definitions of a possible hint and how they relate to the board words. In this paper, we propose a spymaster algorithm that can generate hints, leveraging this multi-definition strategy. First, we examine contextual embeddings and dictionary embeddings to embed one word into multiple embeddings. We then work to combine multiple models we have created into one and use the experts framework to finetune our weights. Using our final model, we found up to 27% improvement in some categories from our baseline word-embedding model based on fastText. Overall, our paper shows the effectiveness of multi-dimensional embeddings and moves towards creating a spymaster algorithm with human ability.

1 Introduction

1.1 What is Codenames?

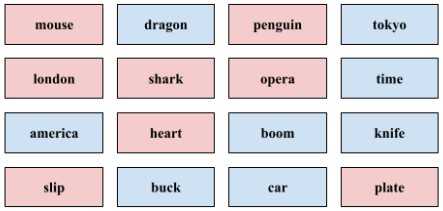
Codenames is a turn-based game played by two competing teams of players, the 'red' team and 'blue' team. Each team consists of one spymaster and at least one guesser. The goal of each spymaster is to give hints to their guesser(s) about which words are assigned to their team. Each round, the playing team's spymaster crafts and reveals a one-word hint, h, and positive integer, c, to their team. The spymaster should intend for the hint h to be related to c words on the board. The hint must be semantic, not phonetic or syntactic.[1] The guessers on that team must then guess which words the spymaster hinted at. The first team to have all of their words be guessed (by their own guessers and/or by the opposing teams') wins.

1.2 The "Spymaster Problem"

Put simply, we aim to automate the creation of *useful* hints given game boards (i.e., making a 'spymaster' bot). Specifically, given a set of 16 English words, and a subset of size 8 being 'our' words (conventionally, the '**blue**' words)—the other 8 are 'their' or '**red**' words—we want to produce the one-word clue (and number) that maximizes the chance of **blue** *human guessers* guessing **blue** words. Variations of the "spymaster problem" have been explored by several researchers over the last few years. Kim et al. were the first to leverage a deep learning technique—namely, word embedding models—to search for useful hints [3]. Others, such as Jaramillo et al. and Koyyalagunta et al., expanded on their work by testing more embeddings models [4] [2] and performing human testing [4]. Below is an example of some of their models' hint-generating capabilities:

[1] Phonetic example: using "cat" to hint at "hat." Syntactic example: using "tool" to hint at "toolbox."

Preprint. Under review.



	word2vec-300	GloVe-300	fastText
hint, count	**washington, 2**	**world, 2**	**sword, 2**
intended word 1	**tokyo**	**america**	**dragon**
intended word 2	**america**	**time**	**knife**

Figure 1: Spymaster output with various word representations on an example game board. "Washington" is a good clue for its intended words, "tokyo" and "america," but runs the risk of guessers also guessing "london." Similarly, "world" is an example of a hint that is too generic, relating heavily to the red word, "london." "Sword," on the other hand, is an example of a high-quality hint, as it is related to its intended blue words while not strongly hinting at any red words.

Despite promising results from the literature several challenges remained:

- **Codenames game data is scarce and heterogenous**—with only a few datasets publicly available—making it hard to fairly and accurately assess spymaster bot performance.
- **Homonyms**[2]**, hypernyms**[3]**, and meronyms**[4] are word types that current models underperform on.
- **Different embedding models have different "expertises"** or categories of words that they tend to perform better on, so only using one at a time may not be the best strategy.

