Final Project Report

Using CodeNames to Evaluate Human Perception of Inter-word Relationships

Team Members: Divya Satrawada, Mansi Achuthan, Nam Nguyen, Kyle Thompson

1. Introduction

In this work, we create two autonomous agents that play codenames, a two to four player word association game. We use our agents to compare formal methods and learned embeddings for playing linguistic games. In the age of deep learning, we are curious to see if explicit knowledge representations can be competitive. If the two systems perform similarly, one might prefer the knowledge-based agent for its interpretability. The steps used in the reasoning process are more transparent than those used in the deep learning reasoning process. Therefore, if they perform very well in some cases, or very poorly in some cases, humans are more likely to understand why.

For both of our agents, we define methods of finding the similarity between two words. One of the agents uses a semantic database called WordNet to find its similarities, and one uses learned embeddings from a model called dict2vec. We use these similarities to inform the clues of our two spymaster agents. We then evaluate these agents through simulation, and human testing. We conclude that although the behavior of the formal methods agent is vastly different from the deep-learning based agent, their performance is not significantly different. However, the performance of both agents is inferior to the performance of a human spymaster.

2. Problem Specification

Game Description

Our goal is to build agents that play codenames. Codenames is a word-association game typically played on a five by five board of cards. Each card contains the name of either a red agent, a blue agent, an assassin, or an innocent bystander [3]. All of our boards will have 9 red agents, 8 blue agents, 1 assassin, and 7 innocent bystanders. While the game is typically played with two teams, we will focus on a version of the game with only the red team. The team is composed of a “spymaster” and “guessers.” At each turn, the spymaster provides the rest of their team with a clue. A clue is composed of a single word, w, and a number, n, which signals to their team that there are n codenames for their team that relate to w. The agents must choose at least one card to turn over that they think belongs to their team. After they turn one card, they can choose to pass or turn over another card. They can continue turning cards until

they've turned one more card than the number stated by the spymaster. If the agents choose a card that does not belong to their team, their turn is over. Furthermore, if the card holds the assassin, the game is lost.

System Requirements

To evaluate the performance of the formal-methods agent, which we will call the WordNet agent, and the deep-learning agent, which we will call the Embedding agent, we must create infrastructure that allows the agents to play codenames. The infrastructure should allow humans to play codenames with autonomous agents as both the spymaster, and the guesser. The infrastructure should also allow for simulation of games between two autonomous agents. Additionally, the infrastructure should report meaningful statistics about the games played so that we can evaluate the spymaster agents.

To behave intelligently, our agents must be able to calculate the similarity between words in our clue vocabulary, C, and words in our board vocabulary B. Any word on the board must belong to the board vocabulary, and any clue must belong to the clue vocabulary. We have 394 words in our board vocabulary, and 4188 words in our clue vocabulary [6]. We use words from a previous codenames implementation for our board vocabulary. For our clue vocabulary, we use the top 5000 words from a word frequency site [10].

Given the similarities between the set of potential clues, and the words on the board, the agents must be able to identify the clue and the number of guesses that produces the best expected result. Therefore, we must have a metric that measures the utility of any clue and associated number.

Finally, our system must be able to play games in a reasonable amount of time. The time required to produce a hint must be comparable to the time needed by a human. We can set the maximum time allowed per hint at 3 minutes.

3. Related Work

Previous Codenames Agents

Kim et. al created the first published codenames agent [5]. The authors investigated methods of calculating similarity via the WordNet database, word2vec embeddings, and GloVe embeddings. The authors evaluated their spymasters and guessers by pairing them with each other and measuring the win rate of each pair. They find teams with two embedding-based agents cooperate better than teams with one embedding-based agent and one WordNet-based agent. They conclude that the embedding agents provide more intuitive hints for human guessers.

Koyyalagunta et. al built on the work of Kim et. al by further testing knowledge-based methods, and embedding-based methods for playing codenames [6]. These authors tested similarities found by traversing a lexical database called Babelnet, and by computing the cosine similarity of

embeddings (GloVe, fastText, BERT). The authors had trouble with obscure words, so they introduced an algorithm called DETECT which penalizes words that are not used frequently, and non-common relationships. The authors evaluated their work by getting actual rankings between hints and words from Amazon Mechanical Turk. They found that the embedding methods to be most successful in their evaluation with the human-labeled data.

Jaramillo et. al investigated a wide variety of codemaster, guesser pairs for playing codenames [4]. The authors considered approaches using the TF-IDF algorithm, Naive Bayes on a set of Wikipedia pages, and GPT-2 embeddings. They evaluated their agents through self play. Each of their spymaster agents played with each of their guesser agents and themselves. Their most successful spymaster-agent pair was the GPT-2 spymaster, and GPT-2 agent which reported a win rate of 0.9.

