WinCodenames.com

Github Repo

Final Project for ISMT S-117
Andy Bryant
August 7, 2020

Summary

Win Codenames is a clue generator for the board game Codenames that uses multiple word embeddings (GloVe, Word2vec, and fastText) to calculate various distance-based metrics for each legal word and rank the results using an SVR model trained on human-in-the-loop gameplay. This was accomplished with a bespoke frontend interface, populated with boards and clues (generated offline by a processing module) that were then served by a backend system. There is still a great deal of work to do to improve this project, but I'm proud of the progress I've been able to make so far.

Game

If you're not familiar with the game, here is a great video summary. You can also play an online word-based version here.

Background

This project has been an interest of mine since I first enrolled in the course. I've done a good amount of research on how I might approach it and found a number of articles and code repos out there that demonstrated solid functionality. However, only a few of them appeared to embed into an interactive environment (i.e. online game that humans could use), consider real gameplay contexts (e.g. legal board sizes, how humans actually play the game and come up with clues, etc.), and they didn't produce results that were particularly compelling. I hoped to address all of these in my project.

Here are some examples:

- Solid math with GloVe embeddings. The calculations were great and I ended up using a lot of them, but there wasn't any consideration for initial word selection, nor was there ample time spent on ranking the resultings clues after the fact. I saw a lot of improvement here and think I succeeded in building on it.
- Nuanced approach with Word2vec. This person used the NLTK library and Word2vec embeddings to recursively find related words through their connections with each other, rather than making calculations on every word in the corpus. It's a good approach, one that I explored, but ultimately I found that introducing a stopping function into the brute force method used in the first example was sufficient.

- Another Word2vec example. I got some good info here, especially about the word classifier. But again, this example (and the others) do not account for the features of actual gameplay.

Another major shortcoming of the above approaches was that the fixed word embeddings did not account for multiple senses, so homonyms and words with n-interpretations would be flattened into a single vector. So I decided to explore the possibility of using multiple embeddings in the same algorithm, hoping that their distinct representations might equate to a broader "meanings" that the program (and ultimately the human) could choose from.

Proposal

I formed this hypothesis, presented it in my project proposal, and got some great feedback from Stuart. Here are a few of his concerns/suggestions and how I addressed them (paraphrasing for brevity):

- *Get data from real boards...*

I generated 300 boards from real Codenames words, as well as their resulting clues for people to choose from. I wasn't able to extract boards from real gameplay due to time constraints, but hopefully this is a sufficient stand-in.

- *Comparing Word2vec and GloVe embeddings probably isn't enough...*

Instead of comparing the two, I pivoted to including them both, along with fastText, in the processing. I also removed the part that would decode the vectors at the end, opting instead to use humans for validating the output.

- *Think of specific ways that humans play...*

I played a few games with this in mind and thought through my own decision-making process. Which friend words on the board do I focus on and how many (another specific point he brought up)? How much do I try and avoid assassin words and foe words? How do I ultimately rank my clue options? How much time do I spend generating clues? I address all of these in the code.

- *Testing/validating will be difficult…*

Indeed! I devised a hypothetical scenario, as suggested, and included humans to test, validate, and ultimately train another model.

With this great feedback (and more), I updated my hypothesis and designed an approach for how to move forward. I'll talk through that plan, as well as how it changed, by way of the code itself.

Code

This code includes three repositories - process, frontend, backend - that have been merged into one for the sake of sharing in one zip file.

Note: the frontend and backend are not particularly suited for this purpose, especially the frontend (built with javascript). I am including environment variables and other hidden files that would otherwise be shared securely or embedded in a runtime hosting environment. If there are issues running on your machine, please use the frontend and backend links to running services as proof of development.

In addition, I've included a requirements.txt file at the top of file in addition to those in the child directories. You only need to install the top-level file, but I've decided to keep the other ones, as they are present in the respective modules on github. I've also included a virtualenv (win-codenames/codenames-env) if you'd like to use that.

Process Module
This handles all of the processing for the project and is broken up into three sub-modules: vectors, games, and reviews. I'll review them in that order.

