Roman Herrera, Valentina Vallalta, Alina Saratova, Kobe Lin

Your initial vision and its evolution:
We initially thought we could create a computer player that would give clues based on the words and would eliminate the need for a game master. Unfortunately, we realized pretty quickly that the computer was a lot worse and making connections between words on the board. We decided to leave it and start working on a computer player that would guess words given the clue. We created a board in python which we printed out every time so it would be easier to visualize the game. We had two boards: one was for player view and the other was for spy master view. We then would come up with a clue and a number and let the computer figure it out from there. We realized that even this approach was a little funky because the computer was sometimes not on the same page as us. For example: we put in the clue “vaccine” and it selected “needle”, which was expected, and “ketchup”, which was not expected. A huge part of this process was figuring out how to score connections between words so we can get the highest connections but at the same time not have it be against the rules. For example, if one of the words was “cat” we would not be able to give the clue “cats”. In order for the computer to make a guess, we went through the words and used a word model to see what the similarities were.

Finally, we went back to our original goal and tried to implement a give clue function, but it was still not great. It was difficult to fine tune it and it was really a hit or miss. Nevertheless, we could now implement the option to play as either the spymaster or the player now, which we did. We also added a feature where the player could see the already chosen words and their respective colors and the spymaster saw the already chosen words whitened out so they could just focus on the other words on the board. Although this game is definitely not on par as real life players, it was very interesting to explore how computers handle word relationships and laugh at the hilariously bad choices the computer made along the way. We learned a lot from implementing the game and troubleshooting a lot of errors that had to do with strings.

The progress you made -- and the final system's capabilities:
We were able to complete a lot of the goals we originally set out to do. We were able to successfully simulate a Codenames game with the computer playing as spymaster for both teams and playing as an operator for both teams. We first were able to generate a valid codenames board by getting a csv file of all possible words and randomly generating a board. Then we used wordtovec and the Google API to write a function that produces a hint for a given team and board, which simulates a spymaster’s turn, and a function that gives a specified number of guesses given a board, which simulated a player's turn. This way people can choose to play as either spymaster or crewmate and the computer fulfils the other role. Additionally, throughout our code, we had to make many limitations to the hints that could be given and how turns would go in order to adhere to the Codenames rules and think about how to code these restrictions into our game so that it could be as close as possible to an actual codenames game. The place where this was most important was when generating a hint so that the hint could be valid in terms of the rules. Then, even though we weren't able to make a flask interface to make

it easier, we made print functions that print the crewmate view of the board or the spymaster view of the board accordingly with corresponding colors where applicable so that users could still play and visualize the board through the terminal.

At the very end we encountered a problem with the terminal even though it was definitely not showing this error before. We had to configure our VsCode settings to get it to work properly.

How you'd extend the project further, if there were time/opportunity:
One thing we would do to extend our project if we had the time would be to create a Flask interface for our project. Our project currently is able to run fully in the terminal, such as generating a (colorful) board, playing a game, inputting clues, etc. However, being able to put our project into a web interface would be pretty cool, and might make it more accessible to people who are inexperienced with or not comfortable with working in a terminal environment. Plus, it would give us the opportunity to be more creative with the design of our project in how things look. Another way we would extend our project is when we are looking at scoring word similarities with each other, we would want to check various capitalizations of words. When working, we found that the capitalization of words would change the outcomes of our similarity scores. As such, our current workaround is to check everything as is, but with more time, going through and implementing a way to better check this would be nice. Another way we could extend our project further is by testing different word models against each other. This would be a pretty fun way to see which model is “better” at Codenames, and just have fun making different models compete with one another. Another thing that would be nice to try adding is keeping track of previous clues that have been given so that the model would not give the same clue for the other team. Lastly, another thing that we would like to figure out is some way to have a permanent dataset stored so that we wouldn’t have to keep re-running everything on every startup. Right now, this process takes a significant amount of time to wait for the dataset models to be loaded in for use. Our workaround is that once it has been loaded, we are able to keep using it within the same “ipython” environment. However, if we were to push this for more widespread use with more people, this start-up process would be annoying to sit through everytime.

How/whether you'd tackle the project differently in retrospect:
We were pretty pleased with the way we approached the project. The order in which we developed the entire project was reasonable and effective, working on one function at a time. However, there was one major thing that we debated doing differently the entirety of our project: implementing classes. When we were brainstorming how to approach our project, we decided on the functionality we wanted, such as being able to guess a word, give a word, and generate a board. We disagreed on how best to implement this though. Some of us thought classes were the best approach and others thought we should just type up some functions initially to get started. We ended up just typing up some functions, like generate board and various helper functions. As we progressed further, one of our members strongly suggested implementing classes, such as a board class, guesser class, and clue_giver class, to clean up a lot of our inputs and outputs of our functions. For example, as we added more functionality, we had to go back and retroactively add more parameters to many functions, like give_clue. Soon, we had to

keep track of far too many things between calls of functions, where it would have been easier to just have a class keep track of all that information for us. However, we had made too much progress before we really felt that we needed the class. Due to this, we found it was likely just, in the short-term, easier to just finish than to go back and convert everything into classes. That being said, we did find that the last parts we coded were much more complicated than they needed to be.

Something much smaller that we wished we did more of was create helper functions to help reduce redundant code. A few of our functions, like give_clue or guess_clue, got particularly lengthy and definitely could be made smaller by helper functions. Again, classes would've helped with this process so we didn't feel as encumbered by the amount of info we were passing into these smaller helper functions. Overall though, we thought we approached the project well, except for not using classes and helper functions to organize our code.