1.3 Ensemble Similarity and Other Improvements

This paper aims to improve the state-of-the-art by tackling the above challenges:

- To help alleviate **Codenames game data scarcity** and more fairly evaluate "spymaster bots," we built and evaluated a "guesser bot" that uses OpenAI's GPT-3 large language model to model human guessing. To ensure that our guesser reasonably approximates human guessers, we compared its guesses to humans' guesses over a dataset of 1,462 Codenames game scenarios [5].
- To tackle the issue of under-performance on **homonyms, hypernyms, and meronyms**, we modify Kim et al.'s algorithm to work with multi-embedding representations of words. Specifically, we used "dictionary / definition" embeddings and "context" embeddings.
- Finally, to take advantage of the **varying "expertise"** of different word representations, we extend Kim et al.'s algorithm by introducing an "ensemble similarity" technique. This involves weighting and combining different models' hint-board word similarity scores. To learn these weights, we apply the "Multiplicative Weights Update" algorithm (MWU) [1].

2 Related Works

The basis of our work is **Kim et al. (2019)**. They introduced the concept of embedding words to find the best possible hint as a spymaster. In particular, they built several spymaster and guesser bots: word2vec bots, GloVe bots, and concatenations of both. Equipped with these word embedding models, Kim et al.'s general spymaster algorithm (assuming we are on the blue team) is to loop across all subsets of the blue words, B, of a fixed size, s_h, and find the word from the 10,000 most common English words that achieves the highest 'score' with that subset. To define a score, suppose we have a subset $S \subseteq B$ of the blue words and a potential hint word, h, from our dictionary. Let R be the set

[2]Homonyms: words that sound or are spelled the same but have different meanings (e.g., "bank" as a financial institution or river bank).

[3]Hypernyms: general category terms that encompass more specific items (e.g., "vehicle" for truck, car).

[4]Meronyms: words for parts of a whole (e.g., "petal" is a part of a "flower").

[5]Scenarios extracted from 794 games in the Cultural Codes dataset in Shaikh et al (2023): `https://github.com/SALT-NLP/codenames/tree/main`

of red words, $\text{sim}(x, y)$ represent the cosine similarity between two word embeddings, and f be our embedding function. The score of hint h on subset S is defined as follows:

$$\text{score}_1(h, S) = \begin{cases} \min_{w \in S}(\text{sim}(f(w), f(h))) & \min_{w \in S}(\text{sim}(f(w), f(h))) > \max_{r \in R}(\text{sim}(f(r), f(h))) \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

In other words, the score of hint h on subset S is defined as the minimum similarity[6] between the hint and any word in S, if that similarity is larger than h's similarity to any red word, else the score is 0. Under this scoring system, Kim et al. found that the best performing spymaster used a word2vec+GloVe embedding function. One limitation of this model relates to multi-definitional words (i.e., homonyms). For example, a word like 'die' has multiple meanings, such as 'to cease living' and 'a cube with numbers on its faces'. This makes it a potentially useful hint if words like 'death' and 'game' are on 'our' words on the board. However, single word embedding models struggle to capture these multiple meanings, which can lead to suboptimal hints, as seen in Figure 2, below:

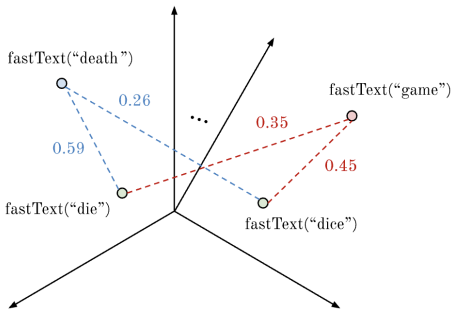


Figure 2: Simplified view of the relative positioning of a few words in the fastText embedding space. Ideally, a spymaster bot would leverage the 'die'↔'death' similarity as well as the 'dice'↔'game' similarity when using the hint, 'die,' but this is not possible with the above framework.

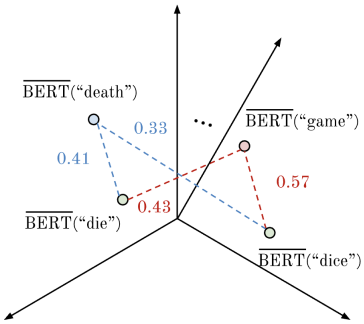


Figure 3: Simplified view of the relative positioning of a few words in the BERT *averaged* contextual embedding space (aka $\overline{\text{BERT}}$). Compared to Figure 2, we see a much stronger similarity between 'die'↔'game' which is a more accurate representation of the word's multiple meanings.

