Picking the Best Code: A *CodeNames* Clue Generator Using NLP

Ryan Hood
Student ID: 2021493755
Email: ryan.hood@utdallas.edu
Department: ECSS

Shanjida Khatun
Student ID: 2021535679
Email: shanjida.khatun@utdallas.edu
Department: ECSS

Abstract (Ryan Hood: 0%, Shanjida Khatun: 100%)

Codenames is a game that requires deep, multi-modal language understanding. Each team of two teams split into a code giver and code guesser, and the guessers must determine which of 25 possible words on the board correspond to the clue. The nature of the game requires understanding language in a multimodal manner – e.g., the clue 'cold' could refer to temperature or disease. Although board games and video games have been studied for decades in artificial intelligence research, challenging word games remain relatively unexplored. Word games are not as constrained as games like chess or poker. Instead, word game strategy is defined by the players' understanding of the way words relate to each other. We proposed an algorithm that can generate *Codenames* clues from several embedding methods such as word2vec, GloVe. We also introduced a scoring function that measures the quality of clues to improve clue selection. We also developed certain functions to improve clue quality and overcome the computational barriers.

1 Introduction (Ryan Hood: 25%, Shanjida Khatun: 75%)

Codenames is a task that is difficult even for humans, who sometimes struggle to generate clues for boards. As with all games, the element of difficulty is necessary to produce an exciting challenge. *Codenames* rely on a deep understanding of language. Traditional language tasks often focus on one axis of language understanding such as analogies or part-of-speech tagging, while *Codenames* requires leveraging many different axes of language, including common sense relationships between words. The Motivation behind this project is the word game *Codenames* provides a unique opportunity to investigate common sense understanding of relationships between words, an important open challenge.

Codenames is a word game for 4 people split into 2 teams of 2 people each (red team and blue team). Each team is split into a code giver and code guesser. On the table is a 5x5 grid of words such that all players can view the words. The code givers have access to a special card which identifies all the 25 cards on table as either red card, blue cards, black cards (known as assassins), and tan cards (known as civilians). The goal of the red team, say, is to have their code guesser correctly guess all the red cards in the middle before the blue team's code guesser guesses all the blue cards. Guesses occur after a word is given by a code giver. The code giver, when it is their team's turn, provides their player a single word and a number. The word should relate to as many of their team's words as possible, and the number is how many words the code giver intended to be related. Whatever number is provided, the code guesser can only guess that number +1 times. For example, if "Brick 3" is said, then the code guesser can guess up to 4 words. The team that goes first must guess 9 words, and the second team has to guess 8 words. If a code guesser ever guesses the assassin card, that team loses immediately. There is no penalty for guessing civilians except losing one of the team's potential guesses.

An artificial intelligence solution to this game will require the use of NLP to come up with the best words. One way is through the idea of word embeddings. Word embeddings are a mapping between words and a high dimensional vector such that words close in semantic meaning are close together in the high dimensional space.

2 Related Work (Ryan Hood: 0%, Shanjida Khatun: 100%)

Game competitions have been a popular testbed for a variety of AI techniques. These have ranged from strategy games in the Starcraft AI Competition to first-person shooters in the VisDoom competition that is held in 2018. More recently, there has been interest in games that require either textual understanding, cooperation, or epistemic reasoning in the Text-Based AI Competition (Atkinson et al.2018), The Hanabi Competition (Canaan et al. 2018), and One Night Ultimate Werewolf (Eger and Martens 2018) respectively. The recently announced *Codenames* AI competition requires textual understanding, cooperation, and epistemic reasoning. The code master and guesser need to understand the words being presented to them, they need to work towards their common goal, and perhaps most importantly they need to understand what their partner understands, so that they can successfully cooperate.

Work by Bard et al. used reinforcement learning to train Hanabi playing bots. They found that bots that had a paired partner were able to play nearly perfectly, but when paired with a partner that did not have the same communication strategy the bots performed horribly, with nearly no chance of winning, demonstrating the challenge of working with unknown teammates when no external communication is allowed. While, to our knowledge, this work is the first to compare different word similarity and word embedding approaches in the context of games, this is not the first work to compare them in other natural language processing contexts.

