(Spy)Mastering Codenames

By: Arjun Arora, Heidi Chen, Daniel Classon

CS 221 Fall 2018 P-Progress

Overview and Model Description

We are training a model to emulate the Spymaster in the game Codenames. Our current model uses unsupervised learning and stochastic gradient descent to take in a Codenames board and output a clue word associated with all blue words. We decided to conduct our algorithm unsupervised because labelling enough permutations of Codenames boards would take too much time, and because there is usually more than one “best” or valid clue per board. To avoid such hours of human-training and labeling boards --i.e. supervised learning -- we developed a custom loss function in lieu of a human judge.

We first reduced the problem to ignore constraints on our human guessers’ intelligence & subjective ability to evaluate clues. We defined words as their word2vec vectors from the gensim package and pre-trained Google News codec, which contains 3 million words, each of dimension 300. The heuristic that we use in our model is minimizing the cosine distance between positively-associated blue words and maximizing the cosine distance between negatively-associated other words. We additionally weight the game-ending assassin word more negatively, such that any solution that minimizes loss is far away from the assassin word.

We chose stochastic gradient descent because it allows us to tinker with our optimal loss function and train for thousands of cycles unsupervised.

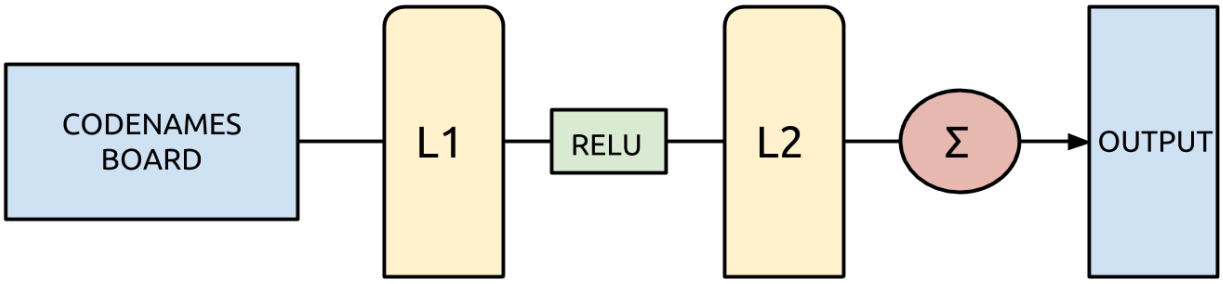
First-Draft Loss Function:

$$Loss(board,\ hint)\ =\ log(\frac{1}{m}\sum_{r\ in\ red\ words}^{m} cosineDist(r,\ hint))\ + log(\lambda * cosineDist(assassin,\ hint))$$

$$- log(\frac{1}{n}\sum_{b\ in\ blue\ words}^{n} cosineDist(b,\ hint))$$

One challenge in implementing our pipeline is that our model outputs an optimal word-vector, rather than a word, and word2vec is not an invertible function. In other words, every word maps to a word-vector in a higher-dimension, but a word-vector can be created that does not map back to a word in our codec. To solve this first issue, we created a utility function that translates the word-vector output by our model during training and testing into the nearest word in the codec. We defined “nearest word” as the word in the codec that minimizes the cosine-distance between our output word-vector. One point that we can improve upon for the final is finding a faster way to find the min distance in the codec. The reason we chose cosine-distance instead of euclidean distance is because most of the vectors in our codec are sparse, so words with little distinguishing features, i.e. closer to the origin, might be incorrectly calculated as “similar” by to euclidean distance.

Implementation of model



We built a two-layer neural network in Pytorch with a custom loss function to model our Spymaster. We use two linear layers with a RELU activation in between them. We then take the mean across the first dimension and squeeze to ensure an output of shape (batch_size,word2vec_size). This model was chosen for its small size and comprehensibility as we had to decide not only the model but also whether we would use supervised/unsupervised learning and how to construct a loss function that could accurately express what outputs should be favored.

Experimental results

We are successfully able to minimize the loss on subsequent trials, down to an arbitrarily small value. Currently, we limit the training to a 1000 iterations rather than let it converge. The output hint vector does have a small value but is often not a “good hint” when compared with the oracle (human evaluators), which indicates our model is not learning the features of a good hint either. However, given the loss of our output vector and the “nearest word” word2vec of our output vector have similar loss means that our output vector is mapping to words correctly. This points to our loss function being consistent and potentially a good starting point for future loss functions.

Sample Inference Run

Board: {"blue": ["scale", "nut", "bark", "tooth", "hole", "spot", "yard", "cold", "field"], "red": ["conductor", "contract", "shadow", "dinosaur", "lap", "ray", "key", "train", "mine", "tie", "fan", "bottle", "glove", "bow", "tablet"], "assassin": ["watch"]}

Clue: unaccompanied

Loss of output vector: 1.2477; Loss of nearest word to output vector (i.e., clue): 1.1905

Future Optimizations & Choices

One of our ideas going forward that will improve the quality of hints is to consider only a certain subset of the blue words in our loss function, rather than the average cosineDist between all the words. This is a more similar behavior to both our human oracles and our initial proposal--a spymaster that outputs (hint, # of related words) rather than (hint, 9) every time. A good arbitrary goal is to only try to match 2 or 3 words every time--which is about average for our oracles. A future loss function could look something like this, where we have arbitrarily selected k blue words to "match" with.

$$Loss(board,\ hint)\ =\ log(\frac{1}{m}\sum_{r\ in\ red\ words}^{m} cosineDist(r,\ hint))\ + log(\lambda * cosineDist(assassin,\ hint))$$

$$- log(\frac{1}{k}\sum_{b\ in\ k\ chosen\ blue\ words}^{k} cosineDist(b,\ hint))$$

Currently, our inference step takes about a minute to run as we find the min cosineDist of all words in our 3 million word codec. This is exceptionally slow, so we are still tackling ways to speed it up.

Another idea to improve our training would be to generate boards that include words from the entire Google News corpus, not just the 400 words in the Codenames game, to expand the range of word2vecs that our model interacts with.

Overall, we hope to increase the accuracy of our model and ultimately evaluate it by playing many games with human teammates and the AI spymaster.