Another significant drawback of Kim et al.'s work is that spymaster bots are tested against bots with similar "similarity-ranking" algorithms, so the results are not necessarily representative of how effective spymasters are when paired with humans. Unsurprisingly, spymasters paired with guessers with the same underlying embedding system performed optimally with 100% success rate [3].

Jaramillo et al. (2020) did some expansion on Kim et al.'s work, finding a transformer based GPT-2 model was able to perform competitively with Kim et al.'s word2vec+GloVe model [2]. **Koyyalagunta et al. (2021)** were able to greatly improve the results found by Kim et al. They fully revamped the scoring system proposed by Kim et al. by adding "DETECT, (DocumEnT frEquency+diCT2vec), a scoring approach that combines document frequency, a weighting term that favors common words over rare words, with an embedding of dictionary definitions" [4]. To do the document frequency, they use a cleaned subset of Wikipedia. Koyyalagunta et al. also used the contextual embedding model BERT. Each word was embedded under multiple different contexts and the average of each context was used as the main embedding. Finally, Koyyalagunta et al. were able to use human testing from Amazon Mechanical Turk with over 1440 total samples. Using human testing, Koyyalagunta et al. found DETECT scoring to universally improve performance with the best performance embedding being fastText [4]. Thus, while BERT allowed for a more robust embedding of words, as homonyms could be better captured (see Figure 3, above, for an example), it still under-performed word embedding models, such as fastText.

[6]Note that when we say "similarity" between words, we mean the cosine similarity between their vector representations under a specific embedding model.

Lastly, **Shaikh et al. (2023)** apply the above techniques to a related game, Codenames Duet. Importantly, they also released a Cultural Codes dataset consisting of 794 games with 7,703 turns by human players [5]. We use this data to measure the effectiveness of our own "guesser" bot below.

3 Project Background

In our midway report, we started by improving the runtime of the spymaster algorithm by Kim et al. (as described below in **Methods**). To measure the effectiveness of our and all existing methodologies, we also built a "guesser bot" using GPT-3 and evaluated it using the Cultural Codes dataset (also detailed below). Equipped with the "guesser bot," we then tested the spymaster algorithm on all of the word representations we observed in the literature. For our performance metrics, we tracked 1) the percentage of words that were **intended** to be guessed by the spymaster bot that were indeed guessed by the guesser bot, 2) the percentage of guesses that **correctly** guessed 'our' words, and the percentage of hints that were "**perfect hints**." A hint is considered perfect if all of the intended board words were guessed by the "guesser bot." We present our findings below:

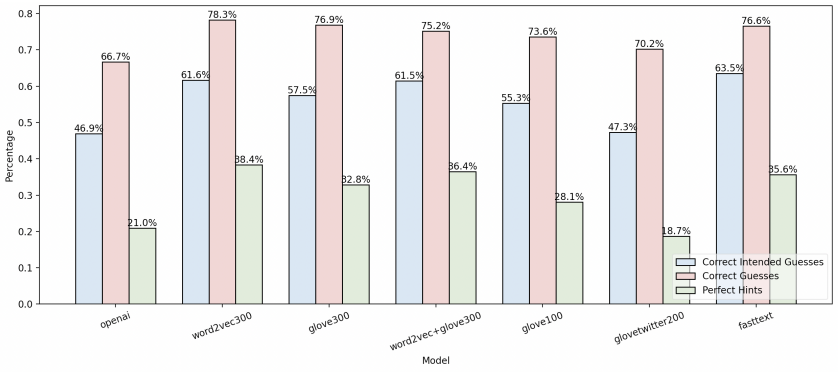


Figure 4: Results From midway. The most effective of the tested word embedding models was fastText, with word2vec and GloVe following closely behind.