Naili et al. compared Word2Vec, GloVe, and LSA vectorial semantic approaches in the domain of topic segmentation finding that both GloVe and Word2Vec outperform LSA (Naili, Chaibi, and Ghezala 2017).

Ruckl et al. compared GloVe, Word2Vec, and concatenated combinations thereof in several different contexts, Argumentation Mining, Sentiment Classification, Opinion Polarity, and Question-Type Classification. They found that concatenations of GloVe and Word2Vec worked the best, meaning that while the approaches are similar, they each have relative merits that can improve upon each other. In his master's thesis Handler uses WordNet to assess the types of relationships found by word2vec, finding that word2vec finds more similarity between synonyms, hypernyms, and hyponyms ahead of meronyms and holonyms (e.g., it finds more similarity between 'wheels' and 'tires' than 'wheels', and 'car'). However, none of these approaches assess how well the different approaches agree with each other in terms of similarity which our work addresses.

Kim et al. (2019), who proposed an approach for *Codenames* clue-giving that relies on word embeddings to select clues that are related to board words. They evaluated the performance of word2vec and GloVe *Codenames* clue-giver bots by pairing them with word2vec and GloVe guesser bots. Although this evaluation approach is easy to run repeatedly over many trials, as it is purely simulation-based, the evaluation is limited to how well the clue-givers and guessers "cooperate" with one another. "Cooperation" measures how well GloVe and word2vec embedding representations of words are aligned on similarity or dissimilarity of given words. To be more explicit, two methods with different embeddings "cooperate well" if word embeddings that are relatively close based on the clue-giver's embeddings are also close based on the guesser's embeddings, and vice versa. As a result, perfect performance (100%-win percentage) comes from pairing a clue-giver and a guesser who share the same embedding method. However, this "cooperation" metric does not evaluate whether a clue given by a clue-giver is a good clue, that is, a clue that would make sense to a human.

Kim et al. also explored the use of knowledge graphs for *Codenames* clue-giving, but ultimately did not consider knowledge-graph-based clue-givers in their final evaluation due to poor qualitative performance and computational expense.

In contrast, Divya et al. proposed a method for an interpretable knowledge-graph-based clue-giver that performs competitively with embedding-based approaches. The knowledge-graph-based method has a clear advantage in interpretability over the embedding-based approaches.

Jaramillo et al. (2020) compared the baseline word2vec and GloVe word representations with versions using TF-IDF values, classes from a naive Bayes classifier, or nearest neighbors of a GPT-2 Transformer representation of the concatenated board words. Like Kim et al. (2019), they evaluated their methods primarily by pairing clue-giver bots with guesser bots. They included an initial human evaluation, where 10 games were played for both the baseline (word2vec and GloVe) and the Transformer representations as clue-giver and guesser, but the human evaluation is limited to only 40 games. Again, since evaluations from bots may not represent human judgments, our human evaluation is more realistic and extensive, conducted through Amazon Mechanical Turk with 1,440 total samples.

In 2019, Zunjani and Olteteanu proposed a formalization of the *Codenames* task using a knowledge graph but did not provide an implementation of their proposed recursive traversal algorithm. We found that recursive traversal does not scale to the computation required to run repeated evaluations of *Codenames* for each blue word, we must find each associated word *w* in the knowledge base that has *Association(w,b)* greater than some threshold *t*, and repeat this process every trial. In BabelNet, because each word may be connected to tens or hundreds of other words, this becomes unscalable when traversing more than one or two levels of edges.

In 2018, Shen et al. also proposed a simpler version of the *Codenames* task with human evaluation. The experimental setup focuses on comparing different semantic association metrics, including a knowledge-graph based metric. Their task differs from ours in two keyways. First, each of their trials considers three candidate clues drawn from a vocabulary of 100 words, whereas Divya et al. considered candidate clues drawn from the larger vocabulary of all English words. Second, their usage of ConceptNet is different from our usage of BabelNet because they use vector representations derived from an ensemble of word2vec, GloVe, and ConceptNet using retrofitting whereas Divya et al. leveraged the graph structure of BabelNet.