Our main contribution to the research problem of constructing codenames agents is a well motivated utility function. All three previous works did not provide much of a justification for their utility functions. Kim et. al chooses the set of agent words such that the smallest similarity between the clue and the set is larger than the largest similarity between the clue and an opponent word, and the largest similarity between the clue and an opponent word is smaller than a riskiness threshold [5]. Koyyalagunta et. al takes a similar approach, except they take the weighted sum between the average similarity between the guess and the agent words, and the maximum similarity between the guess and the opponent words [6]. Catalina et. al. use Laplace smoothing for their Naive Bayes implementation, the maximum term frequency of the most similar document for TF-IDF, and a KNN approach with weighting for their GPT-2 implementation [1].

Sources of Word Similarity

Our formal methods agent is based on the lexical database, WordNet [7]. WordNet groups words into synsets, or groups of synonyms. The words in a synset can also be called the lemmas of a synset. WordNet provides relationships amongst these synsets. For example, it defines that the synset corresponding to pet dogs is a subclass of the synset corresponding to domestic animals. WordNet provides many relationships of this sort. We use a version of WordNet offered by a Python library called Natural Language Toolkit which made it very easy to integrate the database into our agents [2].

Our word embeddings agent uses word embeddings from a model called dict2vec. Dict2Vec is the subject of Julien Tissier's PhD thesis [8]. Tissier noted that embeddings learned by models like word2vec sometimes entailed unpredictable relationships between words. He reasoned that this unpredictability could be a product of the corpus used to train the embeddings. For example, the co-occurrence of two words on Wikipedia might be less meaningful than the co-occurrence of two words in each other's dictionary definitions. For this reason, Tissier used a skip-gram architecture similar to the one proposed in word2vec, except he trained the model

over a corpus of lexical dictionary entries. As a result, dict2vec is more effective than word2vec at capturing formal relationships between words.

4. Implementation

Methods and Justification

To test the performance of formal-based and deep-learning-based approaches to playing codenames, we will use Wordnet, and dict2vec embeddings to create two spymaster agents. We then use a novel utility function to inform the clue choices by both spymasters. We will compare our simulation results to the results found in Jarmillo et al and Kim et al. [4], [5]. We also conduct a small human evaluation of our methods.

System Architecture

Our system is organized in four main types of modules: the Codenames simulation module, the similarity caching modules, the similarity loading module, and the strategy modules. Figure 1 shows a UML diagram of the system architecture described in this section.

Codenames Simulation Module

This module contains the code to simulate the game board, each player's turn, as well as functions to display results and evaluation statistics.

The module contains a Reader interface and a TerminalReader class, which implements the interface. This class is used for all standard input and output processing. The function read_picks reads the human agent's guesses, read_clue reads the human spymaster's clue, print_stats outputs the number of agent and opponent words left on the board when the game ends (with an option to reveal these words), and check_guess tells the human agent whether or not a chosen codename belongs to their team.

The main Codenames class handles the game play. There are functions to load the board's word bank and the clue's word bank, initialize the game board, play as a spymaster, play as an agent, and print the final results of the game. There are five options for the user to choose from: (1) play a game with the bot being the spymaster, (2) play a game with the bot being the agent, (3) simulate a series of gameplays where two bots play against each other, (4) get overall stats from all games, or (5) exit the simulation. The user can choose between two options for the bots: a bot trained on word embeddings and a bot that uses WordNet path similarities (using the combined strategy described below).

The simulation was based on the structure of Thomas Ahle's GloVe bot for Codenames [1]. The Reader interface, TerminalReader class, and Codenames class were used as a starting point for our UI. We replaced the strategy functions for the computer agent (make_guess) and computer spymaster (find_clue), as defined below. We also removed the word to vector and similarity

utility functions, and replaced the word bank used for Alhe's implementation with our own word banks, as described above.

Similarity Caching Modules

We quickly realized that calculating the similarity between each pair of words at the time of the game would be far too expensive. Therefore, we precompute the the similarity for every tuple in $C \times B$, and at runtime we load these similarities into a hash table. In our system, the CacheWriter class and its subclasses save the similarities between all pairs of words from the clue bank and the board bank. The similarities are saved for each type of similarity we investigate. Each of them are loaded as required at runtime.

Similarity Loading Module

The Similarity class loads the similarities calculated by the cache writers into a similarity object. The loaded similarities are then used by a Strategy object, as described below. The cache is specified through a parameter to the constructor. For example, if we are aiming to load meronym/holonym distances between pairs of words, we pass in "mer_holo.json" to the Similarity constructor. The resulting Similarity object will have a hashtable in memory where the key is a board-word, clue-word pair, and the value is the similarity of the pair according to the cached file.

