

# **AI Codenames**

**Adam Purcell**

Final-Year Project – BSc in Computer Science

Dr. Derek Bridge

Department of Computer Science  
University College Cork

April 2022

# Abstract

Codenames is a board game that is all about finding relationships between words. In natural language processing, word embedding is a term used for the representation of words for text analysis, typically in the form of a real-valued vector that encodes the meaning of the word such that the words that are closer in the vector space are expected to be similar in meaning. The goal of this project was to build a web-based application which allows users to play an online implementation of Codenames in real-time over the internet, and to implement an AI player which uses word embeddings such as Word2Vec and GloVe to play the game.

# Declaration of Originality

In signing this declaration, you are confirming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

**Signed:** .....

**Date:** .....

# Acknowledgements

Thanks to Dr. Derek Bridge for being my supervisor during the course of this project. His guidance and support throughout the year was extremely beneficial. Another thanks to any of my friends and family who participated in the user trials during the evaluation of the project.

# Contents

<b>1</b>	<b>Introduction</b>	<b>9</b>
<b>2</b>	<b>Background</b>	<b>12</b>
<b>3</b>	<b>Design</b>	<b>14</b>
3.1	Architecture . . . . .	14
3.2	Models . . . . .	16
3.3	User Interface . . . . .	17
3.4	User Interaction . . . . .	18
3.5	Technologies . . . . .	19
<b>4</b>	<b>Implementation</b>	<b>23</b>
4.1	Clue Area . . . . .	28
4.2	Game Board . . . . .	29
4.3	Chat Area . . . . .	30
4.4	Game Stats . . . . .	31
<b>5</b>	<b>AI Spymasters</b>	<b>32</b>
5.1	Analysis . . . . .	32
5.1.1	Word Embeddings . . . . .	32
5.1.2	word2vec . . . . .	32
5.1.3	GloVe . . . . .	36
5.2	Related Work . . . . .	39
5.3	Design . . . . .	40
5.4	Implementation . . . . .	41
<b>6</b>	<b>Evaluation</b>	<b>44</b>

6.1	Related Work . . . . .	44
6.2	Methodology . . . . .	45
6.2.1	Models . . . . .	45
6.2.2	User Trial Phase 1 . . . . .	47
6.2.3	User Trial Phase 2 . . . . .	48
6.3	Implementation . . . . .	49
6.4	Results . . . . .	50
6.4.1	User Trial Phase 1 . . . . .	50
6.4.2	User Trial Phase 2 . . . . .	51
<b>7</b>	<b>Conclusions and Future Work</b>	<b>54</b>
7.1	Conclusions . . . . .	54
7.2	Future Work . . . . .	55
<b>A</b>	<b>Appendix</b>	<b>57</b>

# List of Figures

2.1	Codenames board . . . . .	13
3.1	Multiplayer game with a client-server architecture . . . . .	14
3.2	UML diagram showing relationship between players, teams and games . . . . .	16
3.3	Wireframe for game area . . . . .	17
3.4	Flowchart for user interaction . . . . .	18
4.1	Landing page . . . . .	23
4.2	Create/Join game view . . . . .	24
4.3	UML diagram of <b>Game</b> class . . . . .	25
4.4	Join game form . . . . .	26
4.5	Initial view for spymaster . . . . .	26
4.6	Initial view for operative . . . . .	27
4.7	Mid-game view for spymaster . . . . .	27
4.8	Mid-game view for operative . . . . .	28
4.9	Invalid clue pop-up . . . . .	28
4.10	Clue already sent snackbar . . . . .	29
5.1	CBOW Model . . . . .	33
5.2	Generate training data for CBOW . . . . .	34
5.3	Skip-Gram model . . . . .	35
5.4	Skip-Gram example . . . . .	35
5.5	UML diagram for <b>Spymaster</b> class . . . . .	40
5.6	End game with AI spymaster . . . . .	43
6.1	UML diagram for <b>Experiment</b> class . . . . .	46
6.2	UML diagram for <b>Evaluation</b> class . . . . .	46

6.3	Phase 1 part 1 view . . . . .	47
6.4	Phase 1 part 2 view . . . . .	48
6.5	Phase 2 view . . . . .	49
6.6	Clues chosen for Table 6.4 . . . . .	52
6.7	Clues chosen across all experiments . . . . .	53
A.1	Phase 1 part 1 board for user 2 . . . . .	57
A.2	Phase 1 part 1 board for user 3 . . . . .	58
A.3	Phase 1 part 1 board for user 4 . . . . .	59
A.4	Phase 1 part 1 board for user 5 . . . . .	59
A.5	Board 1 for Phase 2 . . . . .	60
A.6	Board 2 for Phase 2 . . . . .	60
A.7	Board 3 for Phase 2 . . . . .	61
A.8	Board 4 for Phase 2 . . . . .	61
A.9	Board 5 for Phase 2 . . . . .	62
A.10	Clues chosen for Table A.9 . . . . .	63
A.11	Clues chosen for Table A.10 . . . . .	64
A.12	Clues chosen for Table A.11 . . . . .	65
A.13	Clues chosen for Table A.12 . . . . .	66



# List of Tables

5.1	Co-occurrence Matrix . . . . .	36
6.1	Phase 1 part 1 results for user 1 . . . . .	50
6.2	Phase 1 part 2 results for user 1 . . . . .	50
6.3	Average results for precision@2 and recall@4 . . . . .	51
6.4	Phase 2 results for user 1 . . . . .	52
A.1	Phase 1 part 1 results for user 2 . . . . .	57
A.2	Phase 1 part 2 results for user 2 . . . . .	58
A.3	Phase 1 part 1 results for user 3 . . . . .	58
A.4	Phase 1 part 2 results for user 3 . . . . .	58
A.5	Phase 1 part 1 results for user 4 . . . . .	58
A.6	Phase 1 part 2 results for user 4 . . . . .	59
A.7	Phase 1 part 1 results for user 5 . . . . .	59
A.8	Phase 1 part 2 results for user 5 . . . . .	59
A.9	Phase 2 results for user 2 . . . . .	62
A.10	Phase 2 results for user 3 . . . . .	63
A.11	Phase 2 results for user 4 . . . . .	64
A.12	Phase 2 results for user 5 . . . . .	65

# Chapter 1

## Introduction

In the today's day and age, *Artificial Intelligence (AI)* is a buzz word that gets used a lot in the media whenever Computer Science is mentioned. However, more often than not, this term is used incorrectly by journalists. Due to their lack of technical knowledge on the topic, they often either misconstrue what exactly AI is to the general public, or else they over exaggerate success stories. This leads people to think that the current state of AI is much further along than it actually is, or else leaves them with an inaccurate representation of what AI is all about. With that being said, we must ask ourselves what exactly is Artificial Intelligence?

At its simplest form, Artificial Intelligence is a field, which combines computer science and robust datasets, to enable problem-solving. It also encompasses sub-fields of machine learning and deep learning, which are frequently mentioned in conjunction with Artificial Intelligence. These disciplines are comprised of AI algorithms which seek to create expert systems which make predictions or classifications based on input data<sup>1</sup>.

At its core, AI is nothing more than a collections of methods, which do something impressive. Whether that is predicting how much you should sell your house for, or drawing pretty pictures, all depends on what the goal of the AI is. AI is not some omnipotent piece of software that can do everything. In general, an AI is developed to carry out a specific task, and to do that task well.

---

<sup>1</sup><https://www.ibm.com/cloud/learn/what-is-artificial-intelligence>

There has been great development in this area in the last decade. Self-driving vehicles, news filters, medical diagnosis tools, bomb disposal robots, virtual assistants and image classifiers are all made possible due to the research that has been carried out in this field. Two very interesting areas in AI are language processing and gameplay.

*Natural Language Processing (NLP)* refers to the branch of computer science—and more specifically, the branch of artificial intelligence or AI—concerned with giving computers the ability to understand text and spoken words in much the same way human beings can<sup>2</sup>. NLP is so prevalent in our lives today that we often take it for granted. Things we use every day like spell check, autocomplete, spam filters, and virtual assistants such as Siri/Alexa/Google Assistant are all forms of NLP. In all of these examples, computers are able to identify appropriate words, phrases, or responses by using context clues, much like a human would.

*Reinforcement Learning (RL)* is the training of machine learning models to make a sequence of decisions.<sup>3</sup> There advancements with regards to having AI play games through the use of RL. Some famous computer programs that have been developed to for playing games include AlphaGo<sup>4</sup>, AlphaGo Zero<sup>5</sup> and AlphaZero<sup>6</sup>. In March 2016, AlphaGo beat 18 times world champion Lee Sedol 4-1. In October 2017, AlphaGo Zero (trained on self-play only) beat AlphaGo 100-0. In December 2017, AlphaZero beats StockFish at chess and Elmo at Shogi. Another example is how a group of researchers at MIT have been working on equipping RL with knowledge of how humans learn and act. They created an agent that can play video games called EMPA (the Exploring, Modeling, and Planning Agent) and tested it on about 90 Atari-like games[1]. Games serve as a create way to recreate key problems for AI to overcome. Whether its dealing with incomplete information like in Poker, or creating environment where AI must make real-time, spur of the moment decisions. Games also provide a low-risk environment in which to test and train AI.

This project aims to combine NLP and gameplay by having a AI play the

---

<sup>2</sup><https://www.ibm.com/cloud/learn/natural-language-processing>

<sup>3</sup><https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/>

<sup>4</sup><https://en.wikipedia.org/wiki/AlphaGo>

<sup>5</sup>[https://en.wikipedia.org/wiki/AlphaGo\\_Zero](https://en.wikipedia.org/wiki/AlphaGo_Zero)

<sup>6</sup><https://en.wikipedia.org/wiki/AlphaZero>

board game Codenames. In Chapter 2 we will some background as to how Codenames is played and what the aim of the game is. In Chapter 3 we will show how we designed our application, and in Chapter 4 we will go through how we created a web-app which allows users to play the game over the internet. In Chapter 5, we shall discuss the kinds of AI that we used during the project: *word2vec* and *GloVe*. We will discuss the theory behind how these word embeddings are created, and we shall introduce the algorithm that was used to allow the AI to play the game. Then, in Chapter 6, we shall discuss how we evaluated the performance of our AI through human trials. Finally, in Chapter 7, we will analyse our findings and draw conclusions on how the AI performed, and end the report by proposing some ideas as to how this work could be taken further in the future and any interesting ideas that we did not get a chance to carry out during the course of the project.

# Chapter 2

## Background

Codenames is a card game for 4-8 players that was released in 2015. It was designed by Vlaada Chvátil and published by Czech Games Edition. It is a guessing game in which codenames (words) on the board are related to a clue-word given by another player.

Before we explain how the game works, there is some terminology associated with the game we must explain: an *agent* is word on the board, *spymasters* are players who can see the colours associated with each agent, and who must send one word clues to their team, *operatives* are players who cannot see the colours associated with agents, and who must try and guess which agents are on their team from the spymaster's clue, *civilians* are agents who does not belong to the red or the blue team, and the *assassin* is an agent who does not belong to the red or the blue team, and must be avoided.

The typical board consists of 25 words in a 5×5 grid. A number of these represent the red team's words, a number represents the blue team's words, one word represents the assassin, and the remaining words represent civilians. The team that starts has nine agents, with the other team having eight.

Initially, only the spymasters can see all the positions of the agents, civilians and assassin on the board, so the other team members (the operatives) can only see the 25 words, but they are unaware of what words relate to their team. In the game, two teams compete against each other, and each teams' spymaster gives one word clues that can point to multiple words on the board. The spymaster is free to choose the clue that they send, once it does

not break a certain set of rules such that it cannot be a word already present on the board, nor can it contain or be contained in a word already on the board. This being said the game's rules state to not be too strict. A hint is sent in the form of a word and a number. The number relates to the number of words that the clue relates to. The number of guesses an operative has is this number plus one.

What makes the game interesting is that a clue may match both the red and blue agents, or possibly a civilian or the assassin. Thus it is the goal of the spymaster to choose an appropriate clue that matches as many of his team's colour, without relating to the other words. When a clue is sent, the operatives make guesses as to which words they believe the spymaster is trying to match with his clue. When a word is guessed, it is covered with the appropriate card, which is either a red or blue agent card, a civilian card, or the assassin card. A round ends when either a word belonging to the opposing team or a civilian is chosen, or when the operatives who's move it is has finished guessing. If the assassin is guessed, the game is over and the team who guessed that word loses. If an operative still has guesses remaining, but is unable to deduce word the clue relates to, they may also end their turn by declaring that they are finished guessing.



Figure 2.1: Codenames board

Figure 2.1 shows a typical board. In this image, any covered words are agents that have already been guessed by the operatives, while the uncovered words are agents that the spymasters must try and connect. The spymasters have a special card that tells them the position of the agents.

# Chapter 3

## Design

### 3.1 Architecture

It was essential that this game would be playable over the internet. In order to achieve this a client-server model was chosen. The backend server's responsibilities were knowing which clients were connected, it maintained game state, and it handled any requests sent by the client. The client was responsible for the presentation layer, which showed the current game state, and also taking input from the user, and sending messages to the backend.

