CS5740: Assignment 2
https://github.com/cornell-cs5740-20sp/assignment-2-nlpeace

Jack Rossi
jdr342

Kushal Singh
ks2377

1 Introduction (5pt)

Word embeddings are an approach to featurizing text that seeks to embed a sparse word space into a dense vector. In this paper we seek to derive these vectors for a selection of relevant words. We will use a neural network approximating the skip-gram model with negative sampling to derive these vectors, and evaluate our embeddings against human-derived similarity rankings.

We experimented with corpus sizes, optimization algorithms, objective functions and many other model hyperparameters. Ultimately, using batch sizes of 50,000 words, window-based contexts, the an extended corpus of 3 million sentences, and binary cross entropy loss performed best, providing development correlation of 0.48, and test correlation of 0.160.

2 Model (10pt)

Formally, this model uses embeddings to calculate the probability that a given word, context (w, c) pair is found in the training corpus. We used window-based contexts. Thus, for a sentence consisting of words $x_1, ..., x_n$, we defined for each x_i the window around it, $x_{i-d}, ..., x_i, ..., x_{i+d}$ where the window size is $2d + 1$. We seek to find the embeddings, θ, such that the following objective is maximized.

$$argmax_\theta \prod p(D = 1|w, c) \prod p(D = 0|w, c) \quad (1)$$

Where D is an indicator of whether or not the pair was found in the corpus. In practice, this model computes the dot product of embeddings of pairs, subsequently using the sigmoid function to scale the output to probabilities. The embeddings are then updated by backpropagating errors. We tried two loss functions, the first of which is Binary Cross Entropy, given in equation 2.

$$BCE = -(ylog(f(x)) + (1-y)log(1-f(x))) \quad (2)$$

Where x is a specific (w, c) pair, $f(x)$ is the network output for example x, and y is class identity of example x, $y\epsilon\{0, 1\}$. Another loss function we tried was Mean Squared Error, given in equation 3.

$$MSE = \frac{1}{n} sqrt(\sum_{i=1}^{n} (y_i - f(x_i))^2) \quad (3)$$

Where x, $f(x)$, and y are given by the previous equation 2, and n is the number of training examples.

3 Experiments (12pt)

In experiment 1 we explored basic choices in the structure of the neural network, including the dimensionality of the word embeddings, the loss function used, and the size of the data used for training. We tried two loss functions as described in the previous section, two corpus sizes (1000, 10,000), and three embedding dimensionalities (10, 50, 100). These options were permuted together to give twelve unique results. In experiment 2 we tested additional hyperparameters of the model and training process, including the optimization algorithm used, the learning rate, and the number of epochs for which the neural network was trained. We used two optimization algorithms, Stochastic Gradient Descent (SGD), and the Adam adaptive learning rate algorithm. We supplied these algorithms with two potential learning rates, 0.1 and 0.01, and we trained the models for either 2, 5, or 7 epochs. As before, these three factors were combined to produce twelve unique results.

4 Data

Data (5pt) Initially, we used sentences from the training-data.1m file provided by statmt.org to formulate our contexts, which we then passed into our neural network. To improve our performance on the test set, we decided to also use sentences from the 1 billion word language modeling benchmark, also provided by statmt.org. This gave us a lot more contexts to train our network on, which yielded better performance in the test set. We computed our total vocabulary as the sum of all the words that appeared more than once among the 1,779,691 sentences.

Table 1: Data statistics

Metric	Value
Vocab Size	115,338
Number of Sentences (1m file)	897,691
Number of Sentences (1b file)	882,000
Total Number of Sentences	1,779,691
Positive Contexts	90,375,336
Negative Contexts	42,318,878
Total Contexts	132,694,214
Number of Development Examples	353
Number of Test Examples	2034