4 Methods

4.1 Guesser Bot

As explained in our midway report, we created a GPT-3 "guesser bot" to measure the accuracy of our models. Our guesser bot algorithm consisted of randomly permuting the board words, and then prompting GPT-3 with the hint word and number, asking it to guess the intended hint words. To test how well it models human guessers, we compared its guesses in 1256 "guesser scenarios" from the Cultural Codes Dataset to corresponding guesses from the human players. The bot achieved a human-guess accuracy of 54%—the percent of guesses by the guesser bot that matched human guesses. Additionally, we found an overall "blue"-guess percentage (% of guesses that correctly guessed a blue word) of 60% by the bot and 80% by human players. We noticed that the effectiveness of models measured by Amazon Mechanical Turk testing in Koyyalagunta et al.'s work [4] were heavily correlated with the effectiveness found by our "guesser bot." We tried using other popular LLMs for testing like Claude and ChatGPT-4. However, these methods were found to be too expensive or too slow for our uses. Another problem we encountered is that ChatGPT-3 would sometimes "misfire". ChatGPT-3 would either not provide a clue on the board or provide the wrong number of clues. These misfires were rare enough that we were able to simply ignore these data points.

4.2 Spymaster Algorithm

While the basis of methods is Kim et al's spymaster algorithm, the original algorithm was considered too slow for our uses. As detailed above, their approach involved scoring each of the 10,000 potential hints for every possible subset of 'our' words of the desired size. Assuming an unfixed hint subset size, s_h, and n 'our' words, this means 2^n iterations (similarity computations) for each of the $10,000$

potential hints—exponential with respect to the size of 'our' words. However, since all algorithms up to this point have fixed s_h prior to hint generation, this results in $\binom{n}{s_h}$ iterations for each hint.

For $s = 2$, we improve on this by at least an $\frac{n}{\log n}$ factor. The key insight is that, for a potential hint word, h, calculating the cosine similarity between $f(h)$ (for some embedding function f) and each word in *every* subset of 'our' words leads to some redundant calculations between subsets with overlapping elements. Instead, we can simply compute $\text{sim}(f(h), f(o_i))$ for each word, o_i, in 'our' words in $\text{O}(n)$ time, sort these similarities in $\text{O}(n \log n)$ time, and select the words corresponding to the top s_h similarities to be the best subset for that hint. This results in approximately $\text{O}(n \log n)$ work per hint[7].

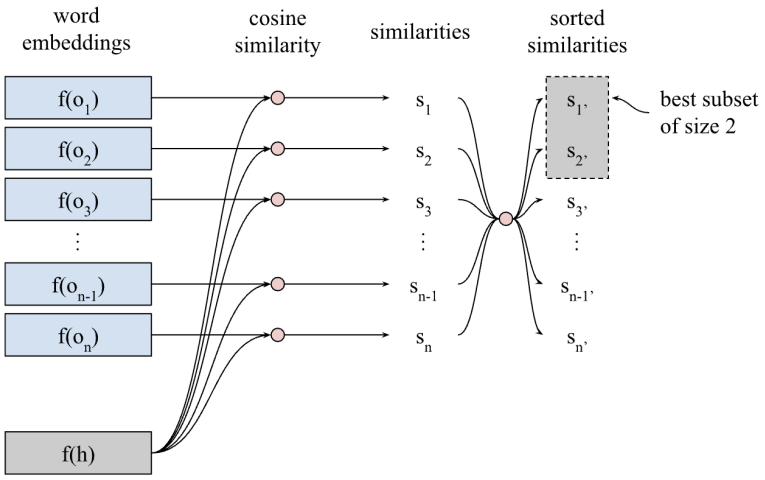


Figure 5: Spymaster algorithm optimization visualized (steps take place left-to-right)

Recall, Kim et al.'s scoring function returns only returns a positive score when the minimum similarity between the hint, h, and words in the chosen subset, S, are greater than the maximum similarity of h to any of 'their' words. Our adapted algorithm accounts for this by calculating $\text{sim}(f(h), f(t_i))$ for each t_i in 'their' words and then comparing the highest of those similarities to the lowest similarity from the hint subset.