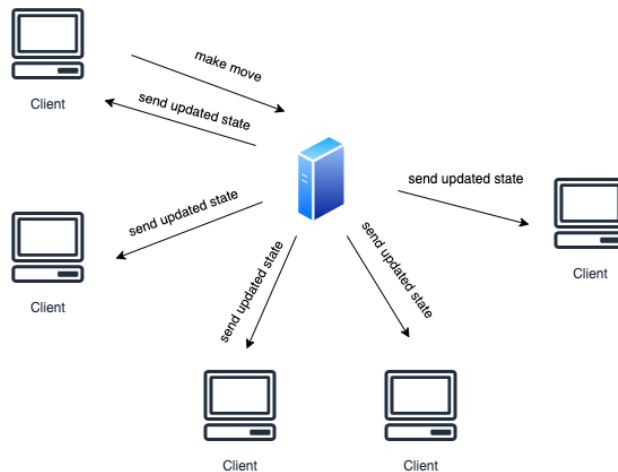


Figure 3.1: Multiplayer game with a client-server architecture

An important requirement that this game needed to have was that it needed to be able to update in real time and maintain the consistent state among all connected users. It would be impractical for one user to make a move and get an updated version of the game, yet having other clients that are looking at an old version of the game. An example of how this should look can be seen in the Figure 3.1

This model is slightly different from the typical request-response model, as here only one client has sent a request (the player making a move), but all other clients need to be sent a response. In order to achieve the required behavior, there were a few options: HTTP Polling, HTTP Long Polling or Websockets.

### **HTTP Polling**

In HTTP Polling, the client keeps making requests to the server at a regular interval, typically 1-2 seconds. The server gets this request, it does its work and sends back a response. However, in this scenario, a response could be empty. If the server has nothing to send back, it will send back an empty response. This results in a lot of unnecessary network calls. The 1-2 second interval also adds a delay to the response that the client receives.

### **HTTP Long Polling**

HTTP Long Polling differs from HTTP Polling because instead of sending a request to the server at a regular interval, the client maintains an open connection for a long time and it waits for the server to respond. Once the server has a response to send back it sends it and the connection to the server is closed. Once the client receives the response, it opens up another connection with the server and the process is repeated. This improves upon HTTP Poling as there will be no empty response. In HTTP Long Polling, a timeout is also added to the request. If the server has not sent back a response after the timeout has occurred, the connection is closed and the client opens another connection to the server.

### **Websockets**

A WebSocket is a persistent connection between a client and server. WebSockets provide a bidirectional, full-duplex communication channel that operates over HTTP through a single TCP/IP socket connection. At its core, the WebSocket protocol facilitates message passing between a client and server. How it works is that a client does a WebSocket handshake with the server and then TCP connection gets established between the client and



the server through the WebSocket. Because this is a bidirectional connection, this means that the server can send a message to the client any time, it does not need to receive a request from the client first. Likewise, the client can send a message to the server at any time. This also reduces the overhead of handshaking over and over again like in HTTP Polling or HTTP Long Polling, as the handshake is just done at the very start.

While both HTTP Polling and HTTP Long Polling are viable options for this project, WebSockets are the most suitable. Each time a player makes a move, it can be sent through a WebSocket connection, and the server can update the game state and send the new state to all the players through their WebSocket connections. This will ensure that all clients have the same state and there will be no inconsistency between them.

## 3.2 Models

As this app is effectively an online representation of a board game, there are three main components that make up the app: players, teams and the game state. The relationship between all of these can be seen through the following UML diagram.

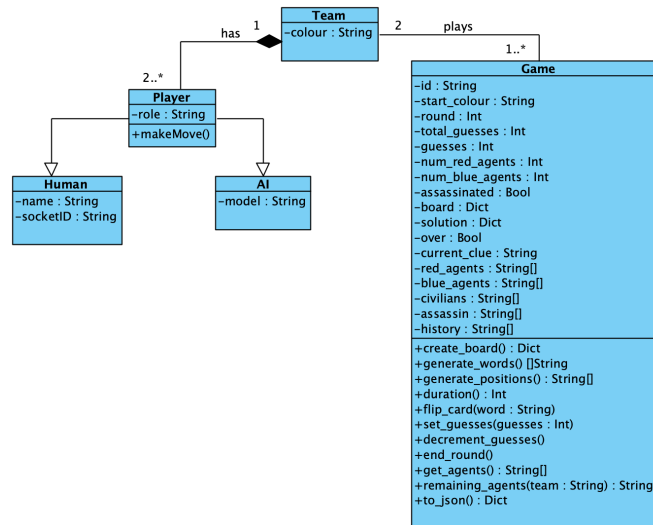


Figure 3.2: UML diagram showing relationship between players, teams and games

As shown in the Figure 3.2, a player is either a human or an AI. Each team consists of two or more players, and two teams are needed to play a game.

### 3.3 User Interface

When it came to designing the user interface, the goal was to make it both minimal and compact, but while also displaying all the relevant information for someone to be able to play the game and know what is going on. The game stats and the game board would be in the centre, with an area on the left for the clues and a chat area on the right.

The game stats would contain information like the current round, the number of remaining agents, the number of connected players and the number of guesses remaining. The game board would show the current state of the board, which agents had been guessed, and which still remained. The clue area on the left is where clues sent from the spymasters would appear. The chat area is where teams can discuss what they believe a clue might mean. In other online representations of Codenames, a chat area is not included, so it assumes those playing the game are either together, or connected in a voice chat on a different platform. While this surely works, having an in-app chat system would make playing more enjoyable. A rough wireframe of this design can be seen in the Figure 3.3.

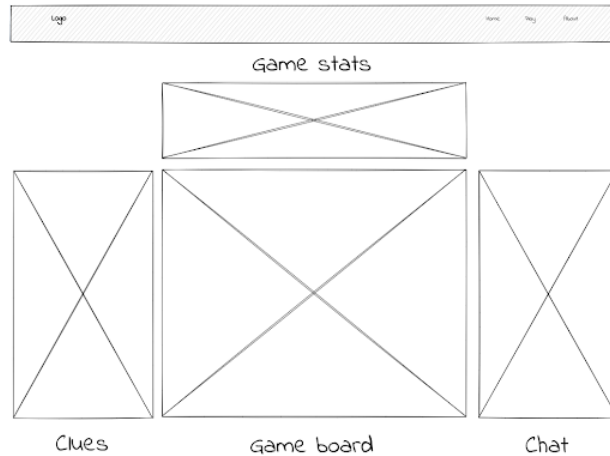


Figure 3.3: Wireframe for game area

### 3.4 User Interaction

When a user initially visits the app, they can either create a game, or join an existing game. Once in a game, they must choose their team and their role. If they are playing as a spymaster, they must send a valid clue and wait until it is their turn to send a clue again. If they are an operative, they must wait for their team's spymaster to send a clue, and then they must make their guesses until all the agents have been found. Figure 3.4 shows a flowchart describing this interaction.

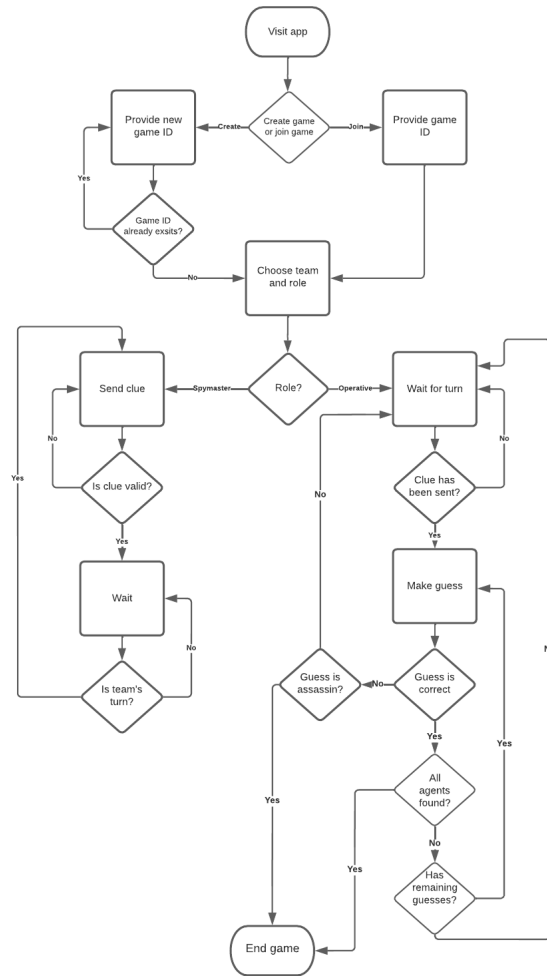


Figure 3.4: Flowchart for user interaction

When it came to designing how the app would look and operate, research was conducted into how other online implementations of Codenames were designed. Some examples included the official web implementation<sup>1</sup>, an implementation by GitHub user *jbowens*<sup>2</sup> and another implementation by GitHub user *koldoon*<sup>3</sup>. Each implementation had interesting and questionable design decisions, but the basis for the design of the overall flow of the game was a combination of the best features from each app. For example, in *jbowens*'s implementation, users are free to choose their own game IDs, however on the game board the difference between seeing what an operative sees versus what a spymaster sees is just a toggle, meaning anyone could cheat and see through the eyes of a spymaster. The implementation by *koldoon* makes users chose their role, and then depending on their selection they are be presented a different board, which isolated the spymaster view from the operative views, which is clearly a preferable design and was the the design that was chosen for this project.

## 3.5 Technologies

As Codenames is a board game which is meant to be enjoyed by multiple players, it was essential that this project supported real-time updating multi-user playability. In order to achieve this the app was separated into two main parts: the frontend which dealt with all the user interactions and displayed all information needed, and a backend which handles all the Websocket connections which allow for these real time updates.

When it came to the frontend there were multiple frameworks as well as Vanilla Javascript that could have been chosen. The most popular frameworks on the market are React, Angular and Vue. However, there is a new technology that is starting to gain a lot of traction due to its speed, and its ease of use: Svelte. Svelte is a radical new approach to building user interfaces. Whereas traditional frameworks like React and Vue do the bulk of their work in the browser, Svelte shifts that work into a compile step that happens when you build your app<sup>4</sup>. Having heard Svelte be praised for how

---

<sup>1</sup><https://codenames.game/>

<sup>2</sup><https://www.horsepaste.com/>

<sup>3</sup><https://codenames.koldoon.ru/>

<sup>4</sup><https://svelte.dev/>

quickly you can build modern, reactive websites it was chosen as the framework for this project. In particular, SvelteKit, a framework for building web applications of all sizes, with a beautiful development experience and flexible filesystem-based routing<sup>5</sup> was used.

Very little third party libraries were used during the project, as SvelteKit was more than capable of producing the required functionality by itself, however in order to interface with the Websockets, the Socket.IO was used. Socket.IO is a library that enables real-time, bidirectional and event-based communication between the browser and the server<sup>6</sup>. This library is what made it possible to send events to, and react to events received from the backend server.

In order to give the web app a more modern look and feel, the Material UI component library Svelte Material UI (SMUI)<sup>7</sup> was used. This also helped decrease the time for development, as the provided UI components were extremely intuitive to work with and added great functionality to the app. Initially, the decision to use Tailwind CSS<sup>8</sup> had been taken, however the time required to get comfortable with using this utility library did not seem to be worth the investment. The ease of use, albeit at the cost of customization, of SMUI was the reason it was chosen over Tailwind CSS.

Handling socket events sent from the frontend and emitting events in response, maintaining and updating the game state and running the AI models are the main responsibilities of the backend. The backend of this project is written in Python. The framework used on the backend is Flask<sup>9</sup>. When it comes to backend frameworks in Python, the two most popular would be either Flask or Django<sup>10</sup>. Having prior experience with Flask was the reason this was chosen over Django.

In order to work with sockets, the flask-socketio library<sup>11</sup> was used. This library is the Python implementation of the Socket.IO library used on the frontend, so interfacing between the two was straightforward. This library

---

<sup>5</sup><https://kit.svelte.dev/>

<sup>6</sup><https://socket.io/>

<sup>7</sup><https://sveltematerialui.com/>

<sup>8</sup><https://tailwindcss.com/>

<sup>9</sup><https://flask.palletsprojects.com/en/2.0.x/#>

<sup>10</sup><https://www.djangoproject.com/>

<sup>11</sup><https://flask-socketio.readthedocs.io/en/latest/>

made it very easy to set up Websocket events on a server, which would listen to messages sent from the client. When one of these events were received, the backend would change the state of the game object, and then send back the updated state to the client.

Python is one of the most popular programming languages when it comes to AI. This was also a strong reason for choosing this as the backend language, rather than something like NodeJS or Golang. As the AI in this project heavily revolved around NLP and Vector Space Modelling, the gensim library<sup>12</sup> was chosen as the main library for working with the word vectors. This library provided extremely helpful methods for working with trained word vectors.

Both the frontend and backend were set up to be run as Docker containers. A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another<sup>13</sup>. As both the frontend and backend ran as separate containers, they had to be networked together so that they could communicate. This was done using Docker Compose, a tool for defining and running multi-container Docker applications<sup>14</sup>. The system has a SvelteKit image and a Flask image, which each contain all the code and dependencies needed to run each app. The environment required for each of these is defined in a Dockerfile, which is then used to create the Docker image. These images are what is used to create the containers. Docker Compose allows for these different images to be built and the containers to be run and networked together all from one place.

