CAPSTONE PROJECT

SEMANTIC SIMILARITY USING N-GRAMS AND NEURAL N-GRAMS

NGOC (MIKE) TRAN

UDACITY MACHINE LEARNING ENGINEER NANODEGREE PROGRAM

FEB 20TH, 2021

I. DEFINITION

PROJECT OVERVIEW

Semantic similarity between two words is a metric that measures the similarity between the meanings of the two. Historically, there are many ways to compute semantic similarity. According to Slimani (2013), it can be measured by ontologies, topologies, graph-based and statistical similarity. In our context, we are only interested in calculating semantic similarity based on statistical inference. Specifically, we will build a statistical language model and transform all the words into vectors. To compute the semantic similarity, we will compute the cosine distance between two vectors represented the two words. Cosine similarity ranges from [-1,1] where a 1 is interpreted as identical and -1 is different (Sidorov 2014).

Semantic similarity is one of the most important applications of a statistical language model. It does not only allow for finding interchangeable words in the same context, but also extends the research of natural language processing to find the similarity between larger mediums like paragraphs or texts. Another application of semantic similarity is plagiarism detector which we already did in the previous Udacity project. A good semantic similarity measurement depends on how good is the statistical language model. To date, there are many types of statistical language models: n-grams, decision tree models, linguistically motivated models, exponential models, and neural networks (Rosenfeld 2000).

PROBLEM STATEMENT

A statistical language model is used to model the joint probability distribution of words in a language. The n-grams model has been used widely and achieved many successes. In 2003, Bengio et al. proposed a new model that used a neural network as a base which is called a neural probabilistic language model. The authors claimed that the model is 24% better than a traditional n-gram. In this project, we will try to replicate the work of Bengio et al. (2003) by implementing both models: the classic n-grams model and the neural n-grams model to validate if later is actually better. We will also explore the neural model on the task of calculating semantic similarity between words in English.

METRICS

In natural language processing, the log perplexity is a metric to evaluate the statistical language model. We will use it as the evaluation metrics to compare models. Furthermore, we will calculate the cosine distance for 20 pairs of similar words pick randomly from the dictionary.

II. ANALYSIS

DATA EXPLORATION

The dataset we will use is the Brown Corpus (Francis 1979). The Brown corpus consists of 1,181,041 English words including punctuation. The uncommon words (which appeared less than 4 times) were merged to reduce the vocabulary size. After all, according to Bengio et al. (2003), there are 16,383 words in the dictionary.

The Brown Corpus has been updated several times since then. Therefore, our implementation resulted in a little bit different from the reports from Bengio. Specifically, there is a total of 1,161,192 words including punctuation. Our dictionary is at the size of 14,507.

Following Bengio et al, we divided the corpus into a training set of 800,000 words, a validation set of 200,000 and the rest is for testing.

EXPLORATORY VISUALIZATION

The table below summarizes the statistics of our dataset and compares it with the one reported by Bengio et al.

TABLE 1: DATA SET STATISTICS

The Brown Corpus		
	Bengio et al (2003)	Our implementation
Data Size (words)	1,181,041	1,161,192
Unique Words	47,578	45,387
Vocabulary Size (Frequency >4)	16,383	14,507

ALGORITHMS AND TECHNIQUES

Several algorithms will be used in this project: vectorization, n-gram, softmax, stochastic gradient descent, and perplexity.

Counting grams is a standard technique that we also previously completed in the previous project (Plagiarism Detector). In this project, we will implement a more advanced version which is the interpolated trigram model. An interpolated trigram model combines

and smooths out the result of unigram, bigram, and trigram with some weights (Goodman, 2001). There could be some more fine-tuning to the weights but usually, the trigram should have the most weights.

For the neural model network, we need to first vectorize all the words from our dataset. Bengio reported using three variants of the embedding size: 30, 60, and 100. In our implementation, we use an embedding size of 20 which fits more for our hardware allowance. Again, setting the embedding size is the "curse of dimensionality" which we could try and tune with our evaluation set.

A softmax function is needed for the forward pass. It will guarantee that the output layer will result in a normalized probability distribution. In our project, we will use the default LogSoftmax from PyTorch.

EQUATION 1: SOFTMAX FUNCTION

$$\text{LogSoftmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

Stochastic gradient descent (SGD) is the basis of all neural networks. It will first randomly set the values of the input, then "step" over them to generate an output and calculate the loss. Knowing the loss, it will do the derivative to calculate how much the input needs to change to minimize the loss. We use a PyTorch optimizer SGD with a learning rate of 0.1 and a weight decay of 0.00001 which are much bigger than the one in the paper: 0.001 and 10^{-8} respectively.

The perplexity is our metric for this project. The log perplexity of a language model indicates how well it predicts a word given the previous words. We want to minimize the perplexity to maximize the probability to predict the new word. It is calculated as below where $P(w_1,\ldots,w_n)$ is the joint probability of words.