Pre-processing and Handling Unknowns (10pt) We preprocessed the data by loading sentences into a dataframe and then processing each sentence in turn. We lowercased all words and removed words that were not strictly alphanumeric characters. We also removed stop words (via nltk.corpus stopwords API). We did not conduct lemmatization because we felt that various word forms should maintain distinct embeddings. We generated word, context pairs according to the definition in the Model section, and we generated negative samples by pairing target words with contexts randomly sampled from the corpus. For windows looking d units to the left and right of each target word, we generated 2d+1 negative samples for each positive sample. Initially we trained embeddings for all the words in the training set, and assigned to new words the average of all the other embeddings. After observing poor performance of this method, we instead implemented a procedure of trimming rare words and replacing them with a RARE catch-all. For each word viewed once or less in the training corpus, we replaced those tokens with a special RARE token. This RARE token had it's embedding learned as any other word. Then, when new words were observed in the development and test sets, they were assigned the embedding of this RARE token. We also used nltk's wordnet functionality to find synonyms and, consequently, tag parts of speech on these synonyms for words that did not appear in our training set. We then appended the different parts of speech ('N' for noun, 'A' for adjective, 'V' for verb, 'R' for adverb) to our word-to-index dictionary. So, all nouns pointed to a particular index, all adjectives pointed to another index, as such. This remarkably improved our performance because our contexts (which are underpinned by these indices) now contain information about parts of speech, which can scale a lot easier to rare and unseen words. Specifically, as opposed to placing all rare and unseen words into a catch-all RARE bucket, we can now extract more granular information about these words (parts of speech), which improves the performance of the neural network.

5 Implementation Details (3pt)

We implemented the model using PyTorch, and experimented with many hyperparameters during the development process. The batch size, learning rate, optimization algorithm, epochs, and loss functions were decided through experimentation.

6 Results and Analysis

Test Results (3pt) We evaluated three alternative models, whose resulting performance is summarized in Table 1. The first model used 5000 training sentences and 30-dimensional embeddings, but showed very low performance on the test set. As we experimented on the development data, we found that better handling of rare words, and larger training corpora led to better performance. Ultimately the model with 50-dimensional embeddings, enhanced handling of rare words, and 3 milllion training sentences produced our best results, test correlation of 0.16077.

Table 2: Correlation scores on the test data by the size of the dataset and the number of dimensions in the embeddings.

data	**dim**	**Corr**
5K	30	0.021
100K	100	0.118
3M	50	0.16077

Development Results (15pt) In our first experiment, we looked at the interrelationship between loss functions, dimensionality of the embeddings, and training data size and these choices' impact on development correlation. Table 1 summarizes our findings; one, BCE loss performed better than MSE loss. Another straightforward finding is that larger training corpora produced better results. The impact of embedding dimensionality was less straightforward. With 1000 training samples, higher-dimensional embeddings fared better. However, this trend was reverse with more data. As our final model used much more data than any of our experiments, we used an intermediate value for embedding dimensionality.

Table 3: Correlation performance from experiment 1

MSE Loss	**NumSentence**	
dimEmbeds	1000	10000
10	0.0003	0.121
50	0.025	0.102
100	0.044	0.096
BCE Loss	**NumSentence**	
dimEmbeds	1000	10000
10	0.035	0.138
50	0.053	0.091
100	0.068	0.047

Next, we looked at additional design decisions in the neural network modelling and training process: optimizer choice, learning rate, and number of epochs. We channeled the learnings from the previous experiment and used 50 dimensional embeddings, 10000 training sentences, and BCE Loss. Table 5 shows the results. In general, the Adam optimizer proved more effective than SGD, though the latter was suprisingly performant with a high learning rate and few epochs. Typically, more epochs provided better performance, but the Adam optimizer clearly worked better with a learning rate of 0.01.

Qualitative Analysis (10pt) To assess causes of error in our model, we rank-ordered the development examples by the difference in their predicted and actual similarity. We reviewed the top 50 examples, and assigned groups iteratively, combining and removing groups until we settled on the summary shown in Table 4. We hypothesize that our model can be improved by including more deliberate handling of synonymy, especially when one synonym is a rare word, as in "dollar" and "buck". We also think that better performance would results from handling proper nouns through our rare word approach. Some errors are more mysterious, and these examples happened to be the second most frequent in our selected set.

Table 4: Correlation performance from experiment 2

Adam	**Learn. Rate**	
Epochs	0.01	0.1
2	0.1217	0.0367
5	0.1006	0.0932
7	0.1664	0.0256
SGD	**Learn. Rate**	
Epochs	0.01	0.1
2	0.0713	0.1505
5	0.0299	0.0743
7	0.0756	0.0364

Table 5: Error analysis.

Group	Count	Example
Synonym	21	gem, jewel
Other	10	chord, smile
Science	9	planet, star
Proper Noun	7	Arafat, terror

7 Conclusion (3pt)

Our best results on the development set yielded a correlation coefficient = 0.487. Our bet results on the test set yielded a similarity coefficient = 0.168. For further experimentation, we would have definitely liked to use more sentences to generate our positive and negative contexts. Due to limited compute time and power, we were only able to generate contexts for approximately 1.8 million sentences. However, we noticed that this immediately yielded an improvement in performance on our development and test sets.