When it came to deployment there were many choices such as Digital Ocean<sup>15</sup>, Amazon AWS<sup>16</sup> or Microsoft Azure<sup>17</sup>. All these however require some form of payment. In the end, we opted for a free cloud hosting service kindly provided for students in UCC by UCC Netsoc: Netsoc Cloud<sup>18</sup>. The greatest benefit of running the app through docker containers was that if it worked on a local machine, it was guaranteed to work on this remote server. All that

---

<sup>12</sup><https://radimrehurek.com/gensim/>

<sup>13</sup><https://www.docker.com/resources/what-containerreference>

<sup>14</sup><https://docs.docker.com/compose/>

<sup>15</sup><https://www.digitalocean.com/>

<sup>16</sup><https://aws.amazon.com/>

<sup>17</sup><https://azure.microsoft.com/en-gb/>

<sup>18</sup><https://netsoc.cloud/>

was involved in getting it deployed was to download the Docker images onto the server and run the containers from these images.

As this project was worked on over the course of two semesters, a good version control system was required. Git is an industry standard so this was the tool used, and we chose GitHub as the repository hosting service. Choosing GitHub provided access to GitHub Actions a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test, and deployment pipeline<sup>19</sup>. Whenever a change to the codebase was pushed to GitHub, a GitHub Action would run which would build the new Docker images and push them to DockerHub. The GitHub Action would then ssh into the remote server where the project was hosted, download the new Docker images, and build and run the updated containers. Had a different repository hosting service, such as GitLab<sup>20</sup>, been chosen, these automated deployments would have required additional services such as Travis<sup>21</sup> or Jenkins<sup>22</sup>, but with GitHub Actions ease of use and integration with GitHub, it is clear that the decision to chose this was the correct one.

---

<sup>19</sup><https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions>

<sup>20</sup><https://about.gitlab.com/>

<sup>21</sup><https://www.travis-ci.com/>

<sup>22</sup><https://www.jenkins.io/>

## Chapter 4

# Implementation

The web application is split into multiple pages using SvelteKits file based routing. When a user initially visits the app, they are presented with a minimal landing page which conveys what the web app is about, as seen in Figure 4.1

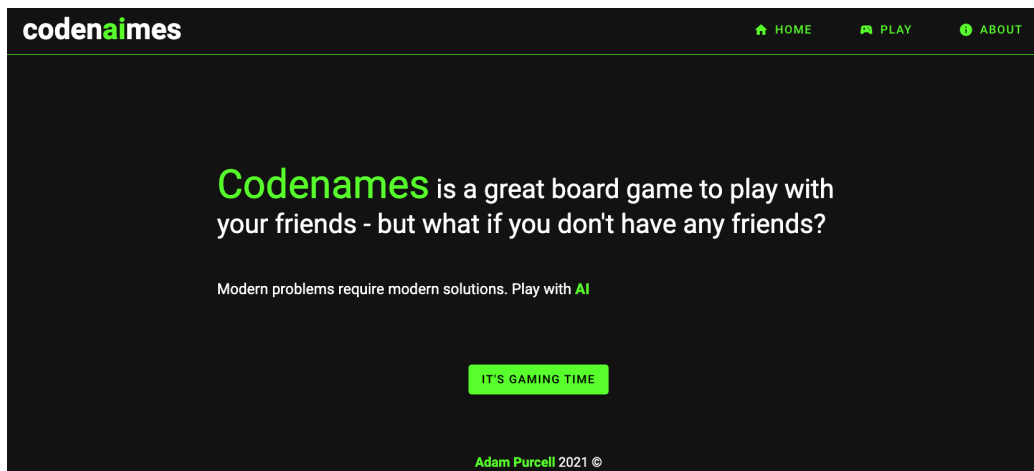


Figure 4.1: Landing page

Upon clicking the **IT'S GAMING TIME** button, a user will be redirected to a new page where they are presented with options to create or join a game, as shown in Figure 4.2.



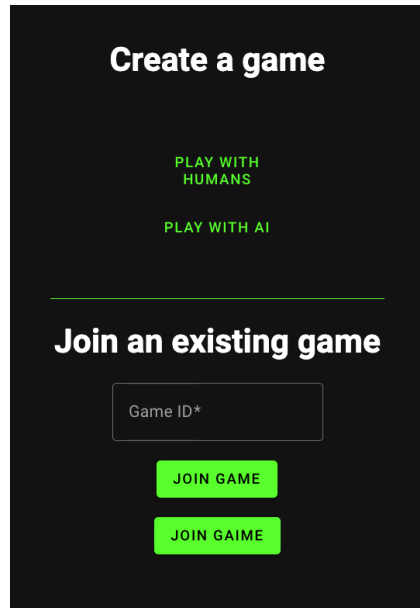


Figure 4.2: Create/Join game view

The general application flow for playing with humans or with AI is very similar, so only playing with humans will be discussed here, and any differences in the implementation for playing with AI will be discussed in Chapter 5.

To join a game, a user must provide the game ID of a game. If the user chooses to create a game they will be redirected to a page where they can choose a game ID for their game. Upon entering a game ID and clicking the create game button, a request will be sent to the backend which will create the game object.

Figure 4.3 shows a UML diagram of the `Game` class, which was shown previously in Figure 3.2. In a `Game` object, the solution to a game is represented as a dictionary where the keys are the agents (words) and the values are the corresponding representations of the agents, i.e. red or blue agent, civilian or assassin. The `Game` object stores this dictionary in its `solution` attribute. The initial state of the game is presented as a dictionary where all the values are initialized as `None`. When a user makes a guess, the `flip_card()` method is run which checks the color of the card from the solution, and updates the value to match this. A new `Game` object is initialized and the `create_board()` method is run, which reads in a text file containing all possible words into

memory, shuffles it, and returns a list of 25 words. A list is created with the correct number of red and blue agents, civilians and the assassin, and is shuffled to ensure a random board. These two lists are zipped together to associate a word with a color, and then these pairs are converted into the solution and board dictionaries which were previously described. The remaining attributes on the game object are additional information such as the number of guesses that have been made, the current clue, the round. The remaining methods are helper methods, for decrementing the number of guesses remaining, ending a round, or converting the game board dictionary to JSON.

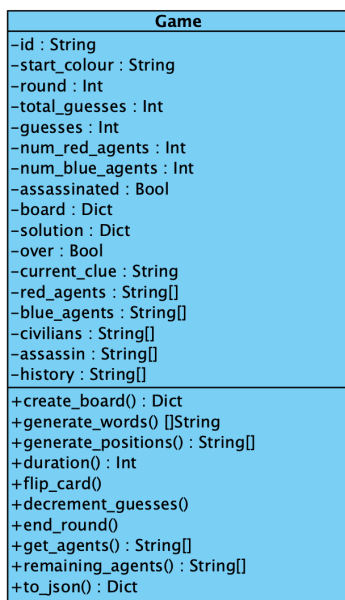


Figure 4.3: UML diagram of `Game` class

Figure 4.4 shows what the form looks like. This form is reactive, so as the user enters their name, or changes their team or role, the button and icon at the bottom will update to reflect the user's decision. The icon at the bottom also shows how a player will look to other players. Reactivity is at the core of Svelte, and the framework provides *reactive declarations* which allow a components state to be computed from other components and recomputed whenever they change. Svelte will automatically update the DOM whenever a component's state changes. Once a user has entered their details and has

**1. Identify yourself**

Name\*  
Adam

**2. Choose your team**

☒ Red ☐ Blue

**3. Choose your role**

☒ Spymaster ☐ Operative

**4. Play**

[JOIN AS SPYMASTER](#) Adam

Figure 4.4: Join game form

confirmed their team and username will be stored in local storage and the user will be redirected to a new page depending on their role.

Figure 4.5 shows what the board initially looks like for a spymaster, and Figure 4.6 shows what it looks like for an operative.

**Clues** 9 - 8 **ROUND 1** [COPY GAME LINK](#)

1 adam joined

FIGURE	PANTS	MISSILE	TRAIN	CELL
EUROPE	PAPER	STREAM	COPPER	LAWYER
OIL	TURKEY	SUB	STATE	BLOCK
FIRE	LONDON	TOOTH	ROW	PUMPKIN
MASS	SWITCH	CALF	ROUND	GROUND

Clue Message

Figure 4.5: Initial view for spymaster

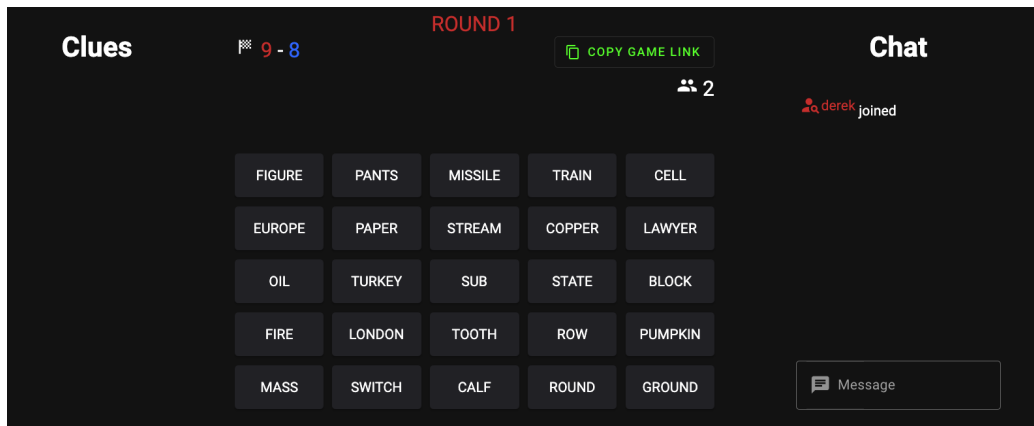


Figure 4.6: Initial view for operative

Figure 4.7 and Figure 4.8 show what it looks like mid game for a spymaster and operative respectively.

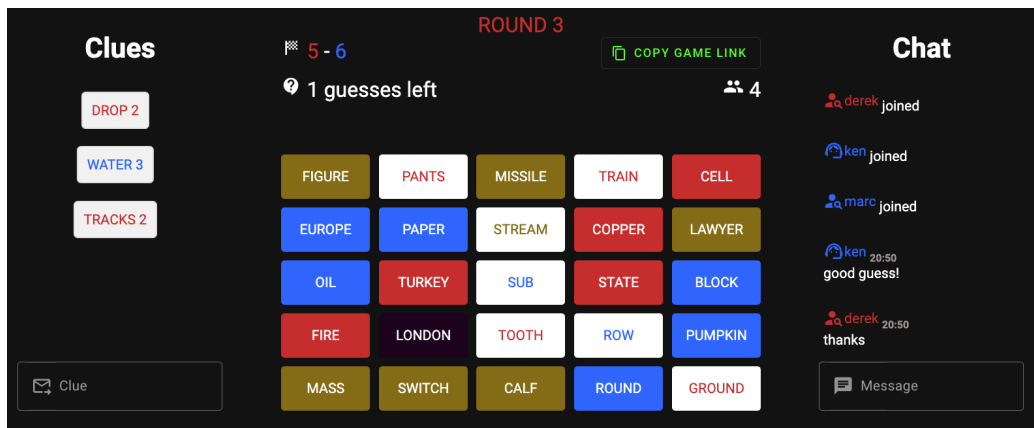


Figure 4.7: Mid-game view for spymaster

Figures 4.5, 4.6, 4.7, 4.8 show the main game area that a user sees. It is broken up into many components as seen in Figure 3.3. We will now discuss each component.

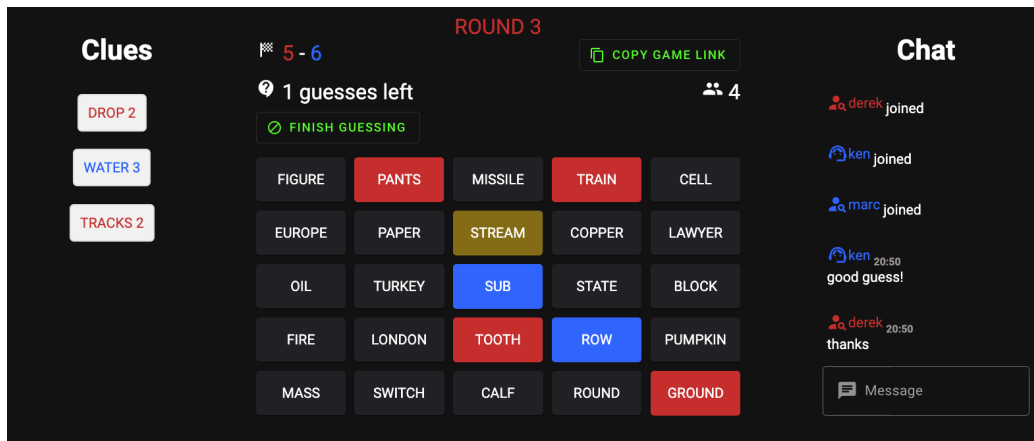


Figure 4.8: Mid-game view for operative

## 4.1 Clue Area

Sending clues is an integral part of playing Codenames. Only a spymaster can send a clue. The clue area is implemented in such a way that an option to input a clue will only be shown to spymasters, and furthermore, only to spymasters whose turn it is. This ensures a fair game as you must wait for your turn to send a clue. A clue is validated before it can be sent, so it must meet those requirements. If a clue is invalid, a dialog pops up informing the user that they must send a clue in the format "word number", as shown in Figure 4.9.

