CS565 - Intelligent Systems and Interfaces - Assignment 2

Lavish Gulati - 170101082

Link to Google Colab Notebook: Click here

2.1 - N-gram language model

0 - Data Preprocessing and Splitting

1. First, we tokenize the corpus into sentences using the nltk sent_tokenize which uses the PunktSentenceTokenizer. It uses an unsupervised algorithm to build a model for abbreviation words, collocations, and words that start sentences; and then uses that model to find sentence boundaries.
2. Then, we tokenize each sentence into words using the nltk word_tokenize which uses an improved TreebankWordTokenizer. The Treebank tokenizer uses regular expressions to tokenize text.
3. After sentence segmentation and word tokenization, we randomly shuffle the data using random.shuffle.
4. We split the data into a training set and fixed test set with 90% and 10% sentences respectively.
5. **Number of sentences in corpus: 761582**
6. **Number of words in corpus: 19602594**

---

1 - Trigram Language Model

Smoothing

1. There are problems of balance weight between infrequent grams (for example, if a proper name appears in the training data) and frequent grams. Also, items not seen in the training data will be given a probability of 0.0 without smoothing.
2. In practice, it is necessary to smooth the probability distributions by also assigning non-zero probabilities to unseen words or n-grams. The reason is that models derived directly from the n-gram frequency counts have severe problems when confronted with any n-grams that have not explicitly been seen before – the zero-frequency problem.
3. Some of these methods are equivalent to assigning a prior distribution to the probabilities of the n-grams and using Bayesian inference to compute the resulting posterior n-gram probabilities.
4. It is, therefore, necessary for the model to adjust the probability estimates for trigrams which are under-represented in the training set. This process is called smoothing.

Discounting

➔ Consider a trigram (u, v, w) and define c*(u, v, w) to be the discounted count of the trigram, given by

$$c^*(u, v, w) = c(u, v, w) - \beta$$

where β is again the discounting value. Then the trigram model is

$$q_D(w|u,v) = \begin{cases} \frac{c^*(u,v,w)}{c(u,v)} & \text{If } w \in \mathcal{A}(u,v) \\ \alpha(u,v) \times \frac{q_D(w|v)}{\sum_{w \in \mathcal{B}(u,v)} q_D(w|v)} & \text{If } w \in \mathcal{B}(u,v) \end{cases}$$

where α(u, v) is the "missing" probability mass.

$$\alpha(u,v) = 1 - \sum_{w \in \mathcal{A}(u,v)} \frac{c^*(u,v,w)}{c(u,v)}$$

- ➔ This reflects the intuition that if we take counts from the training corpus, we will systematically overestimate the probability of trigrams seen in the corpus (and under-estimate trigrams not seen in the corpus).
- ➔ The intuition behind discounted methods is to divide this "missing mass" between the words w such that c(v, w) = 0.
- ➔ **To calculate the most optimal value of β**, we run the log-likelihood test and vary **β** from 0 to 1 and choose that **β** that maximizes the log-likelihood value.

Linear Interpolation

- ➔ In this method, the probability is estimated from a linear combination of trigram, bigram, and unigram relative frequencies.

$$q(w|u,v) = \lambda_1 \times q_{ML}(w|u,v) + \lambda_2 \times q_{ML}(w|v) + \lambda_3 \times q_{ML}(w)$$

$$\lambda_1 \geq 0, \lambda_2 \geq 0, \lambda_3 \geq 0$$

$$\lambda_1 + \lambda_2 + \lambda_3 = 1$$

- ➔ The unigram estimate will never have the problem of its numerator or denominator being equal to 0: thus the estimate will always be well-defined, and will always be greater than 0. However, the unigram estimate completely ignores the context (previous two words).
- ➔ In contrast, the trigram estimate does make use of context but has the problem of many of its counts being 0. The bigram estimate falls between these two extremes.
- ➔ The method has three smoothing parameters, λ_1, λ_2, and λ_3. The three parameters can be interpreted as an indication of the confidence, or weight, placed on each of the trigram, bigram, and unigram estimates. For example, if λ_1 is close to 1, this implies that we put significant weight on the trigram estimate; conversely, if λ_1 is close to zero we have placed a low weighting on the trigram estimate.
- ➔ **We calculate the value of λ** by repeatedly calculating the new λ using the expected count from the function defined above and continuing this process till the value of all the new λ lies within a specific range of λ by ε.

2 - Splitting training dataset

We split the original training set after shuffling into 5 sets of training and validation sets each with 90% and 10% sentences of the original training set.