While this approach significantly improves runtime, it does not address the recurring issue relating to multi-definition words (homonyms) and other semantic relations. To address this, we propose representing a word with **multiple embeddings simultaneously** (*without* concatenating these vector representations). Ideally, this should enable different embeddings for a single word to represent the different meanings or contexts that the word can have or appear in, respectively.

We now need to find a reasonable way of computing the similarity between two words, x and y, where the embeddings $(e_1^x, e_2^x, \ldots, e_n^x)$ represent x and $(e_1^y, e_2^y, \ldots, e_m^y)$, y. Intuitively, we would expect the similarity between x and y to equal the cosine similarity of the best two choices of embeddings from each words' representation. That is exactly what we propose:

$$\text{sim}_{\text{multi}}(x, y) = \max_{i,j} \left(\text{sim}(e_i^x, e_j^y)\right) \tag{2}$$

This approach can be used with several types of embedding models, which we explore below.

4.3 Contextual Embeddings

The first method which we consider combining with multi-embedding representations and go beyond our baseline results was embedding each word under different contexts using contextual embedding models. To do this, we use GPT-3 to generate 3 different contexts for each of our 10,000 words. Then

[7]Since we are assuming a small, fixed s_h, we can speed this up further by using a heap to get the top s_h elements in $\text{O}(s_h \log(n))$ time.

we take a word w and an associated context c and use a contextual embedding model, such as BERT, to embed the entire context. We extract the embedding for w by at the index of the token associated with w. If there are multiple such tokens, we take the average of these embeddings to get our final embedding. In the end, each word has 3 associated contextual embeddings, one for each context.

When running our multi-dimensional spymaster algorithm across multiple different models, we found that most models performed quite poorly by themselves. Only BERT and DistilBERT were able to perform comparably to our previously best performing model FastText. The remaining four models—Roberta, GPT2, XLnet, and Albert—performed significantly worse. From this, we had the idea to combine the outputs of a contextual embedding model (say Bert) with FastText in the following manner:

$$\text{sim}(w_1, w_2) = \text{sim}(\text{fasttext}(w_1), \text{fasttext}(w_2)) + w \cdot \text{sim}_{\text{multi}}(\text{BERT}(w_1), \text{BERT}(w_2))$$

where w is some weight between 0 and 1, fasttext() is the fasttext word embedding model, and BERT() is the BERT contextual embedding model. The motivation here is to give some additional information that contextual embeddings provide to an already powerful model like FastText. These weighted models performed quite well.

4.4 Dictionary Embeddings

In addition to contextual embeddings, another multi-dimensional embedding we tried is dictionary embeddings. As outlined by many of the related works, word embeddings were found to not properly encode certain word relations like hypernyms and meronyms. To address this, we tried embedding the dictionary definition of words instead of the word directly, as the dictionary definitions are more likely to encode such relations. To do this, we used Merriam Webster's Dictionary API[8] to retrieve multiple definitions of each word. Then, we used OpenAI's text-embedding-3-small model to embed each definition. This allowed us to associate each word with n different dictionary definition embeddings.

Similar to the contextual embeddings, we used our multi-dimensional spymaster to test the effectiveness of solely using dictionary embeddings. On its own, the dictionary embeddings performed quite poorly. We then tried combining these embeddings with fastText. This model performed slightly worse than fastText, but was still competitive.