Strategy Modules

The Strategy modules are split into two classes: the Strategy class, which allows the game to be played with a singular type of similarity, and the CombinedStrategy class which runs the game on all types of Wordnet similarities. The combined strategy merges the different similarities by taking their maximum. For example if hypernym similarity is 0.3 and antonym similarity is 0.1, the combined similarity will be 0.3. These classes allow our agents to play as a spymaster or guesser. Their find_clue function produces a clue using the classes calculate_similarity function, and the system-wide utility function defined in the utility section. Their make_guess function uses the calculate_similarly to choose a word based on the hint from a spymaster.

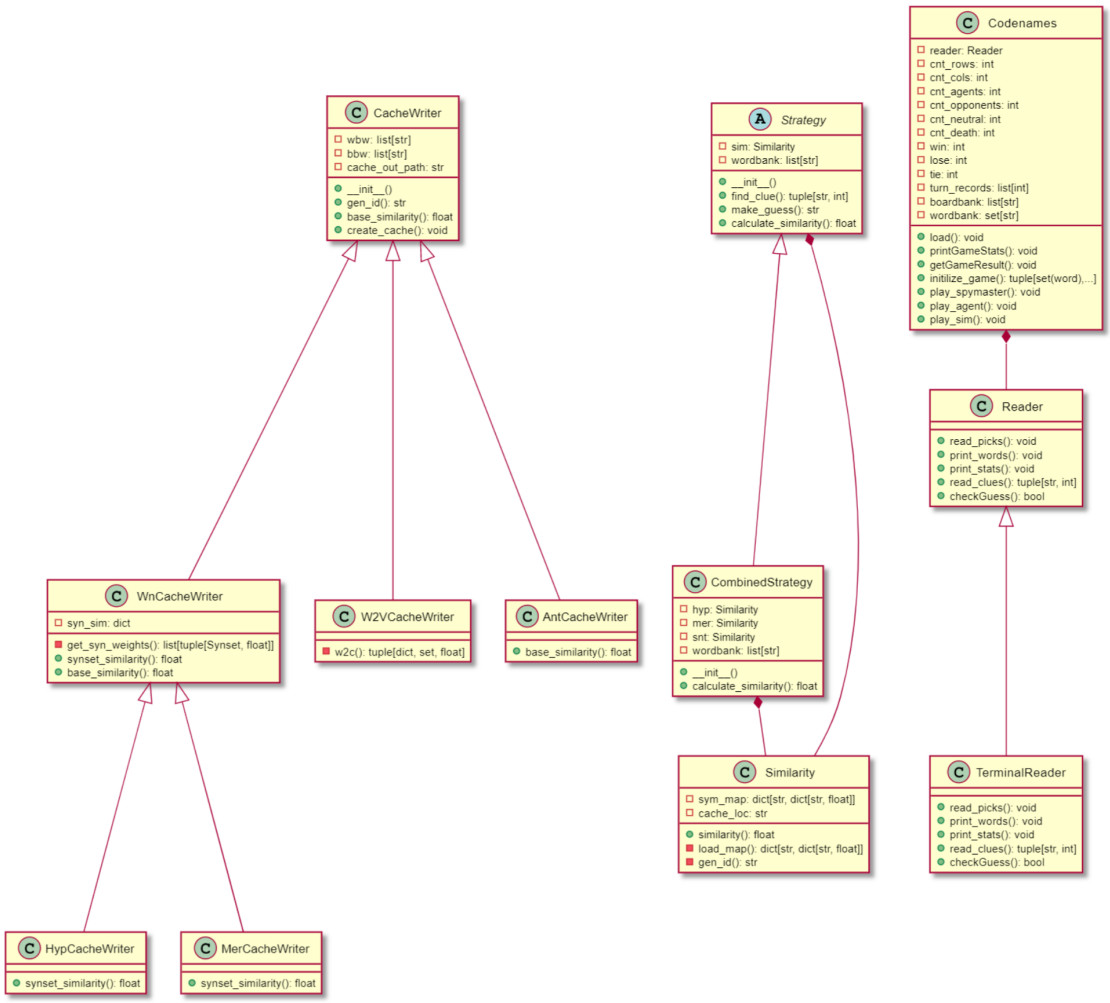

Figure 1. Uml diagram of the system architecture.

Similarities

Hypernym, Hyponym similarity

The hypernym-hyponym is the most simple and common relationship in the wordnet database. One word is a hypernym of another if it is a superclass of the other. For example, “Animal” is a hypernym of “Dog.” Similarly, “Dog” is a hyponym of “Animal.” Wordnet offers many metrics for path similarity that traverse the database along hypernym relationships. We use the metric that creates a score based on the shortest path between our two synsets of interest. This metric reports a 0 for very long paths, and a 1 if the two synsets are identical.