In 2021, Divya et al. proposed an algorithm that can generate *Codenames* clues from the language graph BabelNet or from any of several embedding methods - word2vec, GloVe, fastText or BERT. They introduced a new scoring function that measures the quality of clues, and we propose a weighting term called DETECT that incorporates dictionary-based word representations and document frequency to improve clue selection. They developed BabelNet-Word Selection Framework (BabelNetWSF) to improve BabelNet clue quality and overcome the computational barriers that previously prevented leveraging language graphs for *Codenames*. Extensive experiments with human evaluators demonstrate that our proposed innovations yield state-of-the-art performance, with up to 102.8% improvement in precision@2 in some cases. Overall, this work advances the formal study of word games and approaches for common sense language understanding.

3 Preliminaries (Ryan Hood: 75%, Shanjida Khatun: 25%)

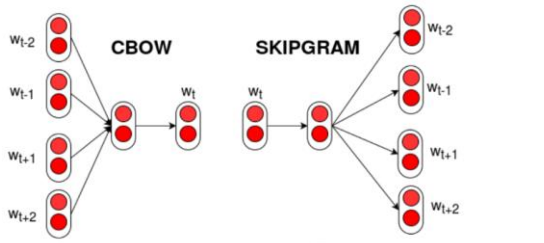
3.1 Word Embeddings

The concept of word embeddings will be very important for this use case. A word embedding is a mapping between a word and a high dimensional vector such that the distances between two vectors relate to the semantic similarity of their corresponding words. For example, the vectors corresponding to "car" and "truck" will be close together in embedding space. Word arithmetic can also be done. For example, if we take the vector for "king", subtract the vector for "man" from it and then add the vector for "woman" to it, we will get a vector close to the vector for "queen". This also allows us to find vectors for phrases by taking the sum of all the vectors for each word in the phrase. To determine the distance between two vectors, we use the cosine similarity.

$$similarity(A,B) = \frac{A \cdot B}{\|A\| \times \|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Finding word embeddings is an unsupervised problem where the models are given enormous amounts of text from which associations can be learned. Now we will talk about the more specific categories of word embeddings.

The first category of word embeddings we will consider is word2vec. Word2vec can either use a skip-gram architecture or CBOW architecture. The skip-gram architecture traverses through the training corpus feeding the neural network each word with the goal of predicting the neighboring words. Once the model has trained for long enough, the actual embedding for a given word is found by giving the model a word and observing the values at the hidden layer. The CBOW architecture works in the reverse direction. Neighboring words are fed into the model and the word itself is predicted by the model. Again, the values at the hidden layers can serve as the word embeddings.



$$\frac{1}{T}\sum_{t=1}^{T} \sum_{-c\leq j\leq c, j\neq 0} log p(w_t|w_{t+j}) \qquad \frac{1}{T}\sum_{t=1}^{T} \sum_{-c\leq j\leq c, j\neq 0} log p(w_{t+j}|w_t)$$

3.2 Global Vector Embeddings

The next category of word embeddings we will discuss is GloVe embeddings. GloVe embeddings are constructed from a very large word-context matrix. Conditional probabilities are calculated for each row, and the ratio of these probabilities can be used to derive meanings of concepts. In the below example, "solid" is strongly related to "ice" but not "steam" resulting in a large probability ratio. The opposite is true for the word "gas" resulting in a very small value of the probability ratio. Words indifferent to the words on the rows (such as "water" which is related to both and "fashion" which is related to neither) will result in probability ratios close to 1. Ultimately, these relationships between words can be used to find word embeddings for all the words on the rows.

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

GloVe is trained by a linear regression that tries to learn weights such that the weights associated with a word try to predict the log of the co-occurrence counts of the word and its contexts:

$$\sum_{i,j=1}^{V} (w_i + b_i - logC_{ij})$$

where *w*, *b* are the weights and bias associated with word *i*, C_{ij} is the co-occurrence count for word *i* and word *j*, and *V* is the size of the vocabulary.

3.3 BERT Embeddings

Another incredibly popular word embedding approach is BERT. BERT models tend to perform better than word2vec and GloVe models in most applications. But this approach is not helpful for our use case since BERT word embeddings are context dependent. This means if a word has multiple meanings, it will have a vector for each meaning. This is useful if we can determine the meaning of a word from context, but we do not have any context in *Codenames*. Each word is a standalone and there is no sentence to provide this context. Even if there was a sentence, the use of BERT embedding would be counterproductive since the best code words often rely on words having multiple meanings. Because of these reasons, we do not use BERT word embeddings in this project.