4.5 Experts Framework / Ensemble Similarity Spymaster

At this point, we have both word embedding models (e.g. fastText and word2vec) and multi-dimensional embedding models (e.g. contextual embedding and dictionary embeddings). While the latter multi-embedding models performed somewhat poorly alone, we noticed that in many examples involving complex semantic relationships (such as in Figure 3), they were able to provide better hints. This led to us attempting to combine the results of multiple models, to leverage all of their individual "**expertise**." Our goal was to find a way to meaningfully combine the following five models: fastText, GloVe, word2vec, Bert Contextual Embeddings and OpenAI Dictionary Embeddings. Initially, we tried arbitrarily assigning weights to each model, and adding together the weighted cosine similarities between the hint word and other words according to each model's word representations, however, this did not lead to good results. Then, we tried to leverage to "experts framework" or "Multiplicative Weights Update" algorithm (MWU). The full algorithm can be seen here[1]. The core of the algorithm is learning which of our "experts" (in this case, embedding models) is the most accurate. When a model performs poorly (gives a bad hint) it is penalized and its weight is lowered by a factor of $(1-\epsilon)$[9]. As seen below, we ran this algorithm for 1,500 games, using our "guesser bot" as the ground truth (penalties are incurred when the "guesser bot" guesses incorrectly given a hint from one of the models). This method proved to be highly effective, and we were able to find our best model using this technique. We coin the phrase "ensemble similarity" to represent this model—an ensemble of our other models.

[8] `https://dictionaryapi.com/`

[9] Our best performing "ensemble" model resulted from an $\epsilon = 0.01$.

5 Results

5.1 Contextual Embeddings

In figure 6, we show the results from the 6 contextual embedding models we tried. We provide the results from FastText for comparison. As noted previously, only Bert and DistilBert performed comparably to FastText, with the other 4 models performed signicantly worse. We are not fully sure why many of these models failed to show results, including more powerful models like Deberta and GPT2. We suspect this is because many of the more complex models are meant to be finetuned for downstream tasks. However, we were unable to perform any finetuning as we don't have an expansive dataset of real games played to train on. We then tried incorporating contextual embedding models

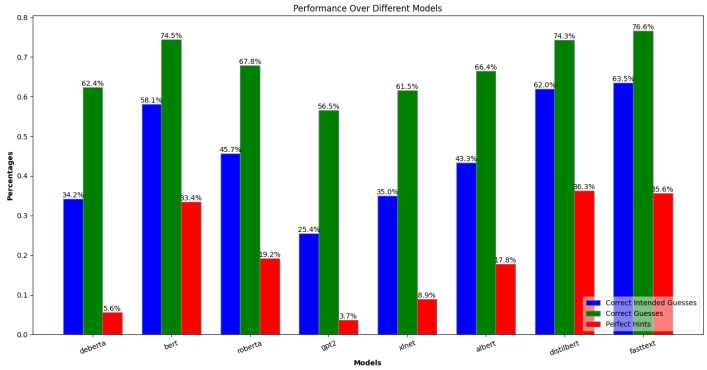


Figure 6: Contextual Embedding Models

with FastText. Through trial and error, we found 3 combinations of models and weights to actually improve results from FastText. The 3 models are Bert with $w = 0.1$, DistilBert with $w = 0.1$ and Roberta with $w = 0.05$. We show the results in a table under Figure 7. The best performing model

Model	Weight	Intended Guesses	Correct Guesses	Perfect Hints
Bert	0.1	64.0	77.8	37.8
Distilbert	0.1	64.4	78.1	38.1
Roberta	0.05	65.8	80.2	39.7
Base FastText	-	63.5	76.6	35.6

Figure 7: FastText with Weighted Contextual Embeddings

was Roberta with $w = 0.05$. This is most interesting because Roberta on its own was not an effective model that performed significantly worse than Bert, DistilBert, and FastText.

5.2 Dictionary Embeddings

We provide the results from the dictionary embeddings in Table 8. The FastText + Dict model uses a weight of 0.1 for the dictionary embeddings. As with the contextual embeddings, the dictionary embeddings perform quite poorly on their own, but when combined with FastText performs quite well.