However, we are looking for the similarity between words, not synsets. We have tried various methods for how to calculate a similarity between two words given the similarities between their synsets, and we found a frequency-weighted approach to be ideal. We say that a pair of words could be interpreted as any pair of synsets they belong to. The probability that the pair gets interpreted as a specific pair of synsets can be modeled by the product of the probabilities that each word is interpreted as a specific synset. This probability is equal to the ratio of the frequency of the word in a particular synset over the sum of frequencies for all synsets. The computation is shown in Algorithm 1.

Algorithm 1 FrequencyWeghtedSimilarity

Input word1, word2
Output similarity

$\vec{\text{W1Synsets}} \leftarrow \vec{\text{Synsets}}(\text{word1})$
$\vec{\text{W2Synsets}} \leftarrow \vec{\text{Synsets}}(\text{word2})$
$\vec{\text{Frequencies1}} \leftarrow \vec{0}$
$\text{W1Total} \leftarrow 0$
for $i \leftarrow 1$ To $|\vec{\text{W1Synsets}}|$ **do**
  $\text{wordFreq} \leftarrow \text{W1Synsets}[i].\text{lemmas}(\text{word1}).\text{freq}()$
  $\text{Frequencies1}[i] \leftarrow \text{wordFreq}$
  $\text{W1Total} \leftarrow \text{W1Total} + \text{wordFreq}$
end for
$\vec{\text{Frequencies2}} \leftarrow \vec{0}$
$\text{W2Total} \leftarrow 0$
for $i \leftarrow 2$ To $|\vec{\text{W2Synsets}}|$ **do**
  $\text{wordFreq} \leftarrow \text{W2Synsets}[i].\text{lemmas}(\text{word2}).\text{freq}()$
  $\text{Frequencies2}[i] \leftarrow \text{wordFreq}$
  $\text{W2Total} \leftarrow \text{W2Total} + \text{wordFreq}$
end for
$\text{similarity} \leftarrow 0$
for $i \leftarrow 1$ To $|\vec{\text{W1Synsets}}|$ **do**
  for $j \leftarrow 2$ To $|\vec{\text{W2Synsets}}|$ **do**
    $w_{ij} \leftarrow \frac{Frequencies1[i] \cdot Frequencies2[j]}{W1Total \cdot W2Total}$
    $\text{WeightedSim} \leftarrow w_{ij} \cdot \text{SynsetSimilarity}(\vec{\text{W1Synsets}}[i], \vec{\text{W1Synsets}}[j])$
    $\text{similarity} \leftarrow \text{similarity} + \text{WeightedSim}$
  end for
end for
return similarity

Meronyms and Holonym

The meronym-holonym represents the part-whole relationship. The meronym of an object is a part of the object. For example, "tail"' is a meronym of "dog". On the other hand, the holonym of an object contains that object as a part. For example, "dog" is a holonym of "tail".
In the Wordnet database, the meronym-holonym relationship is the relationship between synsets -groups of words that share the same concepts. This relationship is broken down further into smaller groups. See groups reported by wordnet and examples for each group in Table 1.

Table 1: Description of Meronym-Holonym Relationship

	Groups	Examples
Meronym	Member Meronym	Animal is member meronym of zoo
	Part Meronym	Crown is path meronym of tree
	Substance Meronym	Sapwood is path meronym of tree
Holonym	Member Holonym	Court is member holonym of judge
	Part Holonym	Country is part holonym of country

	Substance Holonym	Candle is substance holonym of wax

In Wordnet, we can check if a synset has meronym-holonym relationship with another synset by checking if the second synset belongs to any meronym-holonym groups of the first synset. If a relationship exists, we report a similarity of 1. Otherwise we report a similarity of 0. Again, this score pertains to relationships between two synsets. Therefore we use Algorithm 1 to report a single meronym similarity for two words.

Antonyms

The last type of similarity we gather from Wordnet is Antonym similarity. Wordnet only defines antonyms for lemmas, which are the words in a synset. However, a word can have multiple lemmas in different synsets, so we still need to perform aggregation. We can record a 1 for the similarity if the lemmas of the words are antonyms, and a 0 if they are not. Then we can use Algorithm 1 to report the frequency-weighted antonym similarity.

Dict2Vec Embeddings

All we have to find the Dict2Vec similarity between two words is calculate the cosine similarity between their embeddings. Table 2 shows the most similar words for each type of embedding

Table 2: Top Word Pairs for Each Similarity.