3.4 Word2vec vs. GloVe

At a high level, GloVe embeddings look at word occurrence at a global level while word2vec looks at word occurrence at a local level. In most use cases, word2vec embeddings and GloVe embeddings perform similarly despite the very different process for creating them. Usually, the dimensionality of the vectors and the corpus used to train are the two biggest factors affecting the performance of the word embeddings.

4 Our Approaches (Ryan: 100%, Shanjida: 0%)

We aim to create an artificial intelligence system that can compare two word embedding models and determine the best model for playing *Codenames*. There are various approaches to accomplish this, each with their own pros and cons.

The first approach is to use four different models, two being code givers and the other two being guessers. This is problematic because if one of the guesser models is stronger than the other, it will provide an unfair advantage to its partner model.

The second approach is to use two different models as code givers but use a third model as the guesser for both the red and blue team. This fixes the issues with the first approach, but it introduces another issue. The better code giver model will end up being the one that more closely aligns with the shared guesser model. That is, the guesser model is supposed to be neutral, but it will be very hard to ensure this is the case.

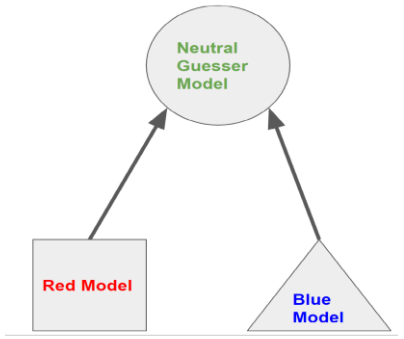
The final approach is to use one model to serve as the red code giver and a different model to serve as the blue code giver, but the code words will always be interpreted by a human. This is the most accurate, as it does not suffer the issues the previous two approaches had. The downside is that it is burdensome for a human to manually interpret all the code words.

We decided to go with the second approach.

5 Implementation (Ryan Hood: 50%, Shanjida Khatun: 50%)

High Level View:

For the red team's model, we use a word2vec model built from GoogleNews pages. For the blue team's model, we use a GloVe model built from Gigaword 5th Edition, and our neutral guesser model was created from gensim's FastText on Wikipedia news articles.



Low Level View:

Our implementation involves 5 total files: Board.py is a class that is responsible for setting up the board including the N x N grid in the middle but also giving those cards their identities. *Codenames*.py is a class that is responsible for playing a complete

Codenames game. Apart from these two key classes, we have a words.txt file which holds all the words that can possibly be chosen to be words in a *Codenames* game. We have a play_games.py file which is responsible for playing a bulk number of games and calculating the statistics of those games.

Script 1 playGames *(numGames, models)*

```
1. for 0 to numGames do
2.   ρ = Codenames (models, modelsType)
3.   π= ρ.board.Firstplayer
4.   σ = ρ.playFullGame ()
5.   π.add(π)
6.   σ.add (σ)
7. return π, σ
```

Above is pseudocode for the playGames class which sits in the *Codenames* class. While the game is not yet over, a code word is found by either the red code giver model or the blue code giver model. Then, the guesser model will pick words from the middle of the board that most closely match the given code word. These guesses are then removed from the current board state. Lastly, the turns change, and the while loop repeats.

Script 2 Codenames (*models, modelsType*)

```
1. $\pi_c$ = getResultSet(codeWords)
2. getScores ($\pi_c$)
3. getValidWord (sortedTuples)
4. getCodeword (codeWords)
5. pickWords (codeWord, intendedMatches)
6. updateBoard (guessedWords)
7. playFullGame ()
```

We can see those lines 4 and 5 in the Script 2 do the heavy lifting of this program and so we will elaborate more on those now. Line 4 is responsible for taking the current board state and returning the best possible code word, and so it plays the role of the code giver. To do this, we take advantage of another method most_similar(). This method takes an embedding model, a list to try to match, and a list to try to match against.