(directory structure)
win-codenames

- process
    - vectors
    - games
    - reviews

Vectors Sub-module
Here I import the different embeddings as text files and process them for use elsewhere in the application. You might be wondering why I would use text files, rather than import GloVe, word2vec, and fastText from existing libraries, like Spacy or HuggingFace. Looking back, this may have been the better option, but at the start I found it very difficult to inspect and modify these models. To get as close to the true data structures, I decided to download the files themselves and process them accordingly. I got GloVe and fastText from here. And I downloaded word2vec through Gensim (the steps I followed are in the comments of generate_vectors.py).

First, I decided to cut down their size. I processed them offline in the terminal, taking only the top 200,000 embeddings. I then found all of the common words among GloVe, fastText, and Word2vec, which ends up being about 28,000 in total (average English-speaking adults known between 25,000-35,000 words, so I thought this was sufficient). My thinking was that having a common set of words in the different vectors would make the generated metrics more consistent. It's also worth mentioning that I chose these three because they have a pretty good diversity of input - Google News, CommonCrawl, and Wikipedia, respectively - all in 300 dimensions. For the sake of this project, I thought this represented enough variance.

To execute this code, run win-codenames/process/vectors/generate_vectors.py. It will create three pickle files with the processed embeddings for use by the games sub-module.

Games Sub-module
This code is the heart of the project. Basically, it loops through all of the common words in the three vector files and ranks them based on metrics related to their value in the game. This is done for each embedding type and then they are sorted based on their combined values and their ranking by the SVR model with the top 5 being returned.

Here are the steps in more detail:

1. Randomly select and categorize words for the game, using a list of actual Codenames words. This would include Friends cards (the ones you want to guess), Foe cards (the ones you want to avoid), an Assassin card (the one you really want to avoid), and neutral cards (the ones that should be avoided, but don't have a huge penalty).
2. Pick the "Top Friend" cards. These are the ones that you want to favor in your clue giving. As per Stuart's request, I thought through how humans play this game and realized that the examples online don't ever select a small subset of the Friend cards to focus on, but rather give clues for the whole board. Instead, I am clustering all of the Friend cards with a given threshold (just above the mean of the cosine distances of them all) and selected those in the first cluster. This is usually 2-3 but can sometimes be 1 or even 5. This process can be improved by finding a better threshold and perhaps changing the clustering metrics (right now it is only cosine distance hierarchies).
3. Loop through all of the possible clue words (~28,000 in total) and calculate their distance metrics, based on GloVe vectors. These metrics include the following (sorted in this order because that's the way I thought that was an approximation of how humans do it):
    a. Goodness. The difference between the distance to the bad cards (Assassin and Foes) and the Top Friend cards.
    b. Bad Minimax. The difference between the minimum distance value of the bad cards and the maximum distance value of the Friend cards.
    c. Neutrals Minimax. The same as above but with the Neutral cards.
    d. Variance. The difference between the greatest distance and smallest distance to the Friend cards. My thinking here was that if a candidate clue word had a wider range, it might be closer to a bigger spread of words, which could be useful to a human playing the game. With one vector space, this might not be as useful, but when combining multiple, there is a compounding effect, which might become particularly helpful in capturing multiple meanings of words (or at least favoring those words that might be closer to more words in multiple vector spaces). It's a hypothesis that deserves more exploration and testing.
    e. I put in a stopping measure which returns na values if the assassin calculation is poor (i.e. it's too close to the assassin card). This was another improvement on the other implementations, decreasing the scored words by thousands each time.
4. If the user sets DEEP=True, then the algorithm, does the same calculations for Word2vec (I refer to them as Google embeddings in the code) and fastText embeddings. But only the top 2000 candidate words are used in these subsequent calculations, so

that the program doesn't have to loop through 28k candidates again. This was another reason that I chose to only have the same words in each vector space.