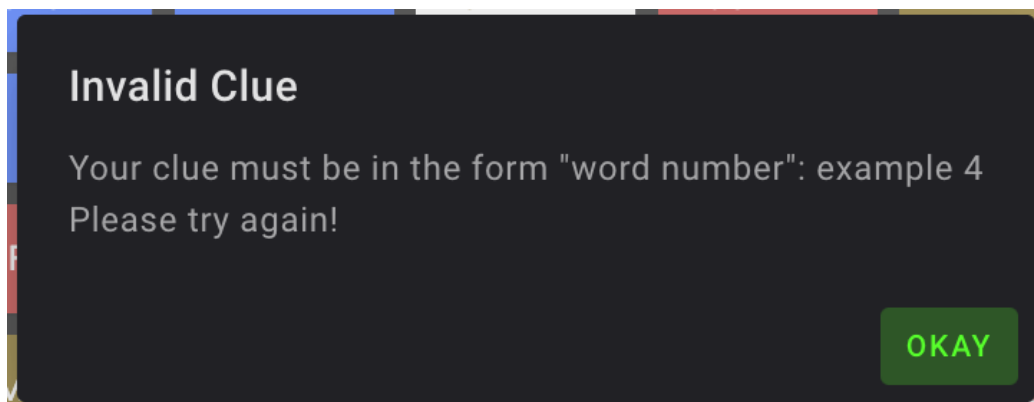


Figure 4.9: Invalid clue pop-up

Once a clue has been validated, the front end will send the clue to the backend through a Websocket using the **send-clue** message. When backend receives this clue it sends this message it uses the game ID that was sent in the message to get the game object associated with this room. It will then set the number of remaining guesses (the number in the clue plus one) and set the current clue in the game Websocket message.

After updating the game object with this information, the backend emits two messages to the frontend. Firstly it sends the state of the game back to the frontend as a **Game** object through a **send-state** message, and secondly it sends the clue back through a **send-clue** message. The **send-clue** message is what sends the clue to all the players in the game. The frontend is set up to listen for this message from the Websockets. When it receives this message, it updates an array containing any clues. Due to Sveltes reactivity, once it notices that the array has changed it will automatically update the DOM to show the new message.

As stated in the rules, a spymaster may only send one message per round. In order to prevent spymasters from cheating, when a clue is sent, it is stored in a variable on the frontend. If a spymaster tries to send a second clue, the previous clue will be compared to the current clue that is stored in the games state, and if they match a snackbar message will pop up letting the spymaster know they cannot send another message this round as shown in Figure 4.10

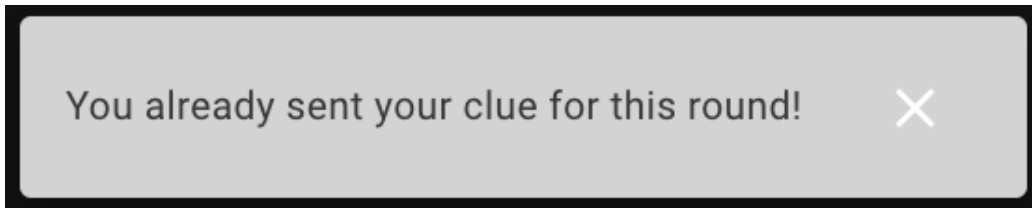


Figure 4.10: Clue already sent snackbar

## 4.2 Game Board

The game board is a 5x5 is the most essential component for playing the game, as these are the words that operatives must try to link to the spymasters clues. When a user wants to guess, all they have to do is click the word

which they think the clue may be linked to. If a player tries to make a guess while it is not their turn, a snackbar message will pop up letting them know it is not their turn yet. When a user makes a guess, a check is done on the client side which checks what colour the agent they voted for corresponds to. If their guess was not correct, a snackbar message will pop up letting know that they were incorrect and explain why, i.e. if it was the other team's agent or a civilian. If it is the assassin the game will end. Once the guess has been made, and any snackbars that might have been required, the frontend sends a **make-move** socket message to the backend.

When the backend receives this message, it gets a reference to the **Game** object corresponding to that game, and calls the **flip\_card()** method on the **Game** object. This updates the state of the game, which the backend then sends to all connected clients through websocket connections with the **send-state** message. When the clients receive this message, they will update their locally stored game. Svelte will then automatically update whatever agent had been voted for, so that all clients know what agent had been guessed, and what the result of that guess was.

## 4.3 Chat Area

The chat area is a simple yet extremely useful component. It consists of an input field where users can type in their messages, and an area to display messages. As stated earlier, this was not necessary for a playable game, however we believed that it would significantly add to the user experience while using the app.

When a user sends their message to the backend it gets sent through the a Websocket through a **send-state** message. When the backend receives this it adds a timestamp to the data and sends it to all the other users through their Websocket connections. When the client receives the new message, it updates its locally stored array of messages. Each message has a corresponding user and team, so that users can know who sent each message. The chat area also makes announcements to everyone when a player joins or leaves the game.

The chat area makes use of two very useful functions provided by Svelte,

`beforeUpdate()` and `afterUpdate()`<sup>1</sup> to ensure that when a new message is sent to update the scroll position of the new message. This ensures that users do not have to scroll in the chat window in order to see the new messages. This kind of behaviour is typical with any chat-based application, so it made sense to also include this sort of functionality in the app. This kind of functionality is difficult to implement in a purely state-driven way, but thankfully Svelte provides this kind of functionality out of the box. This same mechanism also applies to the clue area, so that new clues or messages will always be at the bottom and older ones will get pushed out of site, but should a user want to go back to see them they can scroll back.

## 4.4 Game Stats

Any good game has some sort of heads-up display (HUD) that shows information about the current state of the game. This app is no different. The stats area provides useful info such as the round, how many agents are left for each team, the number of players in the game, the number of guesses left, an option to finish guessing and and an option to copy the game link.

As explained earlier, when a user makes a move the updated game state is sent to all clients, and thanks to Svelte's reactive declarations, these stats change automatically after each move. The colour of the round indicates which team's go it is.

The inclusion of a copy link button means that users have two ways to join a game. Firstly, they can visit the app and enter the game ID and then they will be able to join. While this is a perfectly valid way of joining a game, we thought it would be a lot more user-friendly if someone in the game could send a link to their friend which would allow them to join the game. That is exactly what the copy game link button does. When a user clicks this, a dummy text area is created in the app, whose value is set as a URL to join the game. This URL is then copied to the clipboard of the user, so they can just paste it to their friend. In order to let the user know the link was successfully copied, the wording on the button changes from *copy game link* to *copied game link* for two seconds. This change lets the user know that their clipboard has been updated.

---

<sup>1</sup><https://svelte.dev/tutorial/update>



# Chapter 5

## AI Spymasters

### 5.1 Analysis

#### 5.1.1 Word Embeddings

A word embedding is a numeric vector input that represents a word. Each of these vectors often has tens or hundreds of dimensions, as opposed to using sparse word representations such as one-hot encoding which could have thousands or millions of dimensions. Word embedding methods take a pre-defined fixed sized vocabulary from a corpus of text and learn real-valued vector representations. Word embeddings can be used to find words with similar meanings. In word embeddings, a word is said to be similar to another word if they are often used in similar contexts. If you take for example the word "cat" and "dog", in a large corpus of text these two words will be surrounded by words like "feed" and "play" which gives some indication that they are similar words. Two of the most popular techniques for learning word embeddings are word2vec and GloVe.

#### 5.1.2 word2vec

With word2vec there are two methods for constructing the word embeddings: Skip-Gram and Common Bag of Words (CBOW)[2].

### 5.1.2.1 CBOW

In the CBOW model, the distributed representations of context (or surrounding words) are combined to predict the word in the middle. First, each word is one-hot encoded. Not all words in the sentence are used, only words that appear in a window. If the window size is 3, then the middle word is the word to be predicted, and the surrounding 2 context words are fed into the neural network as input. These are passed to a hidden layer and then the target word is predicted. It is passed through the softmax function to get the probabilities and it is compared with the actual word. The error is then used to update the weights. The window is then slid and the process is repeated. After this process the input weight matrix used to get the embeddings by multiplying by the one-hot vector of a particular word.

An example of what the architecture might look like for predicting a target word from multiple context words is as shown in Figure 5.1

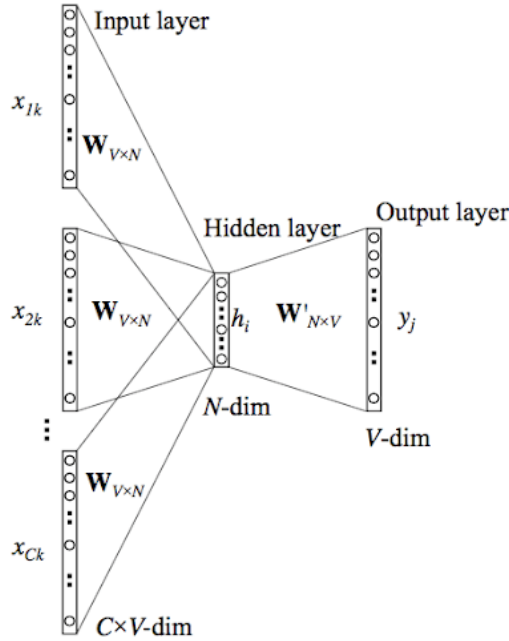


Figure 5.1: CBOW Model

Here input or context word is a one hot encoded vector of size  $V$ , the hidden layers contain  $N$  neurons and the output is again a  $V$  length vector with the

elements being soft maxed. The hidden layers just copy the weighted sum of inputs to the next layer. There is no activation like sigmoid, tanh or ReLU. The only non-linearity is the softmax calculations in the output layer<sup>1</sup>.

An example of what the training window will look like can be seen in Figure 5.2<sup>2</sup>. As you can see in the image, we have a window size of three, which is slide across the text, and each time the target word and associated context words change.

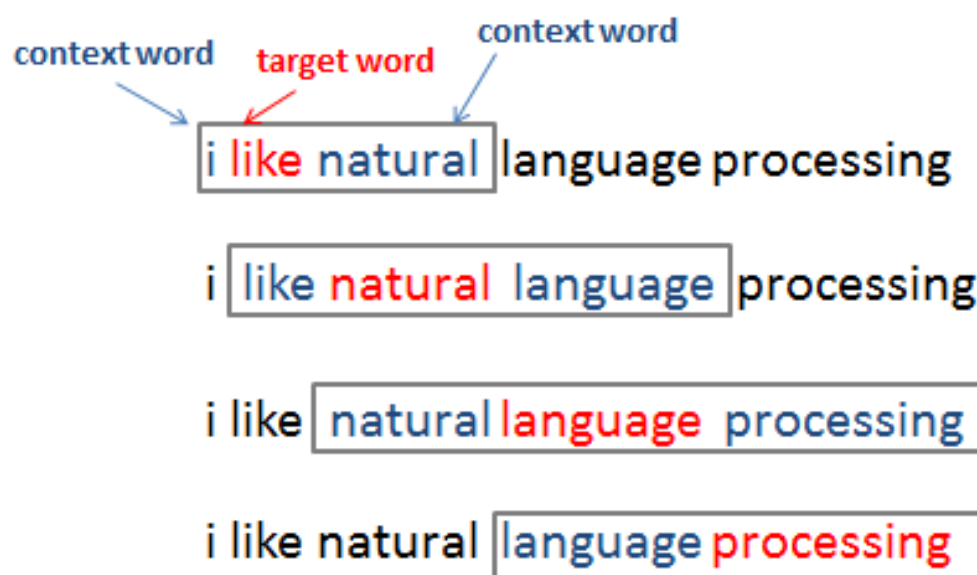


Figure 5.2: Generate training data for CBOW

#### 5.1.2.2 Skip-Gram

The skip-gram model architecture usually tries to achieve the reverse of what the CBOW model does. The window size is chosen and given the centre word as input the context words are predicted and the weights are updated in a similar fashion, as shown in Figure 5.3.

<sup>1</sup><https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa>

<sup>2</sup><https://thinkinfi.com/continuous-bag-of-words-cbow-multi-word-model-how-it-works/>

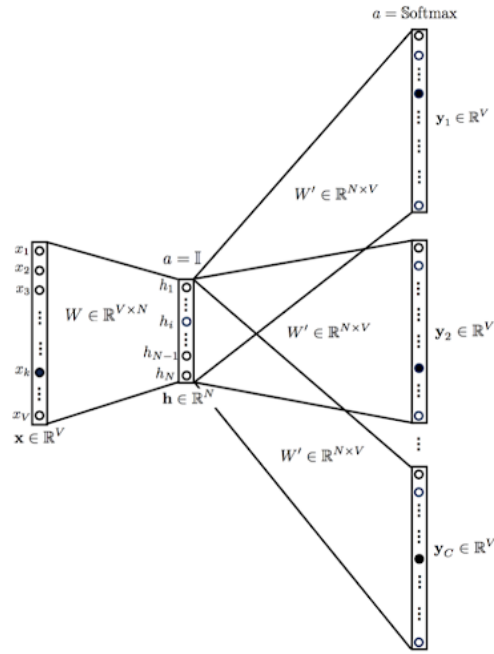


Figure 5.3: Skip-Gram model

An example of what this would look like can be seen in Figure 5.4. The target word here is **jump** and it is trying to predict the context words.