EQUATION 2: PERPLEXITY

$$e^{-\frac{\log P(w_1,\ldots,w_n)}{n}}$$

BENCHMARK

We have the benchmark result from Belgio et al (2003). The authors have reported all the perplexities of each model with various versions of different benchmarks and parameters. We can use those results as a benchmark for our project.

III. METHODOLOGY

DATA PREPROCESSING

Fortunately, we can get the Brown data from the '*nltk*' packet directly. Following Belgio et al (2003), we did little to no data preprocessing. After downloading the data set from the '*nltk*' packet, we built a vocabulary for the data set and indexed all the words. When building the vocabulary, we merged any words that have a frequency less than 4 into one group of 'unk' (unknown words). Indexing words are straight-forward: we iterated through the dictionary and assigned the more common words as a smaller index, starting from 0. There are several NLP techniques to clean the data, for example: removing stop words and stemming words. Since we wanted to stay true to Belgio's experiment, we also did not perform any advanced data preprocessing. Note that data cleaning can make bias and/or give advantages/disadvantages to certain models. In my opinion, performing those NLP tasks will reduce the vocabulary size, which will favor the n-grams model since the neural network is more famous for handling big vocabulary size.

IMPLEMENTATION

We will implement the two models based on the setup described in the paper. Implementing the traditional n-grams model will be straightforward. We already did an n-grams implementation for the previous project: Plagiarism Detector. However, we would need to modify that version to fit with the n-grams model used in the paper which is interpolated trigram model. An interpolated trigram model combines the weighted results of unigram, bigram, and trigram together. The n-grams are calculated by counting them in the training set. To combine them, we multiply each result and its respective weight

following the formula, where α_1 is the weight of the unigram, p_1 is the unigram count for word w, α_2 is the weight of the bigram, p_2 is the count of the word w given word w_{t-1}, and α_3 is the weight of the trigram, p_3 is the count of the word w given word w_{t-1} and w_{t-2}.

EQUATION 3: INTERPOLATED N-GRAMS

$$\hat{P}(w_t|w_{t-1},w_{t-2}) = \alpha_1(q_t)p_1(w_t) + \alpha_2(q_t)p_2(w_t|w_{t-1}) + \alpha_3(q_t)p_3(w_t|w_{t-1},w_{t-2})$$

For our implementation, we want to output the interpolated trigram as log probability, which will enable us to do the next step more easily: calculating the perplexity of the n-grams model. We simply walked over the training set and calculated the interpolated n-grams log probability for each sequence of words. Then, we summed them over and average the result by the length of vocabulary, following equation (2). Here, we did some modifications to improve the n-gram model. Noticing that p_i already contain p_{i-1}, we subtracted p_i with p_{i-1} to reduce the redundant count.

The neural n-grams model has 3 layers: an input layer which is a word-embedding layer, a hidden layer with a tanh function, and an output layer with a softmax function. First, we take the words, represent them as embedded vectors then concatenate them. Then, we will feed forward through the tanh function and change the dimension to match the vocabulary' size. Lastly, we will use the softmax function to have the probability distribution of words. With that distribution, we could calculate the log perplexity which we try to minimize as the loss function. In a reversed manner, we did a backward pass to update the gradient and the weights. Fortunately, the process is simplified with PyTorch by calling the *backward* function on our loss and stepping the optimizer with *step* function.

After training the model, we can explore words' semantic similarity by comparing how close they are in the vector space. We choose the famous cosine similarity which is the cosine of the angle between two vectors. This value will range between -1 (the least similar/the negation of the vector) and 1 (the most similar/the same). This can be easily calculated by dividing the dot product of two vectors by the multiplications of their lengths. The equation is shown below:

EQUATION 4: COSINE SIMILARITY

$$cos(\theta) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}$$

REFINEMENT

For the n-grams model, there is not much room for improvements except for the interpolating function. Training a new maximum likelihood estimator (MLE) to tune the weights of the n-grams is out of scope for this project.

For the neural network, we tuned two hyperparameters: the number of hidden layers (h), and the embedding size (m). Increasing both of them increased the training time greatly. The best neural model from Belgio et al is the one with h = 100 and m = 30. Since we are at a much smaller scale, we tried the model with h = (30, 50) and m = (20, 30). Our best model is running at h = 50 and m = 20 which gives us the best compromise in terms of performance and quality. We also tried a different number of epochs to see the difference. The benchmark model converged at 50 epochs. Due to hardware constraints, we stopped our training at 20 epochs. After that, it overfits the training result and the loss reduces insignificantly after each epoch.

IV. RESULTS

MODEL EVALUATION AND VALIDATION

Our implementation results in the following table:

TABLE 2: OUR RESULTS

Model	Hidden Layers (h)	Embedding Size (m)	Epochs	Perplexity on Test Set
N-grams	-	-	-	4.875
Neural Network	30	20	20	5.764
	50	20	20	5.697

	30	30	20	5.802
	50	30	20	5.559

To comparing to Belgio et al (2003), we have the following table. As can be seen, our n-grams model beats the benchmark model by 15.11% while the neural networks are similar.