Model	Intended Guesses	Correct Guesses	Perfect Hints
OpenAI Dictionary Embeddings	40.3	61.9	12.7
FastText + Dict	62.5	77.8	38.2
Base FastText	63.5	76.6	35.6

Figure 8: Dictionary Embedding Models

5.3 Ensemble Similarity

We present figure 9 for the finetuning of our Ensemble Similarity Model. We use the experts framework to compute the weights over multiple iterations. Note all models start with a weight of 0.2.

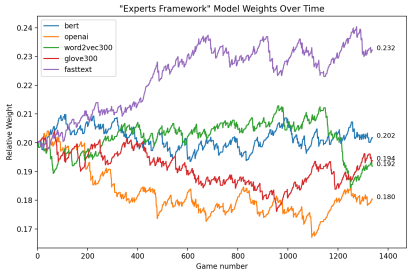


Figure 9: Expert Weights Over Time

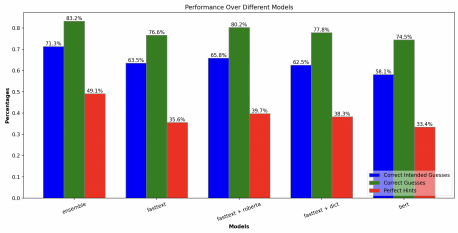


Figure 10: Ensemble Similarity Results

The Ensemble Similarity Model is our best performing model. Here in Figure 10, we show it compared to some of our models and our basline of fastText.

6 Discussion and Analysis

In our paper, we made two important discoveries:

1. Contextual and dictionary embeddings are able to encode more information than a normal word embedding
2. Combining models in a clever way allows for better results than any singular model is able to achieve

Our final Ensemble Similarity Model showed significant improvements in every category as compared to our baseline model fastText. We had a 12% increase in intended guess percetange, 8% in correct guess percentage and 27% in perfect hint percentage. Although this was our best performing model, other models like mixing fastText with contextual embeddings and dictionary embeddings were shown to be effective as well.

In order to justify our findings, we had to make the assumption that ChatGPT-3 is an accurate proxy for human guessing. However, the fact that ChatGPT-3's results are so similar to that of Koyyalagunta et al.'s human testing is a good sign. If we had an extensive dataset of human hints and guesses, we likely could have trained a more powerful model to accurately compute the weights of our Ensemble Similarity Model. Additionally, we could have finetuned more advanced contextual embedding models like Deberta or GPT2. Without this finetuning, these models performed signicantly worse than the other simpler models.

Another avenue of improvement is playing out full games. Currently, we only consider one round of gameplay which takes out more advanced game-theory strategies. A fully trained spymaster bot, for example, would be able to remember previous hints it has given and track what words a guesser could be thinking of.

References

[1] Sanjeev Arora, Elad Hazan, and Satyen Kale. The multiplicative weights update method: a meta-algorithm and applications. *Research Survey*, 8(6):121–164, 2012. Received: July 22, 2008; Revised: July 2, 2011; Published: May 1, 2012.

[2] Charity M. Canaan R. Togelius J. Jaramillo, C. Word autobots: Using transformers for word association in the game codenames. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, pages 231–237, 2020.

[3] Andrew Kim, Maxim Ruzmaykin, Aaron Truong, and Adam Summerville. Cooperation and codenames: Understanding natural language processing via codenames. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 15, pages 160–166, 2019.

[4] Divya Koyyalagunta, Anna Sun, Rachel Lea Draelos, and Cynthia Rudin. Playing codenames with language graphs and word embeddings. *Journal of Artificial Intelligence Research*, 71:319–346, 2021.

[5] Omar Shaikh, Caleb Ziems, William Held, Aryan Pariani, Fred Morstatter, and Diyi Yang. Modeling cross-cultural pragmatic inference with codenames duet. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023*, pages 6550–6569, Toronto, Canada, July 2023. Association for Computational Linguistics.