	Hyponym/Hypernym	Meronym/Holonym	Antonym	Dict2Vec
1	attorney, lawyer	beach, shore	fair, unfair	automobile, car
2	lab, Laboratory	midnight, night	fire, hire	film, movie
3	automobile, Car	forest, tree	heavy, light	bomb, bombing
4	auto, Car	eye, lid	king, queen	gold, silver
5	instructor, Teacher	arm, wrist	cold, hot	change, changing
6	doctor, Physician	bed, bedroom	birth, death	orange, yellow
7	kid, Youngster	police, policeman	gross, net	king, queen
8	pupil, Student	calf, cattle	ill, well	dance, dancing
9	child, Kid	doorway, wall	dark, light	shooting, shot
10	film, Movie	hallway, wall	sleep, wake	garlic, ketchup

Utility Function

We use the utility function to search for a clue. For each possible clue, we find its similarity with all remaining agent, opponent, neutral, and assassin words. We pass these similarities to the utility function which returns a list of scores, $[e_1, e_2, , e_n]$, where e_i is the expected card differential when giving the clue with the number i. For example, correctly choosing two agent cards would correspond to a card differential of 2, whereas choosing one agent card and one opponent card would correspond to a card differential of 0. Therefore, the find_clue function in the strategy class gives the clue, number combination with the highest expected card differential.

The utility function estimates these scores by modeling the probability of the guesser choosing each card given the hint word. Given the similarity between a hint and a word, $\mathrm{sim}_{\mathrm{w}}$, a transformation function, f, and the set of all similarities on the board, $\mathrm{Sim}_{\mathrm{Board}}$, our model of the probability that the guesser will choose the word is given in Equation 1.

$$P(\mathrm{w}) = \frac{f(\mathrm{sim}_{\mathrm{w}})}{\sum_{s}^{(s,id)\in \mathrm{Sim}_{\mathrm{Board}}} f(s)} \tag{1}$$

The utility function can produce the expected value for a particular number of guesses by constructing a tree of possibilities. Each node in the tree represents a card on the board, and each edge in the tree corresponds to the probability of choosing the card. For example, say we have a board with two agent words, a_1, a_2, a neutral word n_1, an opponent word, o_1, and an assassin word, l. The utility would construct the tree in Figure 2.