TABLE 3: COMPARISON RESULTS

Perplexity	Our Implementation	Belgio et al
N-grams	4.875	5.743
Neural Network	5.559	5.529

JUSTIFICATION

Our traditional n-grams are better by 15.11% than the benchmark model. There could be multiple explanations for this. The difference in the implementation could lead to different results. Our modification in the n-grams model is showing its effectiveness. By removing the redundant count, the perplexity of the interpolated n-grams model is calculated more precisely. Lastly, comparing with Bengio et al, our neural network model is equivalent. That equivalent in the more complicated model is excellent proof that our implementation is correct.

SEMANTIC SIMILARITY

We explored the model to find the semantic similarity between several “similar” words. Some interesting results are:

Word 1	Word 2	Cosine Distance
Dog	Car	-0.1478
	Cat	-0.0098
Spear	Rock	-0.2203
	Sword	-0.1892

Atlanta	City	-0.0866
	Place	0.0337
Inadequate	Ambiguous	-0.2360
	Experienced	-0.1257
Husband	Son	0.5580
Wife	Daughter	0.0678

The first example is classic but unfortunately, our model doesn't show much of a difference between Dog vs. Car and Dog vs. Cat. The second example is from Roland et aI. According to our model, "spear" is a little bit more similar to "sword" than to "rock" but not significantly. It's interesting how "place" and "Atlanta" are more similar than "city" and "Atlanta" - negative result for City-Atlanta and positive for Place-Atlanta. Atlanta is, of course, a place, but it is also a city, and since "city" is a sub-category of "place". Commonly, Atlanta would be associated with 'city' more than 'place'. However, the result is not that much different. "Inadequate" is more similar to "experience" than to "ambiguous", which is a bit strange if we think of the first and the third adjectives as having a negative connotation while the second as a positive adj. The last one is fascinating. "Husband" to "son" is a positive high value (0.5580) while "wife" to "daughter" is pretty low at (0.0678). Thinking about it, in the writing context, we usually use "Husband" and "Son" together more than "wife" and "daughter".

V. CONCLUSION

In this project, we have explored and researched the two famous models: the n-grams model and the neural network model for the task of building a language model to find semantic similarity between words. We have successfully implemented both models. Thanks to our improvement in removing the redundant count in the interpolated n-grams, our n-grams model is better by 15.11% than the benchmark model. Meanwhile, our neural network model is equivalent to the benchmark one reported by Belgio et al. After implementing them, we did some more exploration by calculating the semantic similarities

between some “similar” English words. The results are interesting and not always the same as human judgments.

I found it particularly interesting playing with different hyperparameters for the neural model. They all have different impacts on results. For example, more epochs result in more overfitting while the loss in the evaluation set is not reducing significantly. I also found out that it doesn’t make sense to have the number of features (dimension size) bigger than the number of hidden layers. Also if the number of hidden layers is too big, the model will be overfitting. On the other hand, I encounter some difficulties in the project that made it challenging. Some calculations are not very straightforward even I read the source paper very carefully. Past Udacity projects, Udacity’s forum, and PyTorch manual are really helpful for this project.

FUTURE WORKS

The project has several rooms for improvement. For the traditional n-grams model, one could train a separate maximum likelihood estimator to find the weights for the interpolated model. We just use some probabilities that seem to make sense for us: giving the highest weight for the trigrams, less weight for the bigrams, and least weight for the unigram. For the neural network model, we could simply train for more epochs and increase the number of features/number of hidden layers. Our best result still uses a lower number than the benchmark’s ones. For the semantic similarity, we could build a list of the most similar words which could allow us for more dynamic exploring. Also, since all the words are vectors, we could reduce the dimension and plot a graph of some common words to see if they are close with each other on the reduced vector dimension.

REFERENCES

Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A Neural Probabilistic Language Model. *Journal of Machine Learning Research 3,* 1137-1155.

Francis, N., & Kucera, H. (1979). *Brown Corpus Manual*. Brown Corpus Manual. http://korpus.uib.no/icame/manuals/BROWN/INDEX.HTM#bc2

Sidorov, G., Gelbukh, A., Gómez-Adorno, H., Pinto, D. (2014). Soft Similarity and Soft Cosine Measure: Similarity of Features in Vector Space Model. *Computación y Sistemas*. 18 (3): 491–504. doi:10.13053/CyS-18-3-2043.

Slimani, T. (2013). Description and Evaluation of Semantic Similarity Measures Approaches. *International Journal of Computer Applications*. Vol 80. 25-33. 10.5120/13897-1851.

PyTorch documentation — PyTorch 1.7.1 documentation. (2019). PyTorch. https://pytorch.org/docs/1.7.1/index.html

Roland, Douglas & Yun, Hongoak & Koenig, Jean-Pierre & Mauner, Gail. (2011). Semantic similarity, predictability, and models of sentence processing. *Cognition*. 122. 267-79. 10.1016/j.cognition.2011.11.011.

Rosenfeld, R. (2000). Two decades of statistical language modeling: where do we go from here? *in Proceedings of the IEEE*, vol. 88, no. 8, pp. 1270-1278, Aug. 2000, DOI: 10.1109/5.880083.