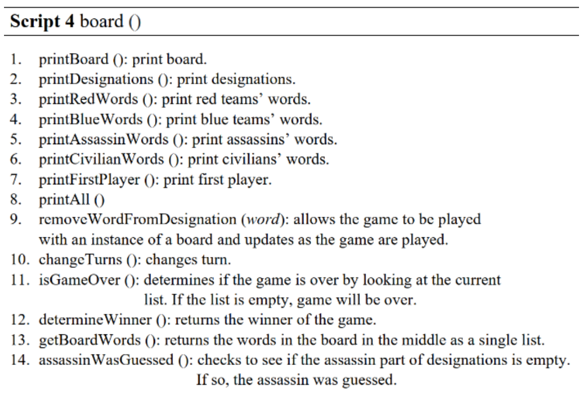
In the above example, we play the role of the red code giver, so we want to match to the red code words, but we do not want to match to the blue code words (or else we risk the guesser scoring points for the blue team). The result is a list of tuples. Each tuple is made of a word and a number; the word is a potential code word, and the number being a score that represents the quality of the match. After we perform some usual NLP cleanup on the result set, we sort these results by the score. If we only sorted by this score, our program would not perform to its potential. This is because the resulting code word would strongly match 1 or 2 words from the middle, and not match any of the other words. We would prefer if our code word loosely matched many words from the middle of the board. This would allow us to maximize the number of words we can score during our turn. So, we implemented our own scoring system which finds the code word from the initial result set that maximizes the number of loose matches to desired words from the middle of the board. We also form a bad set of words, which are words that relate to the assassin word. We perform set subtraction to ensure that we do not use a code word that is present in the bad set. Line 5 is responsible for picking words from the middle of the board and is done by the guesser model. The guesser model will take the code word and the number of intended matches, and it will find the similarity score between the code word and every word from the middle of the board. The top matches are returned. For example, if the guesser is given a code word of "car" with 5 intended matches, then it will compare "car" with every word from the middle and return the top 5 matches.

Script 3 helperMethods ()

```
1. keepSingleWords (resultSet): removes words in the result set.
2. setLowercase (resultSet): makes every word lowercase.
3. removeRepeats (resultSet): removes repeated words in the result set.
4. removePluralCopies (resultSet): removes the plural copies from the result set.
5. removeBoardWords (allBoardWords, resultSet): removes the plural version of
                                   words on the board from the result set.
6. removeBadWords (resultSet, badSet): removes words in the bad set from our result set.
7. fullPipeline (allBoardWords, resultSet, badSet): combines every pipeline step into
                                   one to refine our result set.
8. removeUsedCodeWords (tupleSet, usedCodeWords): removes tuples in the tuple
      set where the word (the first element of the tuple) is in the used code words.
```

We also have helper_methods.py which mainly contains methods to remove stop words, duplicates by tense (such as run and running), etc.

In the below script, this allows someone to create an initial instance of a Codenames board and designations and update it as the game is played. The class includes printing methods along with a method to update the class variables during a Codenames game. First initialize our basic class variables from provided parameters. If a class variable is appended with 'initially', then that class variable corresponds to the initial state of the board which does not change over time. If the class variable is appended with 'currently', then that class variable corresponds to the current state of the board, which will change as words are guessed.

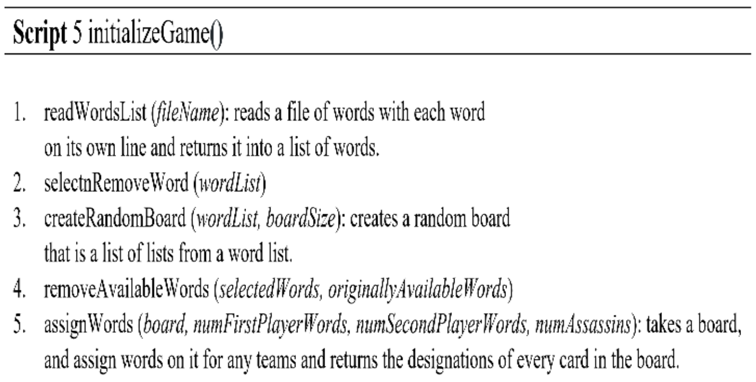


Script 4 board ()

1. printBoard (): print board.
2. printDesignations (): print designations.
3. printRedWords (): print red teams' words.
4. printBlueWords (): print blue teams' words.
5. printAssassinWords (): print assassins' words.
6. printCivilianWords (): print civilians' words.
7. printFirstPlayer (): print first player.
8. printAll ()
9. removeWordFromDesignation (*word*): allows the game to be played with an instance of a board and updates as the game are played.
10. changeTurns (): changes turn.
11. isGameOver (): determines if the game is over by looking at the current list. If the list is empty, game will be over.
12. determineWinner (): returns the winner of the game.
13. getBoardWords (): returns the words in the board in the middle as a single list.
14. assassinWasGuessed (): checks to see if the assassin part of designations is empty. If so, the assassin was guessed.