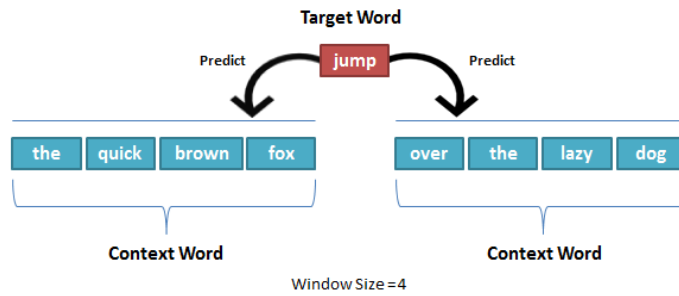


Figure 5.4: Skip-Gram example

### 5.1.3 GloVe

GloVe stands for Global Vectors for word representation. It is an unsupervised learning algorithm developed by researchers at Stanford University[3] aiming to generate word embeddings by aggregating global word co-occurrence matrices from a given corpus.

The difference between GloVe and word2vec is that in word2vec only the local property of the dataset is considered, whereas in GloVe the global property of the dataset. A co-occurrence matrix is constructed based on how frequently a word is seen in some context to another word in a large corpus of text. In the co-occurrence matrix, each row is the “word” and the columns are the “context”. For each term, context words are looked for within some area before and after the word determined by a window size.

In the co-occurrence matrix both the rows and columns correspond to words and the entries correspond to the number of times a given word occurs in the context of another given word. Take the following corpus with two documents as example:

**I love to code. I love pints of Stout.**

Table 5.1 shows the co-occurrence matrix that would be constructed from this corpus.

	<b>I</b>	<b>love</b>	<b>to</b>	<b>code</b>	<b>pints</b>	<b>of</b>	<b>Stout</b>	<b>.</b>
<b>I</b>	0	2	0	0	0	0	0	0
<b>love</b>	2	0	1	0	1	0	0	0
<b>to</b>	0	1	0	1	0	0	0	0
<b>code</b>	0	0	1	0	0	0	0	1
<b>pints</b>	0	1	0	0	0	1	0	0
<b>of</b>	0	0	0	0	1	0	1	0
<b>Stout</b>	0	0	0	0	0	1	0	1
<b>.</b>	0	0	0	1	0	0	1	0

Table 5.1: Co-occurrence Matrix

In GloVe, the relationship of different words can be examined by studying the ratio of their co-occurrence probabilities with various probe words.

Now let's understand at a high level where the equations derived in Pennington et al. (2014) are derived.

First let's define  $x_{ij} = \# j$  appears in the context of  $i$ , so for example in the above matrix  $x_{love} = 2$

Next let's define

$$X_i = \sum_k X_{ik}$$

to be the number of times any word appears in the context of word  $i$ . Here,  $k$  goes from 0 to the number of words in the vocabulary. This is essentially the summation of the column values for a particular row.

Now let's define

$$P(j|i) = \frac{X_{ij}}{X_i}$$

where to be the probability that  $j$  is in the context of  $i$

The starting place for a word vector can be defined as

$$F(w_i, w_j, \tilde{w}_k) = \frac{P(k|i)}{P(k|j)}$$

Here  $F$  is an arbitrary function and the inputs are 3 word vectors where  $\tilde{w}_k$  is the context word while  $w_i$  and  $w_k$  are the middle words. So this function will calculate the ratio of the two probabilities of the context word to the middle words. Currently though, on the left hand side we have vectors while on the right hand side we have scalars, so it is necessary to convert these word vectors into scalars. Also,  $F$  takes three arguments but it is difficult to write a loss function with 3 variables so we also need to try and minimize the number of arguments here. Not only this but there is also the issue of choosing the correct function from the thousands of functions we can apply to vectors. In order to overcome these difficulties Pennington et al. did the following.

To convert the vectors to scalars, the dot product was used

$$F((w_i - w_j)^T \cdot \tilde{w}k) = \frac{P(k|i)}{P(k|j)}$$

The reason for using a vector difference  $w_i - w_j$  is to compute the analogies and the transpose is taken just to be compatible with the dimensions.

To limit the number of applicable functions Pennington et al. assumed that  $F$  obeyed the homomorphism law, which by applying some group theory the left hand side of the equation can be written as

$$\frac{F(w_i^T \tilde{w}_k)}{F(w_j^T \tilde{w}_k)}$$

thus giving that

$$\frac{F(w_i^T \tilde{w}_k)}{F(w_j^T \tilde{w}_k)} = \frac{P(k|i)}{P(k|j)}$$

So,

$$F(w_i^T \cdot \tilde{w}k) = cP(k|i)$$

and we can safely ignore the constant  $c$  because it wont change the form of our relationship. And it turns out that

$$F(x) = e^x$$

is a solution. Thus,

$$w_i^T \cdot \tilde{w}_k = \log_e(P(k|i)) = \log_e(X_{ik}) - \log_e(X_i)$$

Then, introducing the biases

$$w_i^T \cdot \tilde{w}_k + b_i + \tilde{b}_k = \log_e(X_{ik})$$

where  $b_i$  is the bias for  $w_i^T$  and  $\tilde{b}_k$  is the bias for  $\tilde{w}_k$ . Also  $\log_e(X_i)$  has been absorbed into the biases.

Thus, the loss function can be defined as

$$J = \sum_{i,j=1}^V f(X_{ij})[w_i^T \cdot \tilde{w}_j + b_i + \tilde{b}_j - \log_e(X_{ij})]^2$$

In this equation,  $j$  is acting as the context word, and  $i$  is a middle word. If you ignore the  $f(X_{ij})$ , the equation looks similar to a least squares function used in linear regression, which is  $(y - \hat{y})^2$ , which is a convex function thus helping the model to learn better. The  $f(X_{ij})$  term is an arbitrary function with no theoretical background which the researchers included to improve the performance of the model. The  $V$  term is the length of the vocabulary.

## 5.2 Related Work

There have been two papers in which Natural Language Processing has been used to try and play Codenames. The first is Kim et al. (2019)[4]. In this paper, the researchers used WordNet, word2vec and GloVe vectors to develop AI bots to play both the roles of spymasters and operatives, and they performed their evaluation by running a round-robin tournament of 30 games for all pairs of spymaster and operatives (the same 30 boards were presented to all pairs). For each of the spymasters, they considered 3 different threshold levels, to see how the different approaches trade off speed and accuracy, and they recorded the average and minimum number of turns, and also win rates[4].

This work, while similar to the goal of this project, was slightly different to the approach we were hoping to take. Rather than have AI bots play against each other, the goal of our web-app is for humans to be able to play with the AI. Thus, little inspiration could be taken from this paper, however, a different paper by Koyyalagunta et al. (2021)[5], was much closer to the goal that we were trying to achieve so inspiration was taken from here.

This paper proposed an algorithm for generating Codenames clue from the language graph BabelNet or from any of several embedding methods - word2vec, GloVe, fastText or BERT. Unlike Kim et al., whose evaluation was done



purely through simulations, Koyyalagunta et al., used Amazon Mechanical Turk to carry on evaluation on humans. The algorithm proposed by Koyyalagunta et al. works as follows. In order to choose a clue for the blue team, the  $T$  nearest neighbours of each blue word is calculated. Then, they go through each subset of intended words and they add the union of the nearest neighbours for every blue word to a set of candidate clues. Then, each candidate clue is scored using a scoring function  $g(\cdot)$  which produces a large positive value for good clues and a lower value for bad clues. This scoring function will take into account the similarity between the candidate clue and the red words, so a candidate clue will have a high score if it is close to a subset of the blue words while remaining as far away as possible from the red words. In the case of no overlap between the nearest neighbours of the blue clues, the algorithm will choose a clue for one word[5]. This algorithm was implemented in our project<sup>3</sup>

### 5.3 Design

Rather than creating our own word embeddings, we used pretrained one. For word2vec the pretrained model used was trained on the Google News dataset (about 100 billion words)<sup>4</sup> and for GloVe we used Stanford's pretrained model which was trained on Common Crawl<sup>5</sup>. Using these pretrained models saved us computational time and effort to train our own models.

A UML diagram showing the attributes and methods of a **Spymaster** class is shown in Figure 5.5.

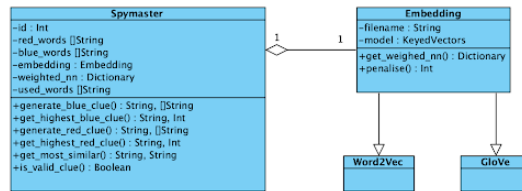


Figure 5.5: UML diagram for **Spymaster** class

<sup>3</sup><https://github.com/divyakoyy/codenames>

<sup>4</sup><https://code.google.com/archive/p/word2vec/>

<sup>5</sup><https://nlp.stanford.edu/projects/glove/>

## 5.4 Implementation

The algorithm designed by Koyyalagunta et al. relies on the use of the `gensim` python library<sup>6</sup>. This is a library that allows the pre-trained word embeddings to be loaded into python, and provides us useful methods while working with these word embeddings.

In our embedding class, we have a `get_weighted_nn()` method which gets the 500 top nearest neighbors for a given word. It also has a `penalize()` method which is used to penalize words that are similar to the opposing team's words.

Some pseudocode explaining how a clue is generated is as follows

---

**Algorithm 1** Generate Clue

---

```
priority_queue  $\leftarrow$  []
for word_pair in word_pairs do
  potential_clues  $\leftarrow$  {}
  for word in word_pair do
    potential_clues  $\leftarrow$  nearest neighbours of word
  end for
  best_clues  $\leftarrow$  []
  best_score  $\leftarrow$   $-\infty$ 
  for clue in potential_clues do
    get combined similarities score
    penalise clue
    update best score
  end for
  best_clues  $\leftarrow$  best_clues + best_scoring_clue
  priority_queue  $\leftarrow$  priority_queue + best_clues
end for
return best clue from priority_queue
```

---

We will now elaborate on the algorithm above. When a `Spymaster` object is instantiated, the chosen embedding (word2vec or GloVe) is used to create a dictionary of nearest neighbors for each possible agent. When generating a clue, a combination of every pair of remaining agents for a team is used.

---

<sup>6</sup><https://radimrehurek.com/gensim/>

Each pair of words is used as a set of chosen words where the highest scores for these words can be calculated. This is calculated by taking each word in the pair and adding all of their nearest neighbors and adding them to a set of potential clues. Then, for each clue in this set, each clue is first checked that it is a valid clue, i.e. the clue isn't an agent, it's an alphabetical word, it isn't stemmed from a different agent. Then, the nearest neighbor's dictionary is checked to see if the potential clue is there for each word in the pair of agents. If for each item in the pair, the clue was in the top 500 nearest neighbors, the similarity is taken from the dictionary and added to a list which keeps track of the scores. If it is not in the dictionary, the similarity is calculated and also added to the list of scores. After this has been completed for both words, the max similarity between the potential clue and the opposing team's agents is calculated and used as a way to penalize clues that are too closely related to the other team's words. The sum of this penalisation value is calculated and this is the score for that potential clue. This is then compared to the previous highest score, and if it is higher the current clue is set as the highest scoring clue and its score is noted. This process is repeated for each potential clue and the highest scoring clue at the end is returned.

This process is repeated for each pair of agents, and the highest score for each pair is added to a min heap so that the lowest score is at the top (note this is actually the highest score, when the score is calculated it is multiplied by -1). The top clues are and the pair of words are then returned from the removed from the min heap. In order to be sure that the same clue is not reused, each time a clue is used it is added to a list of used clues that the spymaster object maintains. New clues will only be returned if they are not present in that list, ensuring that the same clue will not be used again.