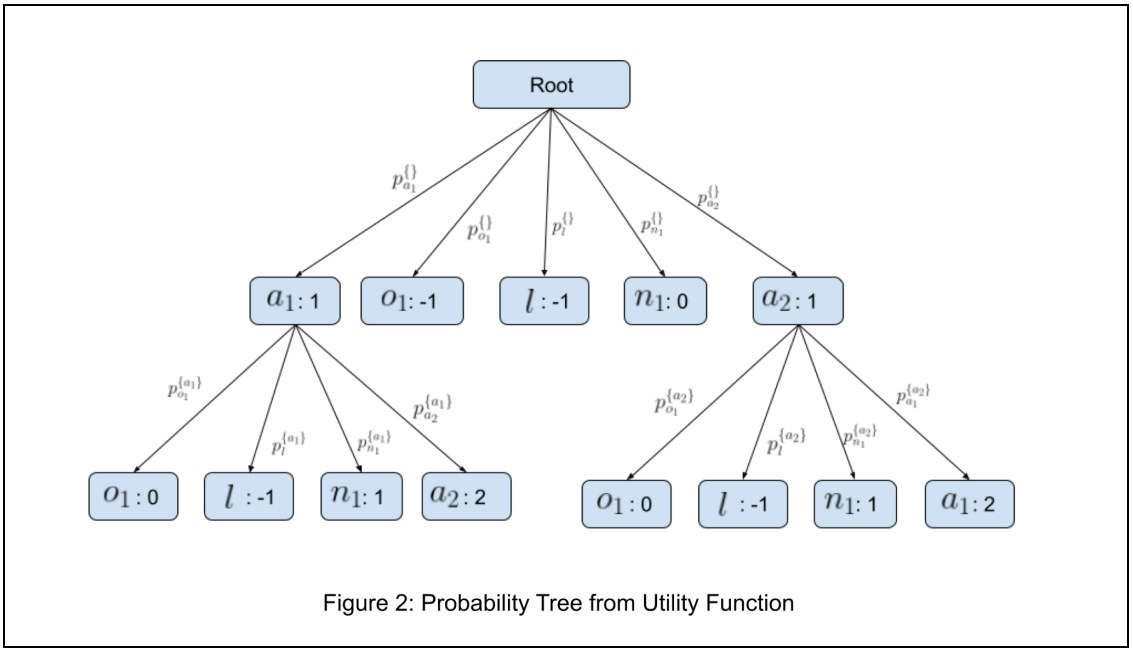

Figure 2: Probability Tree from Utility Function

The tree in Figure 2 uses the notation p_c^s where s denotes the set of agent cards already chosen in the turn. Note that one can find the expected card differential for the turn by following Algorithm 2. This algorithm multiplies the value of each leaf node by the product of all probabilities on the path leading to the leaf node. In computing the expected value, the same subtree will appear whenever the guesser chooses the same set of agent words, but in a different order. However, since we are taking the product of the probabilities associated with these guesses, their order does not matter. Thus, we cache the value result of subtrees which makes the time complexity $O(\frac{n!}{\lfloor \frac{n}{2} \rfloor ! \cdot \lceil \frac{n}{2} \rceil !})$ instead of $O(n!)$. Note that this function will need to be called once for every element in the list $[e_1, e_2, , e_n]$. This leads to some extra computation but the asymptotic complexity does not change. For our purposes, we limit the algorithm to only outputting scores for four guesses. The algorithm used to construct this tree of possibilities is listed as Algorithm 3 in the appendix.

Algorithm 2 ExpectedValue

```
Input ProbabilityTree, NumChoices
Output e_NumChoices
TreeId ← ProbabilityTree.GetId()
if MemoizedValues.Contains(TreeId) then
  return MemoizedValues.Get(TreeId)
end if
e_NumChoices ← 0
for Child in GetChildren(ProbabilityTree) do
  if Child is a LeafNode or NumChoices ≤ 1 then
    e_NumChoices += Child.Prob · Child.Value
  else
    e_NumChoices += Child.Prob · ExpectedValue(Child, NumChoices − 1)
  end if
end for
MemoizedValues.Put(TreeId, e_NumChoices)
return e_NumChoices
```

System Evaluation

Any Strategy in our Codenames simulation can be an agent, a spymaster, or both. These flexible mechanisms allow us to easily evaluate the performance of our bots in various conditions, including bot agent versus human spymaster, bot spymaster versus human agent, and bot agent versus bot spymaster.

5 -Analysis

Simulation

We used simulation as one medium of evaluation for our spymaster agents. We conducted these simulations using various transformation functions, f, in the utility function. Each f that we used was of the form $f(x) = e^{\sigma} x$ with inspiration from the softmax function.

We ran 15 simulations over 15 values for σ for each agent. When the WordNet agent was the spymaster, the Embedding agent was the guesser and vice versa. We show the results in Figure 3. In the figure, the reported score is calculated by Equation 2. We reasoned that a win was always better than a loss, a loss with more turns is better than a loss with fewer turns, and a win with fewer turns is better than a win with more turns. We also noted that the maximum number of turns in a game is 23. It is easy to see that the WordNet Spymaster with the

$$\text{Score}(\text{Result}, \text{Turns}) = \begin{cases} 24 - \text{Turns}, & \text{if Result} = \text{"win"} \\ \text{Turns} - 24, & \text{if Result} = \text{"lose"} \end{cases} \quad (2)$$

Embedding guesser was more performant than the Embedding Spymaster and the WordNet guesser. This result does not directly measure the performance of the spymasters because the guessers are variable. However, we can conclude that either the Wordnet Spymaster is superior to the Embedding Spymaster, or that the Embedding guesser is superior to the Wordnet guesser.

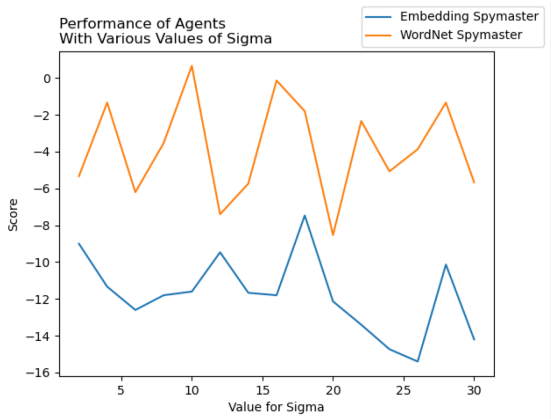

Figure 3: Simulations of Non-Twin Pairs

We also evaluated our agents with their twins. That is, we evaluated a Wordnet Spymaster with a Wordnet agent, and we evaluated an Embedding Spymaster with an Embedding agent. We expected that each of these duos would have a 100% win rate. However, the Wordnet pair only won in 8 of the 15 trials. The embedding pair did have a 100% win rate. The only reason a pair would not have a 100% win rate would be if the spymaster could not find a word that had a higher similarity amongst agent words than opponent and assassin words. This result indicates that with the same bank of words, the Embedding spymaster is able to explore more types of similarity than the Wordnet spymaster.The results of these evaluations are shown in Figure 4. We ran 15 games for each pair of agents. This 100% win rate is better than the best twin rate reported by Jaramillo et. al, and