Below script initializes the game. A game is initialized if the board is set, both spymasters know the identities of all their words, the assassin, and the civilians, and they know who is going first. This returns the board (a list of lists), a dictionary with the designations of all cards, and a string detailing who should go first: red team or blue team.

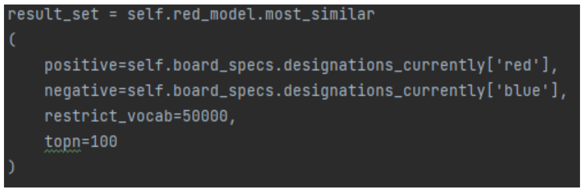
Two parts of the implementation deserve special attention. The first is the process of getting the code words using either the red code giver model or the blue code giver model. Line 1 of script 2 is responsible for taking the current board state and returning the best possible code word. To do this, we take advantage of another method most_similar(), which has the concept of cosine similarity built in.



Script 5 initializeGame()

1. readWordsList (*fileName*): reads a file of words with each word on its own line and returns it into a list of words.
2. selectnRemoveWord (*wordList*)
3. createRandomBoard (*wordList, boardSize*): creates a random board that is a list of lists from a word list.
4. removeAvailableWords (*selectedWords, originallyAvailableWords*)
5. assignWords (*board, numFirstPlayerWords, numSecondPlayerWords, numAssassins*): takes a board, and assign words on it for any teams and returns the designations of every card in the board.

This method takes an embedding model, a list to try to match, and a list to try to match against. In the below example, we play the role of the red code giver, so we want to match to the red code words, but we do not want to match to the blue code words (or else we risk the guesser scoring points for the blue team).



```
result_set = self.red_model.most_similar
(
    positive=self.board_specs.designations_currently['red'],
    negative=self.board_specs.designations_currently['blue'],
    restrict_vocab=50000,
    topn=100
)
```

The result is a list of tuples. Each tuple is made of a word and a number; the word is a potential code word, and the number being a score that represents the quality of the match. After we perform some usual NLP cleanup on the result set, we sort these results by the score. If we only sorted by this score, our program would not perform to its potential. This is because the resulting code word would strongly match 1 or 2 words from the middle, and not match any of the other words. We would prefer if our code word loosely matched many words from the middle of the board. This would allow us to maximize the number of words we can score during our turn. So, we implemented our own scoring system which finds the code word from the initial result set that maximizes the number of loose matches to desired words from the middle of the board. Lines 2, 3, and 4 of script 2 perform this process. We also form a bad set of words, which are words that relate to the assassin word. We perform set subtraction to ensure that we do not use a code word that is present in the bad set.

The next part of the implementation that deserves special attention is line 5 of script 2 which is responsible for picking words from the middle of the board and is done by the guesser model. The guesser model will take the code word and the number of intended matches, and it will find the similarity score between the code word and every word from the middle of the board. The top matches are returned. For example, if the guesser is given a code word of "car" with 5 intended matches, then it will compare "car" with every word from the middle and return the top 5 matches.

6 Experimental Results (Ryan Hood: 50%, Shanjida Khatun: 50%)

6.1 Datasets

For word2vec, we have used a pre-trained Google News corpus (3 billion running words) word vector model (3 million 300-dimension English word vectors). It is mirroring the data from the official word2vec website: GoogleNews-vectors-negative300.bin.gz.

For GloVe, we have used a pre-trained "*glove.6B.100d.txt*" dataset and then we have converted it to the word2vec model. The dataset can be found from this site: https://www.kaggle.com/danielwillgeorge/glove6b100dtxt.

For guesser model, we have used a pre-trained word vectors "*wiki-news-300d-1M.vec*" which is million-word vectors trained on Wikipedia 2017, UMBC web-based corpus and statmt.org news dataset (16B tokens). This dataset can be found from this site: https://fasttext.cc/docs/en/english-vectors.html.