In terms of the web-application, minimal changes had to be made to introduce the AI spymaster into the game. One small change was that users are now should a *request clue* button in the clue area. Another change is that when a game ends, a list of the clues and hints from the game are shown so users can see what the AI came up with, as shown in Figure 5.6

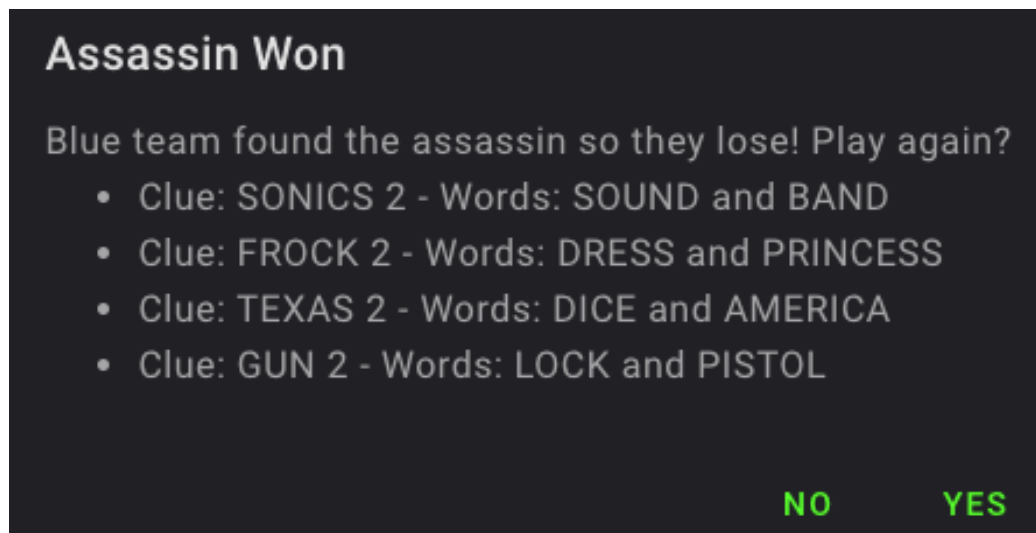


Figure 5.6: End game with AI spymaster

# Chapter 6

## Evaluation

### 6.1 Related Work

As mentioned previously in Chapter 5.2, Kim et al (2019) performed their evaluation by running a round-robin tournament of 30 games for all pairs of spymaster and operatives (the same 30 boards were presented to all pairs). For each of the spymasters, they considered 3 different threshold levels, to see how the different approaches trade off speed and accuracy, and they recorded the average and minimum number of turns, and also win rates[4].

Koyyalagunta et al. (2021) performed their evaluation using Amazon Mechanical Turk (AMT). In their paper, they describe how each AMT worker had to provide 2 to 4 board words (ranked) that they selected as most related to the given clue word. To quantify the performance of different algorithms, they calculated precision@2 and recall@4. Precision@2 measures the number of correct words that an AMT worker guessed in the first two ranks, where correct words are the intended words that the algorithm meant the worker to select for a given clue. Recall@4 measures the number of correct words chosen in the first four ranks[5].

While Kim et al (2019)’s evaluation was a good way of testing how the different AIs performed against each other, this kind of evaluation is not suitable for our project. The evaluation undertaken by Koyyalagunta et al. (2021) was a more apt evaluation for our project. However, we also wanted to evaluate how the AI performs compared to a human. We shall now explain

how we carried out our evaluation.

## 6.2 Methodology

The goals of our evaluation were to

- Evaluate how our AI performs at giving clues
- Compare the AI clues to human clues

We had to make some considerations as to how we could achieve these goals. One of the first considerations we had to make was should we evaluate both our word2vec and GloVe spymasters, or just evaluate one. We came to the conclusion that trying to evaluate both would mean that we must ask users twice as many questions, and ask them to commit twice as much time, so in the end we decided to just evaluate one of the AIs. We next had to decide which of the AIs we should use in the evaluation. We tested the AIs on multiple boards to see which one tended to produce better clues. In general, whenever both AIs produced a clue related to the same two words, the clue produced by GloVe was a better clue. If the AIs produced a clue that was not related to the same two words, the GloVe clue still tended to be the better clue overall, so we decided to use GloVe in our experiment.

Another consideration we had was how we are going to evaluate how the AI clue compares to a human clue. Initially, we had planned on showing the user two words, asking them to provide a clue for those two words, and then show them what the AI produced and ask them to select which clue they thought to be the better clue. This however, would have introduced a bias into the experiment, which would have likely resulted in the user always picking their own clue as the better clue. In order to overcome this, we decided to run our experiment in two phases. A consequence of this is that we now needed twice as many users to participate in the experiment, but as this avoided introducing any biases, we decided it was worthwhile.

### 6.2.1 Models

In order to perform our evaluations, new `Experiment` and `Evaluation` classes were created, which can be seen in Figures 6.1 and 6.2.

Experiment
-embedding : Embedding -id : String -game : Game -spymaster : Spymaster -game2 : Game -spymaster2 : Spymaster -clues : String[] -guesses : String[] -spymaster_clues : String[] -given_spymaster_clues : String
+generate_clue() : String +make_guess() +get_percision() +get_recall() +update_board() +generate_spymaster_words() +take_spymaster_clue() +update_board_spymaster()

Figure 6.1: UML diagram for Experiment class

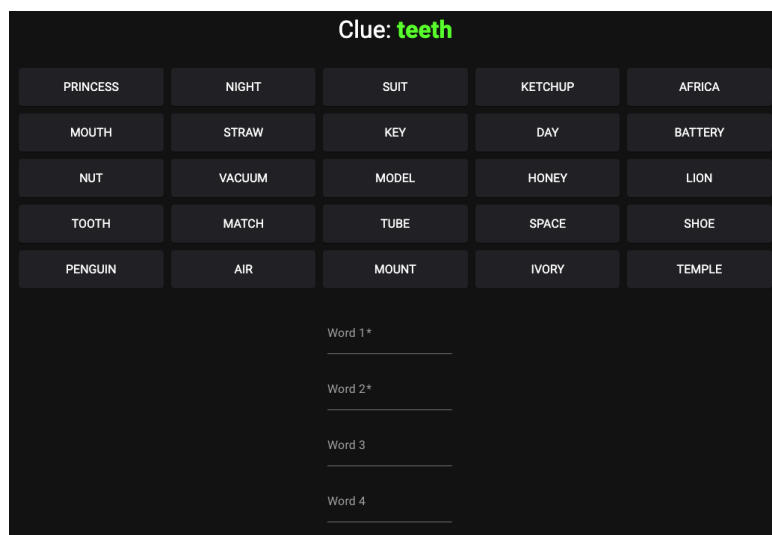
Evaluation
-exp : Experiment -votes : String[]
+get_game_words() : String[] +get_ai_clues() : String[] +get_human_clues() : String[] +calc_human_percent() : Int +calc_ai_percent() : Int

Figure 6.2: UML diagram for Evaluation class

## 6.2.2 User Trial Phase 1

The goal of this phase was to evaluate how good the AI generated clues are, and to gather some human-generated clues.

This phase of the experiment consists of two parts. In the first part, the user was presented with a board that an operative sees (all words are uncoloured). They were presented with a clue, and there were 4 input boxes where they can enter words that they believe the clue is related to. This is the same as the trial that Koyyalagunta et al. (2021) performed with the AMT workers. We used the same performance metrics they used, precision@2 and recall@4 to evaluate the AI clues. An example of what a user saw during this part is as shown in Figure 6.3



The screenshot shows a user interface for a word game. At the top, a clue is displayed: "Clue: **teeth**". Below the clue is a 5x5 grid of words. The words are: PRINCESS, NIGHT, SUIT, KETCHUP, AFRICA (Row 1); MOUTH, STRAW, KEY, DAY, BATTERY (Row 2); NUT, VACUUM, MODEL, HONEY, LION (Row 3); TOOTH, MATCH, TUBE, SPACE, SHOE (Row 4); PENGUIN, AIR, MOUNT, IVORY, TEMPLE (Row 5). Below the grid are four input fields labeled "Word 1\*", "Word 2\*", "Word 3", and "Word 4", each with a text entry line.

Clue: <b>teeth</b>				
PRINCESS	NIGHT	SUIT	KETCHUP	AFRICA
MOUTH	STRAW	KEY	DAY	BATTERY
NUT	VACUUM	MODEL	HONEY	LION
TOOTH	MATCH	TUBE	SPACE	SHOE
PENGUIN	AIR	MOUNT	IVORY	TEMPLE

Word 1\*  
 Word 2\*  
 Word 3  
 Word 4

Figure 6.3: Phase 1 part 1 view

Each user was provided with four clues, and this part of the experiment assumes that each user is an operative on the blue team. Once they had completed the first part of this phase, the second part began.

In the second part of this phase, users were again presented with a board, this time however they were shown it in the spymaster view (all words were coloured). The goal of this part of this phase was to gather human-generated clues. Each user was presented with two blue words, and asked to come up with a clue for those two words, taking the other words on the board into



consideration. There was a text input box where the user could submit their clue. Once the user had provided four clues, this phase of the experiment concluded.

An example of what the user saw during this part is as shown in Figure 6.4.

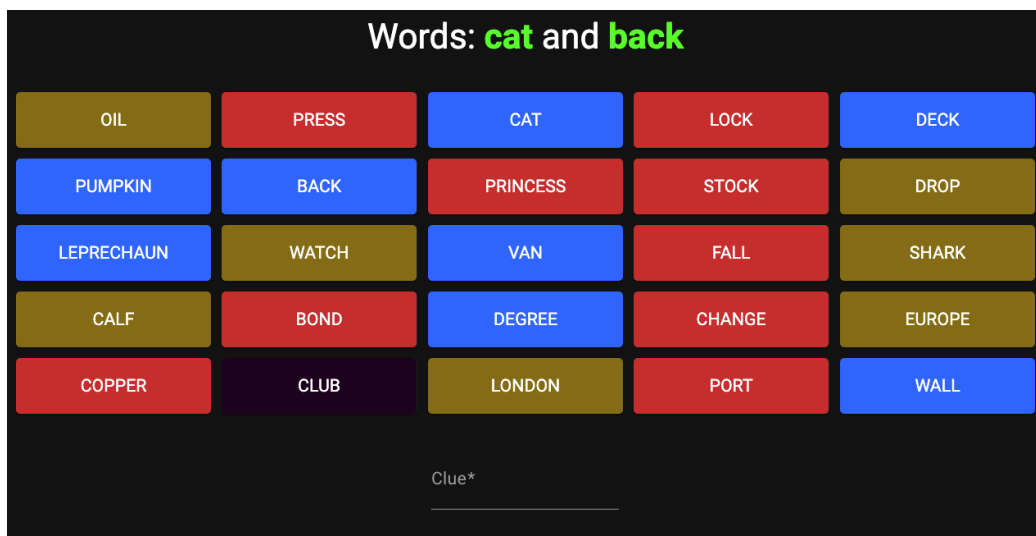


Figure 6.4: Phase 1 part 2 view

### 6.2.3 User Trial Phase 2

The goal of this phase of the experiment was to evaluate how the AI clues compared to a human clue.

In this phase of the experiment the user was presented with a board similar to the second part of the first phase of this experiment. They were again be shown two words, but this time they were also shown two clues. The users weren't be told which clue came from a human, and which clue came from the AI, and they were be asked to choose which clue they thought is the better clue.

An example of what the user saw during this phase is shown in Figure 6.5

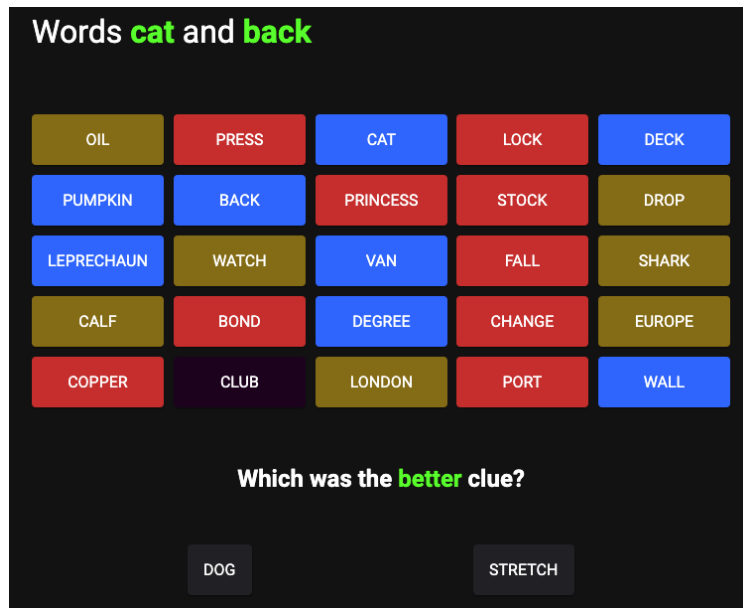


Figure 6.5: Phase 2 view

## 6.3 Implementation

In order to implement this experiment, we created a new `/experiment/` route. We reused the code that was already in place for the spymaster and operative views of the board, and added in text input areas for users to submit their answers.

Each time a user participates in Phase 1 of the experiment, a new `Experiment` object is created, and this object is responsible for providing the clues, and storing answers through its various methods. When Phase 1 has ended, the object was serialized and saved to the disk using the `pickle`<sup>1</sup> module.

During Phase 2, a `Evaluation` object is created which, similar to the the `Experiment` object used during Phase 1, is responsible for taking storing the user's supplied answers. The previously saved `Experiment` objects are loaded back into memory and they are stored stored in the `Evaluation` objects a so that the boards and clues that were previously used can be presented to the new set of users. The user as presented with two buttons where they

<sup>1</sup> <https://docs.python.org/3/library/pickle.html>

could click which clue they believe to be the better clue. When this button was clicked, a socket message was sent to the backend and their choice was saved, and the next set of words and clues were be sent back to the user. This process continued for 5 boards, and after the user has submitted their final answer, the **Evaluation** object was saved to disk the same way that the **Experiment** object in Phase 1 was saved.