5. All sets of words are then combined and their values are normalized based on the number of embeddings used. This creates a conglomeration of the three, which is then sorted again. In this iteration, sorting is done by rank, which is their popularity in the English language, goodness, and the variance metric I proposed above (these sorting lists, along with other configuration settings, can be found in games_config.py). That was another shortcoming of the other solutions - none of them proposed sorting/ranking the returned clue words by their overall popularity.
6. If the user sets USE_MODEL_PREDICTIONS=True, then the results of this second sort are fed into a stored SVR model and predictions are made, which result in a third sort. The predictions are basically the probability that the word (i.e. all of its metrics) will be in the human-chosen category.
7. The final five clues are added to a file, along with the board words (Friends, Foes, etc.) to create a Game record that can then be used by the frontend.
8. I also have an option to save the output as a result record (CSV) for easier reading: create_result_csv=True

I created 300 Game records (with corresponding clues) for use by the frontend and stored them in an MLab instance (running MongoDB).

To execute this code, run win-codenames/process/games/generate_games.py. It will create as many games you set in num_games at the top of the file (it's currently set to 1).

Reviews Sub-module

When people rate a clue on the frontend (I'll go into this in more detail later), their "review" is stored in the database. They are all pulled down when the script in this sub-module is run, the results of which are used to train an SVR model, which is saved and used in the Games sub-module.

You can see an example of a review record here. Basically, it includes the word that the reviewer selected, along with some configuration info that I thought might be useful in posterity (whether or not the clues were shuffled before display, how many were displayed, etc.). This is compared to the corresponding game, which includes the clues that the algorithm generated, as well as its own configuration data. If the human selected the same word, then that set of values (rank, goodness, bad_minimax, etc.) are labeled with a 1. If not, then they are labeled with a 0.

I used this labeled data to fit an SVR model, using hyperparameters that were determined through multiple rounds of randomized and grid search with cross-fold validation. Ultimately, I decided to decrease gamma significantly, so there was more variance in the output, the idea being that I would be able to distinguish records from each other through their probability that they'd be selected by the user. This is why I used an SVR, as opposed to other classification methods: I needed the output of the prediction to be a probability distribution, not a binary classification. You'll notice that the r2 score is negative, which I'm interpreting to mean that the model does not follow the trend of the data (i.e. it can be inferred that it's technically worse than not having the model at all). I found a number of ways to increase r2 and accuracy, including increasing gamma and tweaking some of the other hyperparameters, but ultimately these made the results worse (be reviewing them anecdotally). In other words, when I elected to use this trained model in the final sort of the games sub-module, it chose words that I thought were better than the first two sorting algorithms came up with, despite the poor r2 and accuracy scores. But this is only from a small subset of game outputs (I probably reviewed 30-40 by hand and had no other people involved).

All that is to say that a great deal of work still needs to be done here to improve the output of the model. Future work might include:

1. Increasing amount of data. I got around 19 people to do 300+ reviews. Multiplying both of those by 10 or even 100 would be much better.
2. Decreasing variance in responses. There is a lot that can be done from a user experience perspective to make the responses more consistent (not the reviews themselves, but the way that the reviews were arrived at). For example, there isn't a tutorial or real training on how to use the interface.
3. Additional data. Perhaps I can add a textbox for user feedback or even allow them to add their own clues.
4. Use a different model type. I explored training a regression classifier or an ensemble classifier then calculating a probability distribution from them.

But ultimately, I think the research question is poorly designed. If the goal is to create a clue generator for gameplay, then more should be done to design the review/feedback system around a real game experience. For example, in this system, we ask "Do you think this is a good clue for this board?" But really, the better way to test would be providing a clue and checking if humans can guess the words from it in a real gameplay environment (i.e. their ability to guess will be the ultimate answer of whether or not it's a good clue, as is the case in a real game). We might also include humans in the clue generation, such as getting them to select the top friends words first.

To execute this code, run win-codenames/process/reviews/train_model.py

Frontend Module

This has all of the code for the frontend interface, which is built with vue.js. I built this for a few reasons:

- Primarily, I wanted them to label the data for training purposes. Each time they selected a word and submitted, a review record was created with the relevant information (described above).
- I wanted the users to have a familiar, game-like environment to review the output, hoping that their answers might be more genuine.