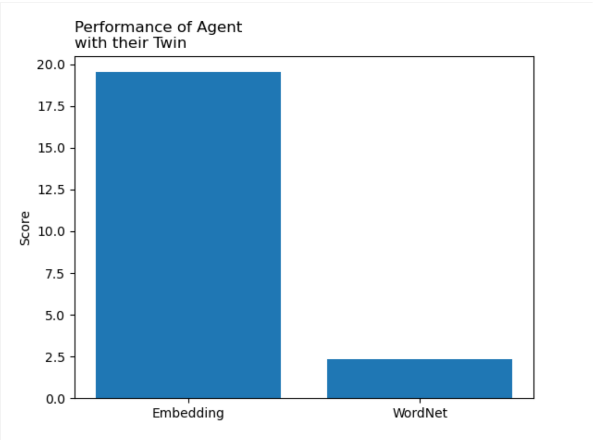

Figure 4: Performance of Agent with Twin

comparable to the twin win rates reported by Kim et. al [4], [5].

Human Evaluation

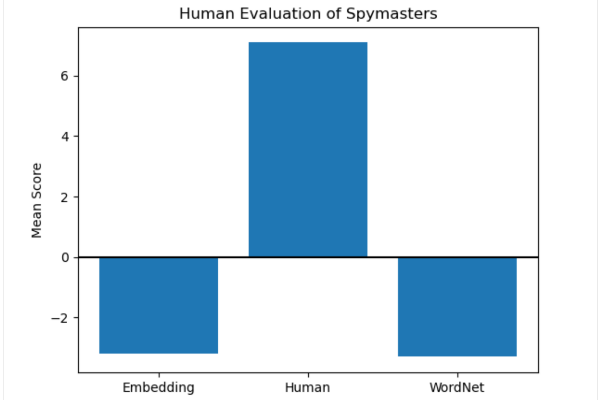

Figure 5. Human Evaluation of Spymasters

To directly compare the two spymaster agents, our group played a series of games against the autonomous agents. Before we played our games, we set $\sigma = 10$ for the WordNet agent and $\sigma = 18$ for the Embedding agent. These values of σ yielded the best results in our simulations. All together, we played 10 games with the Wordnet spymaster, 10 games with the Embedding spymaster, and 11 games with other humans. Figure 5 shows the results of these games. In our games, the human spymasters clearly performed the best with an average score of 7.09. We can roughly interpret this score by saying that on average, human spymasters won their games in 13.9 turns when playing with a human guesser. The two autonomous spymasters reported extremely similar results. The Embedding spymaster reported a score of -3.2, interpreted as losing in 20.8 turns on average, and the WordNet spymaster reported a score of -3.3 which is interpreted similarly. With only 10 trials for each spymaster, all of these results must be interpreted cautiously. We can be reasonably sure that human spymasters outperform the autonomous spymasters, but we cannot comment on the relative performance between the Wordnet, and Embedding spymasters.

Analysis of Results

Though the WordNet and Embedding agents showed similar performance in our human evaluation, their behavior as spymaster was wildly different. The WordNet spymaster was very precise. At each turn, it only associated its hint with one or two words, where the association between the hint, and its corresponding words was very interpretable. This is not surprising, as the WordNet agent based its measure of similarity on is-a, part-of, and antonym relationships. Conversely, the Embedding agent displayed a much looser sense of similarity. At each turn, the Embedding agent associated 2-4 words with its clue. The relationships between the clues from the Embedding agent, and the words they targeted are more contextual. It is more fun to play with the Embedding agent, because the guesser can play by their intuition instead of reasoning about particular words.

The difference between the Embedding agent's concept of similarity, and the Wordnet agent's concept of similarity helps explain the results from the simulation. Because the Embedding agent can recognize a broad set of relationships, it makes sense that it can reason about the

hints from the WordNet agent. Likewise, it makes sense that the WordNet agent may not be able to reason about some hints from the Embedding agent.

6- Link to code

Our implementation of codenames with our spymasters can be found in this github repository: https://github.com/dsatrawada/csc-481-codenames.

7- Summary

Conclusions

Although our limited human trials report similar results for the Wordnet, and Embedding spymasters, our intuition tells us that the Embedding spymaster is more effective. It obviously gives more natural clues. This result is common amongst previous work creating codenames agents [4], [5], [6]. However, we did learn that our loss function is effective by our comparison of twin win rates with the previous works. Our loss function is also more intuitive than the functions used in previous works as it is founded in probability.

Future Work

Our spymasters would benefit most from using a larger clue vocabulary. For example, when presented with the word Shakespeare, words like Romeo and Juliet would be clear and direct clues, as Romeo and Juliet is one of Shakespeare's most famous plays, but they do not exist in our vocabulary. Because of this, the Wordnet spymaster likely had no good words left to choose from towards the end of the game, resulting in poor clues. The embedding spymaster encountered this problem less, however, it would not hurt to have more words to choose from. The performance of the bot would decrease linearly with the number of words at runtime.