## 6.4 Results

### 6.4.1 User Trial Phase 1

The boards that the first user was shown for this evaluation can be seen in Figures 6.3 and 6.4.

<b>Clue:</b>	teeth	pants	continent	leopard
<b>Expected:</b>	nut   tooth	suit   shoe	africa   space	lion   penguin
<b>Top 2:</b>	tooth   mouth	suit   shoe	africa   lion	lion   tooth
<b>Bottom 2:</b>	ivory   lion	model   day	penguin   temple	africa   suit
<b>precision@2:</b>	0.5	1.0	0.5	0.5
<b>recall@4:</b>	0.25	0.5	0.25	0.25

Table 6.1: Phase 1 part 1 results for user 1

<b>Words:</b>	cat   black	deck   wall	leprechaun   degree	pumpkin   van
<b>AI clue:</b>	dog	roof	bachelor	pie
<b>Human clue:</b>	stretch	wood	money	circle

Table 6.2: Phase 1 part 2 results for user 1

Tables 6.1 and 6.2 show the results of Phase 1 of our experiment for one user. In Table 6.1, the row labelled **Clue** refers to the clue that was presented to the user, which they were asked to try and figure out which clues it related to. The **Expected** row are the two clues that the AI was intending to link with the given clue. **Top 2** refers to the words in the first two ranks that were guessed by the user, and **Bottom 2** refers to the words in the last two ranks that were guessed by the users. The row labelled **precision@2** refers to how many of the guesses that the user made in the top two ranks were correct, while the row labelled **recall@4** refers to how many of the guesses that the user made in all four ranks were correct.

The results for Phase 1 Part 1 for the other users who partook in the experiment can be seen in Tables A.1, A.3, A.5 and A.7, along with the associated boards they were shown in Figures A.1, A.2, A.3 and A.4 in the Appendix.

In Table 6.2, the row labelled **Words** refers to the two words that the user was asked to come up with a clue for. The row labelled **AI clue** refers to the clue that the AI spymaster came up with, and the row labelled **Human clue** refers to the clue that the human provided. The results for Phase 1 Part 2 for the other users who partook in the experiment can be seen in Tables A.2, A.4, A.6 and A.8, and the corresponding boards can be seen in Figures A.1, A.2, A.3 and A.4 in the Appendix.

The average score for **precision@2** and **recall@4** across all experiments can be seen in Table 6.3.

<b>precision@2</b>	0.475
<b>recall@4</b>	0.275

Table 6.3: Average results for precision@2 and recall@4

## 6.4.2 User Trial Phase 2

Table 6.4 shows the results of Phase 2 of our experiment for one user. In this table, the row labelled **Words** refers to the words that the user was shown and asked to choose the best clue for. The row labelled **AI** refers to the clue that the AI spymaster had produced, and the row labelled **Human** refers to the clue that was provided by a human. In this table, any cell with a green background indicates that this is the clue that the user chose as the better clue. The percentage of AI clues chosen compare to human clues can be seen in Figure 6.6.

The results for Phase 2 for the other users who partook in the experiment can be seen in Tables A.9, A.10, A.11 and A.12, and the corresponding pie charts are shown in Figures A.10, A.11, A.12 and A.13 in the Appendix. The boards they were shown were the same as the boards in Phase 1 Part 2 and they can be seen in Figures A.1, A.2, A.3 and A.4 in the Appendix.

The percentage of AI clues chosen compare to human clues across all experiments can be seen in Figure 6.7.

Board 1				
<b>Words:</b>	cat   back	deck   wall	leprechaun   degree	pumpkin   van
<b>AI:</b>	dog	roof	bachelor	pie
<b>Human:</b>	stretch	wood	money	circle
Board 2				
<b>Words:</b>	mexico   europe	lemon   oil	ambulance   pants	smuggler   dragon
<b>AI:</b>	america	juice	khaki	smuggling
<b>Human:</b>	spanish	liquid	flash	treasure
Board 3				
<b>Words:</b>	lap   pole	trip   pilot	green   australia	forest   shadow
<b>AI:</b>	schumacher	flight	zealand	cloud
<b>Human:</b>	stripper	airline	forests	shaded
Board 4				
<b>Words:</b>	gold   dress	lead   key	telescope   worm	pie   plot
<b>AI:</b>	silver	crucial	hubble	cake
<b>Human:</b>	frills	essential	space	chart
Board 5				
<b>Words:</b>	revolution   china	compound   copper	row   club	mouth   kangaroo
<b>AI:</b>	chinese	zinc	football	tongue
<b>Human:</b>	coup	chemistry	sport	pouch

Table 6.4: Phase 2 results for user 1

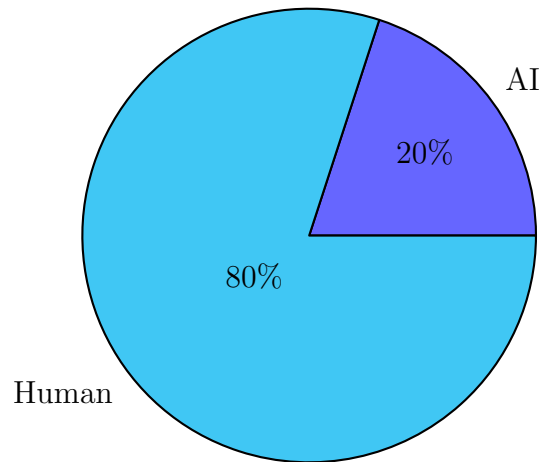


Figure 6.6: Clues chosen for Table 6.4

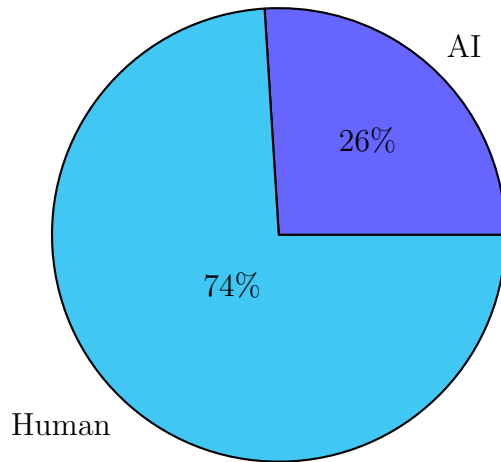


Figure 6.7: Clues chosen across all experiments

# Chapter 7

## Conclusions and Future Work

### 7.1 Conclusions

The goal of this project was to create an web-based application allowing users to play the board game Codenames online with their friends, and to allow users to play with an AI spymaster. Our vision was realised and the application is deployed for anyone to use<sup>1</sup>.

However, not only did we want to create this application, but we also wanted to evaluate how well our AI performed. We wanted to test the quality of the clues that the AI produced and to see how these clues compared to human clues for the same pair of words. The results shown in Table 6.3 show that on average, users were only able to figure out one of the words that the AI was giving a clue for. This seems like the AI did not perform as well as we had hoped, but we cannot completely blame the AI in this sense. If you look at any of the tables where precision@2 is only 0.5, if you google the clue and the expected word that the user did not figure out, more often than not you can see the connection it was trying to make, albeit if you've never heard of it before. This seems to be a flaw in our AI, that it's knowledge is too wide and it ends up making connections that even the best of humans would struggle to make.

---

<sup>1</sup><https://codenames.com>

The results of comparing the AI clues to human clues, shown in Figure 6.7 show that the AI clue was chosen as the better clue only 26% of the time. This again is a slightly disappointing result, as we would have preferred the AI to perform similarly, if not even better, than a human. This result however, seems to tie into the idea that the AI clues are not necessarily *bad*, they are just a bit too niche. It makes sense that a human would be more likely to choose a human clue, as it is probably more easier to deduce about, and also humans possess a much greater grasp of the meaning of words, so they can make interesting connections that an AI would never see or come up with.

## 7.2 Future Work

Probably the most obvious piece of future work that could be carried out in regards to this project is to try out more word embedding models, such as BERT<sup>2</sup> or fastText<sup>3</sup> like Koyyalagunta et al. used[5]. This project mostly focused on using word embeddings, so it would be interesting to implement an AI that used language graphs, such as BabelNet<sup>4</sup>, like Koyyalagunta et al. used[5]. During the time that this project was being carried out, a new version of WordNet was released<sup>5</sup>, so trying to implement an AI using this new version could make for an interesting piece of work.

In our project, we only carried out our evaluation using one pre-trained GloVe model, but more are available<sup>6</sup>. It would be interesting if we could see how each of these models compare, as they all use the same algorithm to generate the word embeddings, but they are trained on different corpora of text. Perhaps a model trained on Twitter might make more human relatable clues than the model trained on Wikipedia, and thus perform better in our evaluation. Even if we did not rerun the evaluation with different models, as a piece of future would it would be great to redo the same experiment but with a larger sample size. We only had five users partaking in Phase 1 of our evaluation, and another five partaking in Phase 2. It is hard to draw meaningful conclusions with such a small sample size. If this evaluation

---

<sup>2</sup>[https://en.wikipedia.org/wiki/BERT\\_\(language\\_model\)](https://en.wikipedia.org/wiki/BERT_(language_model))

<sup>3</sup><https://fasttext.cc/>

<sup>4</sup><https://babelnet.org/>

<sup>5</sup><https://en-word.net/>

<sup>6</sup><https://nlp.stanford.edu/projects/glove/>



was to be carried out again, we would look into using a service similar to Amazon Mechanical Turk<sup>7</sup>, like how Koyyalagunta et al. carried out their experiment[5].

Another piece of work that could be undertaken, is to create our own word embeddings, rather than use pre-trained ones. For example, we could train our own model on, for example, IMDb<sup>8</sup>, and then create boards where all of the words are movie related, and see how the AI would perform in this scenario. This sort of specialising could also have some very interesting developments in the gameplay of our application. For example, if when a user is creating a game, they could chose some specific domains that they are interested in, for example science, music or cinema, and then the AI spymaster that is chosen could try to provide hints that related to something the user is interested in. This in turn could result in better clues as the user may be more likely to get a connection between words when it is more specific to their own hobbies. Creating our own word embeddings, however, would be an extremely expensive task, both in terms of time and resources, hence why we had to use pre-trained models in our project.

As it stands, our AI has no way of explaining itself. It has the ability to propose clues, but no way of explaining *why* it found the relationship. If we could develop a way for the AI to explain its choices, that would make for a fascinating piece of work. This could perhaps be done by looking at the semantic definitions of the word, and try to reason about the relationship between them, or perhaps it could find some way of finding the the context in which the words in question appear near each other in a Wikipedia article. How exactly it would go about this sort of reasoning could be another project in itself, and developing an algorithm that could not only generate clues, but also explain why it chose these words is a very interesting piece of work we would love to carry out had we more time. This would not only help justify the AI's clues, but it would also add a much more human aspect to the gameplay, as half of the fun in playing Codenames with your friends is trying to justify how your outlandish clue was actually an excellent, well-thought-out play on words.

---

<sup>7</sup><https://www.mturk.com/>

<sup>8</sup><https://www.imdb.com/>

# Appendix A

## Appendix

SMUGGLER	CAT	HOSPITAL	LAWYER	JACK
SCORPION	CHANGE	PLOT	FIGHTER	FOREST
KNIGHT	CENTER	CONCERT	EUROPE	SHADOW
PYRAMID	LEPRECHAUN	MOUSE	WITCH	BUG
SPRING	GHOST	GREEN	ANGEL	OPERA

Figure A.1: Phase 1 part 1 board for user 2

<b>Clue:</b>	mice	performances	miguel	assassinate
<b>Expected:</b>	cat   mouse	concert   shadow	jack   angel	plot   witch
<b>Top 2:</b>	mouse   cat	opera   concert	knight   jack	opera   forest
<b>Bottom 2:</b>	ghost   forest	plot   knight	europe   fighter	knight   smuggler
<b>precision@2:</b>	1.0	0.5	0.5	0.0
<b>recall@4:</b>	0.5	0.25	0.25	0.0

Table A.1: Phase 1 part 1 results for user 2

<b>Words:</b>	mexico   europe	lemon   oil	ambulance   pants	smuggler   dragon
<b>AI clue:</b>	america	juice	khaki	smuggling
<b>Human clue:</b>	spanish	liquid	flash	treasure

Table A.2: Phase 1 part 2 results for user 2

HONEY	EMBASSY	TEACHER	ROBOT	YARD
ROULETTE	FIGHTER	ROW	BARK	PENGUIN
PORT	FISH	TRACK	HOOK	BEAR
KANGAROO	ICE	BUTTON	SHADOW	AFRICA
WHIP	PIPE	CRICKET	SUIT	AUSTRALIA

Figure A.2: Phase 1 part 1 board for user 3

<b>Clue:</b>	south	animals	mouse	wooden
<b>Expected:</b>	africa   australia	fish   bear	kangaroo   button	yard   whip
<b>Top 2:</b>	africa   australia	bear   penguin	cricket   australia	cricket   yard
<b>Bottom 2:</b>	ice   bear	kangaroo   fish		track
<b>precision@2:</b>	1.0	0.5	0.0	0.5
<b>recall@4:</b>	0.5	0.5	0.0	0.25