- I wanted to get feedback on the clues themselves (I had a lot of follow-up conversation on the phone and over text about what they thought).

This code is hosted on a free Netlify account. If you want to run it locally, you'll have to make sure you have node and yarn installed, then go to the frontend directory (win-codenames/frontend) and run ```yarn serve``` in the terminal. In order to have the data displayed, you'll have to also run the backend at the same time (explained below).

Backend Module

This is a simple Flask application that connects to the MongoDB instance and serves the data to the frontend. It is hosted at pythonanywhere (this is an endpoint that returns a random game). If you want to run this code locally go to the backend directory (win-codenames/backend) and run ```flask run```. If you navigate to the provided url, you'll see "Not Found." But don't worry, I just didn't put a default page. If the frontend is running, you'll now see the board populating.

Results

This was a good exercise, but ultimately the research question and the proposed solution were both flawed. I think I was too focused on generating good clue words with the different embeddings that I failed to develop a gaming experience that over time would train a model to create even better clues than the vectors ever could alone. To address this, I suggest using the clue generation module to assist humans, who are actually playing the game, and then have the model train on the round-by-round results.

That being said, here are the numbers:

- 19+ reviewers (some didn't provide their names, but I know they were distinct, so it was probably closer to 25) provided 316 clues.
- Humans and computers selected the same top word 15% of the time.
- Humans thought there were no good clues 23% of the time.
- These numbers were lower when GloVe vectors alone were used, but not significantly. Given the extra processing time it might not be worth it to use fastText and Word2vec (at least not in the current implementation).

Some anecdotal evidence from reviewing the results myself:

- It seemed like fastText vectors on their own gave the worst clues overall, but when they were good, they were *really* good. My feeling is that because they were the result of a common crawl, each embedding had a broader sense, so the diversity was higher and oftentimes too high to be specific in this particular context.
- Word2vec and GloVe seemed pretty comparable. I ended up going with GloVe because they are newer embeddings and there appeared to be more recent literature on them.

And some feedback from the users (paraphrasing slightly, as many of these conversations were conducted without being recorded):

- "Some of the words don't have anything to do with each other. I wouldn't have selected these as the top friends at the start of a game."
- "These clues are amazing!"
- "Three word clues are very hard."
- "I really like the idea! I know a lot of work has looked at gamifying labeling tasks for ML work and playing the game reminded me a lot of that. I've also been reading a lot about explainable AI lately and I really started thinking about how to leverage some of that towards making it more game like without necessarily jeopardizing the task."

Finally, here are some of my favorite clues! You can see more extensive output here.

Top Clue	Top Friends	Low Friends	Assassin	Foes	Neutrals
flame	torch, pan	charge mouth bottle orange time mouse fighter	chocolate	phoenix laser trunk carrot spine apple skyscraper leprechaun	pipe compound track mole theater pass mug
diagnosis	microscope disease	fan star table beat marble cook litter	slip	sound cycle maple horn flute pole tag chocolate	rose press berry lawyer octopus scorpion chest
garb	cloak suit	school lead star fish link satellite	brush	knife thumb gas plate lab pool wall dwarf bug	key glass ball missile ghost mole princess
headmaster	teacher ruler	string soldier marble mass key robot	microscope	slip helicopter shop model heart thief band cook jet	lawyer doctor mail bank revolution bow dice
medic	soldier ambulance	sink plate mole track boot slip	lead	battery pirate tablet crash telescope honey pitch scientist stadium	shop nurse log deck eye needle pole

extraterrestrial	alien leprechaun	tick club bank mouse olive trunk	hospital	point worm ice fire tooth cycle kangaroo cap sink	soldier horse dinosaur soul swing spy mole

Conclusion

There is much more to do. But I'm glad I was able to spin up a working web application, run a basic user study, train and deploy a model, and write up a report in less than two weeks. Thanks for the opportunity, guidance, and for an overall great course!