6.2 Results and Analysis

Based on manual single runs of the program, the usefulness of the word embedding approach is evident. Some code words can make connections to multiple different ideas and would be difficult for a human to come up with. For example, the word tele healthcare can connect to ideas in telecommunications as well as medicine. Oftentimes, human code givers focus on one of the ideas and don't see a way to connect both in a single code word.

In 1000 games, our word2vec model won 614 games compared to 386 games by our GloVe model. It also looks like the first player has a significant advantage winning 624 out of 1000 games. This makes sense since if the first player wins the game would end with the first player having an additional turn. If the second team wins, then both teams will have the same number of turns. The creator of *Codenames* tried to balance the game by forcing the team who goes first to score one additional work (they start with 9 words to guess; the second team starts with 8). Based on our findings, the first player still has an advantage. This is consistent with our observations of playing in an entirely human setting.

7 Conclusion (Ryan Hood: 0%, Shanjida Khatun: 100%)

In this project, we have detailed two different teams for the *Codenames*– a *red team* and a *blue team*– and have analyzed these both with a wide variety of Natural Language Processing backends, ranging from classical knowledge-based approaches deriving from WordNet and from modern machine-learned vectorial semantics approaches, word2vec and GloVe. We analyzed these frameworks and the guesser – to see how well these different approaches can cooperate with each other. We found that the vectorial semantics-based approaches universally worked well with each other while the WordNet based approach universally performed less with the vector-based approaches.

In the future, we will develop our own models that would allow us to compare the performance of models trained on different corpuses of text. We will figure out a better way of comparing models. Currently, the better model is just the one that is closer to the guesser in its embedding space. It varies the word embedding dimension. We will also develop visuals to display results.

Bibliography

1. Ashktorab, Z., Dugan, C., Johnson, J., Pan, Q., Zhang, W., Kumaravel, S., & Campbell, M. (2021). Effects of communication directionality and AI agent differences in humanAI interaction. In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, CHI ’21. Association for Computing Machinery.
2. Jaramillo, C., Charity, M., Canaan, R., & Togelius, J. (2020). Word Autobots: Using transformers for word association in the game codenames. In Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 16(1), pp.231-237.
3. Kim, A., Ruzmaykin, M., Truong, A., & Summerville, A. (2019). Cooperation and codenames: Understanding natural language processing via codenames. In Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment,15(1), pp. 160-166.
4. Zunjani, F. H., & Olteteanu, A.-M. (2019). Towards reframing codenames for computational modeling and creativity support using associative creativity principles. In Proceedings of the 2019 on Creativity and Cognition, p. 407-413, New York, NY, USA. Association for Computing Machinery.
5. Shen, J. H., Hofer, M., Felbo, B., & Levy, R. (2018). Comparing models of associative meaning: An empirical investigation of reference in simple language games. In Proceedings of the 22nd Conference on Computational Natural Language Learning, pp.292-301, Brussels, Belgium. Association for Computational Linguistics.
6. Atkinson, T.; Baier, H.; Copplestone, T.; Devlin, S.; and Swan, J. 2018. The Text-Based Adventure AI Competition. *arXiv preprint arXiv:1808.01262*.
7. Bard, N.; Foerster, J. N.; Chandar, S.; Burch, N.; Lanctot, M.; Song, H. F.; Parisotto, E.; Dumoulin, V.; Moitra, S.; Hughes, E.; et al. 2019. The hanabi challenge: A new frontier for AI research. *arXiv preprint arXiv:1902.00506*
8. Canaan, R.; Shen, H.; Torrado, R.; Togelius, J.; Nealen, A.; and Menzel, S. 2018. Evolving agents for the hanabi 2018 SIG competition. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. IEEE.
9. Eger, M., and Martens, C. 2018. Keeping the story straight: A comparison of commitment strategies for a social deduction game. In the Fourteenth *Artificial Intelligence and Interactive Digital Entertainment Conference*.
10. Naili, M.; Chaibi, A. H.; and Ghezala, H. H. B. 2017. Comparative study of word embedding methods in topic segmentation. *Procedia computer science* 112:340–349.
11. Ruckl ¨ e, A.; Eger, S.; Peyrard, M.; and Gurevych, I. 2018. ´Concatenated power mean word embeddings as universal cross-lingual sentence representations. *arXiv preprint arXiv:1803.01400*.