Table A.3: Phase 1 part 1 results for user 3

<b>Words:</b>	lap   pole	trip   pilot	green   australia	forest   shadow
<b>AI clue:</b>	schumacher	flight	zealand	cloud
<b>Human clue:</b>	stripper	airline	forests	shaded

Table A.4: Phase 1 part 2 results for user 3

<b>Clue:</b>	feet	feces	gardens	sarus
<b>Expected:</b>	face   foot	sock   litter	park   grace	platypus   crane
<b>Top 2:</b>	foot   sock	millionaire   lion	park   crane	nurse   ambulance
<b>Bottom 2:</b>	litter   lion	crane   platypus	log   club	ring   fork
<b>precision@2:</b>	0.5	0.0	0.5	0.0
<b>recall@4:</b>	0.25	0.0	0.25	0.0

Table A.5: Phase 1 part 1 results for user 4

NURSE	PARK	LION	FORK	ROBOT
GENIUS	GRACE	RING	CLUB	FACE
SOCK	PIANO	TICK	BATTERY	CROWN
PITCH	AMBULANCE	PLATYPUS	FOOT	CRANE
LITTER	POLICE	MILLIONAIRE	LOG	CROSS

Figure A.3: Phase 1 part 1 board for user 4

<b>Words:</b>	gold   dress	lead   key	telescope   worm	pie   plot
<b>AI clue:</b>	silver	crucial	hubble	cake
<b>Human clue:</b>	frills	essential	space	chart

Table A.6: Phase 1 part 2 results for user 4

TAG	MASS	LION	PLATE	EUROPE
SATELLITE	BACK	BELT	CROSS	VET
DRILL	BOARD	SHOT	GROUND	TAP
POST	DISEASE	TELESCOPE	PLASTIC	PAPER
RACKET	CLUB	DRESS	BILL	JACK

Figure A.4: Phase 1 part 1 board for user 5

<b>Clue:</b>	packaging	legislation	racquet	hubble
<b>Expected:</b>	plastic   paper	europe   bill	plate   racket	vet   telescope
<b>Top 2:</b>	post   paper	bill   board	racket   shot	satellite   telescope
<b>Bottom 2:</b>	plastic   tag	club   europe	club   ground	ground   mass
<b>precision@2:</b>	0.5	0.5	0.5	0.5
<b>recall@4:</b>	0.5	0.5	0.25	0.25

Table A.7: Phase 1 part 1 results for user 5

<b>Words:</b>	revolution   china	compound   copper	row   club	mouth   kangaroo
<b>AI clue:</b>	chinese	zinc	football	tounge
<b>Human clue:</b>	coup	chemistry	sport	pouch

Table A.8: Phase 1 part 2 results for user 5

OIL	PRESS	CAT	LOCK	DECK
PUMPKIN	BACK	PRINCESS	STOCK	DROP
LEPRECHAUN	WATCH	VAN	FALL	SHARK
CALF	BOND	DEGREE	CHANGE	EUROPE
COPPER	CLUB	LONDON	PORT	WALL

Figure A.5: Board 1 for Phase 2

ROSE	FAIR	AMBULANCE	HOLLYWOOD	WIND
HAM	VET	CHEST	LEMON	ROME
SMUGGLER	DRAFT	MEXICO	OIL	PANTS
MOON	ROCK	EUROPE	KEY	WITCH
WHALE	COMPOUND	FILM	DRAGON	WAVE

Figure A.6: Board 2 for Phase 2

WEB	KID	FOREST	GREEN	MAIL
BED	CRANE	TRIP	LAP	PILOT
SHADOW	CHANGE	OIL	DRAFT	CENTAUR
PASTE	PIT	AGENT	HAND	AUSTRALIA
CALF	PRESS	PART	POLE	THIEF

Figure A.7: Board 3 for Phase 2

BUGLE	BALL	PART	GOLD	DRESS
STRAW	PIE	AMBULANCE	ROSE	SHARK
TAP	LEAD	PAN	GAME	TELESCOPE
WORM	HELICOPTER	THIEF	PLOT	PASTE
LIMOUSINE	KEY	PAPER	CHEST	HEAD

Figure A.8: Board 4 for Phase 2



Figure A.9: Board 5 for Phase 2

Board 1								
Words:	cat	back	deck	wall	leprechaun	degree	pumpkin	van
AI:	dog		roof		bachelor		pie	
Human:	stretch		wood		money		circle	
Board 2								
Words:	mexico	europe	lemon	oil	ambulance	pants	smuggler	dragon
AI:	america		juice		khaki		smuggling	
Human:	spanish		liquid		flash		treasure	
Board 3								
Words:	lap	pole	trip	pilot	green	australia	forest	shadow
AI:	schumacher		flight		zealand		cloud	
Human:	stripper		airline		forests		shaded	
Board 4								
Words:	gold	dress	lead	key	telescope	worm	pie	plot
AI:	silver		crucial		hubble		cake	
Human:	frills		essential		space		chart	
Board 5								
Words:	revolution	china	compound	copper	row	club	mouth	kangaroo
AI:	chinese		zinc		football		tongue	
Human:	coup		chemistry		sport		pouch	

Table A.9: Phase 2 results for user 2

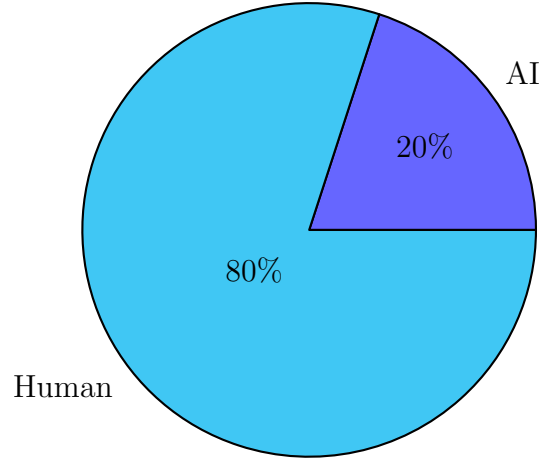


Figure A.10: Clues chosen for Table A.9

Board 1							
Words:	cat   back	deck   wall	leprechaun   degree	pumpkin   van			
AI:	dog	roof	bachelor	pie			
Human:	stretch	wood	money	circle			
Board 2							
Words:	mexico   europe	lemon   oil	ambulance   pants	smuggler   dragon			
AI:	america	juice	khaki	smuggling			
Human:	spanish	liquid	flash	treasure			
Board 3							
Words:	lap   pole	trip   pilot	green   australia	forest   shadow			
AI:	schumacher	flight	zealand	cloud			
Human:	stripper	flight	forests	shaded			
Board 4							
Words:	gold   dress	lead   key	telescope   worm	pie   plot			
AI:	silver	crucial	hubble	cake			
Human:	frills	essential	space	chart			
Board 5							
Words:	revolution   china	compound   copper	row   club	mouth   kangaroo			
AI:	chinese	zinc	football	tongue			
Human:	coup	chemistry	sport	pouch			

Table A.10: Phase 2 results for user 3



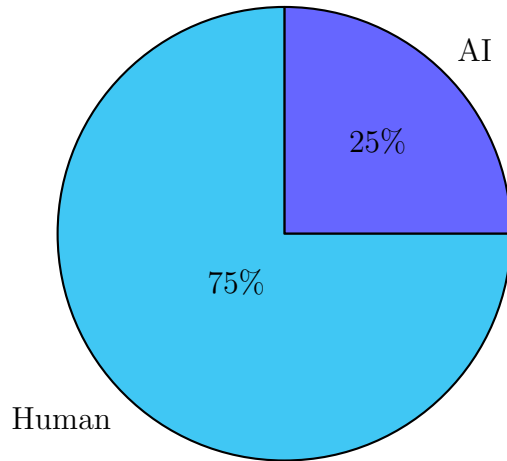


Figure A.11: Clues chosen for Table A.10

Board 1					
<b>Words:</b>	cat   back	deck   wall	leprechaun   degree	pumpkin   van	
<b>AI:</b>	dog	roof	bachelor	pie	
<b>Human:</b>	stretch	wood	money	circle	
Board 2					
<b>Words:</b>	mexico   europe	lemon   oil	ambulance   pants	smuggler   dragon	
<b>AI:</b>	america	juice	khaki	smuggling	
<b>Human:</b>	spanish	liquid	flash	treasure	
Board 3					
<b>Words:</b>	lap   pole	trip   pilot	green   australia	forest   shadow	
<b>AI:</b>	schumacher	flight	zealand	cloud	
<b>Human:</b>	stripper	airline	forests	shaded	
Board 4					
<b>Words:</b>	gold   dress	lead   key	telescope   worm	pie   plot	
<b>AI:</b>	silver	crucial	hubble	cake	
<b>Human:</b>	frills	essential	space	chart	
Board 5					
<b>Words:</b>	revolution   china	compound   copper	row   club	mouth   kangaroo	
<b>AI:</b>	chinese	zinc	football	tongue	
<b>Human:</b>	coup	chemistry	sport	pouch	

Table A.11: Phase 2 results for user 4

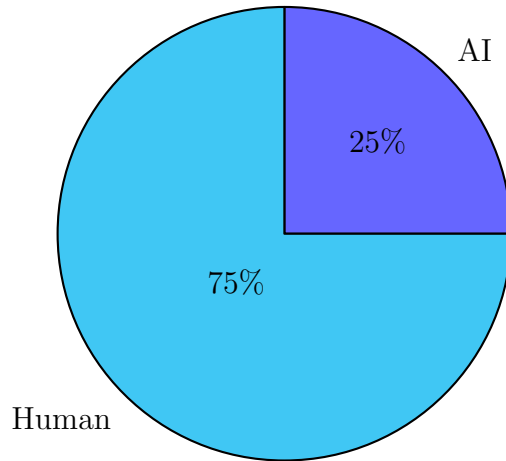


Figure A.12: Clues chosen for Table A.11

Board 1					
<b>Words:</b>	cat   back	deck   wall	leprechaun   degree	pumpkin   van	
<b>AI:</b>	dog	roof	bachelor	pie	
<b>Human:</b>	stretch	wood	money	circle	
Board 2					
<b>Words:</b>	mexico   europe	lemon   oil	ambulance   pants	smuggler   dragon	
<b>AI:</b>	america	juice	khaki	smuggling	
<b>Human:</b>	spanish	liquid	flash	treasure	
Board 3					
<b>Words:</b>	lap   pole	trip   pilot	green   australia	forest   shadow	
<b>AI:</b>	schumacher	flight	zealand	cloud	
<b>Human:</b>	stripper	airline	forests	shaded	
Board 4					
<b>Words:</b>	gold   dress	lead   key	telescope   worm	pie   plot	
<b>AI:</b>	silver	crucial	hubble	cake	
<b>Human:</b>	frills	essential	space	chart	
Board 5					
<b>Words:</b>	revolution   china	compound   copper	row   club	mouth   kangaroo	
<b>AI:</b>	chinese	zinc	football	tongue	
<b>Human:</b>	coup	chemistry	sport	pouch	

Table A.12: Phase 2 results for user 5

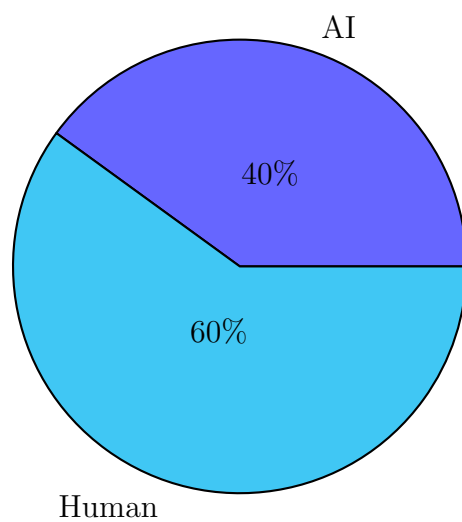


Figure A.13: Clues chosen for Table A.12

# Bibliography

- [1] P. A. Tsividis, J. Loula, J. Burga, N. Foss, A. Campero, T. Pouncy, S. J. Gershman, and J. B. Tenenbaum, “Human-level reinforcement learning through theory-based modeling, exploration, and planning,” *arXiv preprint arXiv:2107.12544*, 2021. <https://arxiv.org/pdf/2107.12544.pdf>.
- [2] X. Rong, “word2vec parameter learning explained,” *arXiv preprint arXiv:1411.2738*, 2014. <https://arxiv.org/pdf/1411.2738.pdf>.
- [3] J. Pennington, R. Socher, and C. D. Manning, “Glove: Global vectors for word representation,” in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543, 2014. <https://nlp.stanford.edu/pubs/glove.pdf>.
- [4] A. Kim, M. Ruzmaykin, A. Truong, and A. Summerville, “Cooperation and codenames: Understanding natural language processing via code-names,” in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 15, pp. 160–166, 2019. <https://ojs.aaai.org/index.php/AIIDE/article/view/5239/5095>.
- [5] D. Koyyalagunta, A. Sun, R. L. Draelos, and C. Rudin, “Playing code-names with language graphs and word embeddings,” *Journal of Artificial Intelligence Research*, vol. 71, pp. 319–346, 2021. <https://arxiv.org/pdf/2105.05885.pdf>.