Additionally, if we wanted to enhance the bots, we could have them learn from our human trials. More specifically, the bots could keep note of which clues work and don't work for each of the words in our vocabulary. This way, the bot would likely be able to give better clues and yield better scores.

Also, more human evaluation could lead to a more reliable comparison between our spymasters. We only had 10 games for each pair which makes our results questionable. Similar to human evaluation, we believe it would be most effective to tune f on human gameplay. Tuning f based on the behavior of our autonomous guessers may give vastly different results than if we tuned f on human games.

8- References

1. Ahle, T. D. (2018). Codenames. Retrieved April 12, 2022, from https://github.com/thomasahle/codenames

2. Bird, S., Klein, E., & Loper, E. (2009). Natural language processing with Python: analyzing text with the natural language toolkit. " O'Reilly Media, Inc.".
3. Join me for a game of codenames. (n.d.). Retrieved April 28, 2022, from https://codenames.game/room/gum-charge-block
4. Jaramillo, C., Charity, M., Canaan, R., & Togelius, J. (2020, October). Word Autobots: Using transformers for word association in the game codenames. In Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (Vol. 16, No. 1, pp. 231-237).
5. Kim, A., Ruzmaykin, M., Truong, A., & Summerville, A. (2019, October). Cooperation and codenames: Understanding natural language processing via codenames. In Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (Vol. 15, No. 1, pp. 160-166).
6. Koyyalagunta, D., Sun, A., Draelos, R. L., & Rudin, C. (2021). Playing Codenames with Language Graphs and Word Embeddings. Journal of Artificial Intelligence Research, 71, 319-346.
7. Miller, G. A. (1998). WordNet: An electronic lexical database. MIT press.
8. Tissier, J. (2020). Improving methods to learn word representations for efficient semantic similarites computations (Doctoral dissertation, Lyon).
9. What is WordNet? (n.d.). Retrieved April 7, 2022, from https://wordnet.princeton.edu/
10. Word frequency data. (n.d.). Retrieved April 13, 2022, from https://www.wordfrequency.info/intro.asp

9- Appendix

Algorithm 3 shows the construction of the tree of possibilities used in the loss function. First the algorithm checks the cache to make sure it is not in a precomputed subtree. Then it recursively creates internal nodes corresponding to choosing each of the remaining agent words. It then adds leaf nodes corresponding to neutral, opponent, and assassin words.

Algorithm 3 BuildProbabilityTree

Input Root, f, Sim_a, Sim_n, Sim_o, Sim_l
Output ProbabilityTree
if MemoizedNodes.Contains(Root.GetId()) **then**
  return MemoizedNodes.Get(Root.GetId())
end if
$\text{Sim}_{\text{Board}} \leftarrow \text{Sim}_a \cup \text{Sim}_n \cup \text{Sim}_o \cup \text{Sim}_l$
for $(s, id) \in \text{Sim}_a$ **do**
  ChildProbability $\leftarrow$ ComputeProbability$(s, f, \text{Sim}_{\text{Board}})$
  ChildValue $\leftarrow$ Root.Value $+ 1$
  ChildId $\leftarrow$ Root.GetId() $- \{id\}$
  Child $\leftarrow$ TreeNode(ChildProbability, ChildValue, ChildId)
  $\text{Remaining}_a \leftarrow \text{Sim}_a - (s, id)$
  BuildProbabilityTree(Child, Remaining_a, Sim_b, Sim_n, Sim_o, Sim_l)
  Root.AddChild(Child)
end for
for $(s, id) \in \text{Sim}_n$ **do**
  ChildProbability $\leftarrow$ ComputeProbability$(s, f, \text{Sim}_{\text{Board}})$
  ChildValue $\leftarrow$ Root.Value
  Child $\leftarrow$ LeafNode(ChildProbability, ChildValue)
  Root.AddChild(Child)
end for
for $(s, id) \in \text{Sim}_o$ **do**
  ChildProbability $\leftarrow$ ComputeProbability$(s, f, \text{Sim}_{\text{Board}})$
  ChildValue $\leftarrow$ Root.Value $- 1$
  Child $\leftarrow$ LeafNode(ChildProbability, ChildValue)
  Root.AddChild(Child)
end for
for $(s, id) \in \text{Sim}_l$ **do**
  ChildProbability $\leftarrow$ ComputeProbability$(s, f, \text{Sim}_{\text{Board}})$
  ChildValue $\leftarrow$ Root.Value $- |\text{Sim}_o|$
  Child $\leftarrow$ LeafNode(ChildProbability, ChildValue)
  Root.AddChild(Child)
end for
MemoizedNodes.Put(NodeId, Root)