3 - Performance on Validation Set

Discounting performance on validation set

```
Set 1 - beta: 0.61 , perplexity: 385.98464393175266
Set 2 - beta: 0.61 , perplexity: 335.20801152217865
Set 3 - beta: 0.66 , perplexity: 362.5238493150579
Set 4 - beta: 0.64 , perplexity: 378.2673871942632
Set 5 - beta: 0.62 , perplexity: 346.0446900001597
```

Interpolation performance on validation set

```
Set 1 - lambda: [0.32771195, 0.396509469, 0.27577857] , perplexity: 91.2276931627
Set 2 - lambda: [0.32953357, 0.39630213, 0.27416428] , perplexity: 92.4252160136
Set 3 - lambda: [0.327929299, 0.3969918, 0.27507882] , perplexity: 91.7769515582
Set 4 - lambda: [0.33073740, 0.396440976, 0.27282161] , perplexity: 93.0446877962
Set 5 - lambda: [0.32940723, 0.39648050, 0.274112265] , perplexity: 92.347352795
```

4 - Performance on Test Set

Discounting performance on test set

```
Set 1 - beta: 0.65 , perplexity: 384.1294669832734
Set 2 - beta: 0.61 , perplexity: 379.87093784436684
Set 3 - beta: 0.63 , perplexity: 383.77810953043127
Set 4 - beta: 0.64 , perplexity: 382.5792679308408
Set 5 - beta: 0.64 , perplexity: 382.0304207051159
Variance in perplexity: 2.8562457111186608
```

Interpolation performance on test set

```
Interpolation performance on test set
Set 1 - lambda: [0.32968333, 0.396521102, 0.273795557] , perplexity: 91.9973987971
Set 2 - lambda: [0.32950444, 0.396588431, 0.27390712] , perplexity: 91.7826863838
Set 3 - lambda: [0.32960189, 0.396502825, 0.27389527] , perplexity: 91.8785094042
Set 4 - lambda: [0.32961886, 0.39635289, 0.274028246] , perplexity: 91.9142464951
Set 5 - lambda: [0.32949078, 0.39650747, 0.274001734] , perplexity: 91.824637033
Variance in perplexity: 0.006872811974727759
```

Laplace performance on test set

```
Set 1 - perplexity: 589.2159070170205
Set 2 - perplexity: 589.9118492378913
Set 3 - perplexity: 588.7354057333338
Set 4 - perplexity: 591.6856934734319
Set 5 - perplexity: 589.5165635032213
Variance in perplexity: 1.2805890031643805
```

---

5 - Summary

The results are summarized in the following table:

	Average Perplexity	Variance in Perplexity
Discounting	382.47764	2.85625
Interpolation	91.82464	0.00687
Laplace	589.81308	1.28059

We observe that the perplexity value is in the order: Laplace > Discounting > Interpolation. This means that Interpolation achieves better performance than Discounting while Laplace has the worst performance.
For variance in perplexity, the order is: Interpolation < Laplace < Discounting. Only Interpolation has variance almost equal to zero, while variance for Discounting and Laplace is quite high.

2.2 - Vector Semantics: GloVE implementation

1 - GloVe embedding method implementation

1. First, we build vocabulary given the corpus which returns the frequency and word index of each word in the corpus. We use nltk sent_tokenize for sentence segmentation and nltk word_tokenize for word tokenization same as in 2.1.
2. Then, we build the co-occurrence matrix which is actually represented by a list containing tuples (centre word ID: i, context word ID: j, cooccurrence: Xij) to optimize usage of memory. We iterate over all context words in a window of each centre word and increment each cooccurrence Xij by 1/d where d is the distance between the centre word and the context word.
3. After we get the cooccurrences, we train the GloVe weight vector W (of size 2V x D where V is vocabulary size and D is the dimension of vectors) using adaptive gradient descent (AdaGrad).
4. AdaGrad is an algorithm for gradient-based optimization that does the following: It adapts the learning rate to the parameters, performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features. For this reason, it is well-suited for dealing with sparse data.
5. Finally, we obtain the GloVe word embeddings by adding the centre word vector and context word vector for each word.

Optimization method: Adaptive Gradient Descent (AdaGrad)

Vocabulary size: 76825

Parameters used:

Parameter	Value	Explanation
CONTEXT_WINDOW	10	Size of the window taken during the evaluation of co-occurrence matrix
DIM_SIZE	100	Vector dimensions of the word embeddings
ITERATIONS	50	Number of iterations to run AdaGrad to train the word vectors
LEARNING_RATE	0.05	The initial learning rate for AdaGrad optimization
X_MAX	100	Value of x_{max} used in the weighting function $f(X_{ij})$
ALPHA	0.75	Value of α used in the weighting function $f(X_{ij})$
SMALL_RATIO	0.1	Fraction subset of the corpus taken. Here, 10% of the whole corpus was taken due to memory issues.

---

2 - AdaGrad on GloVe training

1. Once we've prepared the co-occurrence matrix X, the task is to decide vector values in continuous space for each word we observe in the corpus. We produce vectors with a soft constraint that for each word pair of word i and word j,

$$\vec{w}_i^T \vec{w}_j + b_i + b_j = \log X_{ij}$$

where b_i and b_j are scalar bias terms associated with words i and j, respectively.

2. We minimize the objective function J, which evaluates the sum of all squared errors based on the above equation, weighted with a function f:

$$J = \sum_{i=1}^{V} \sum_{j=1}^{V} f\left(X_{ij}\right) \left(\vec{w}_i^T \vec{w}_j + b_i + b_j - \log X_{ij}\right)^2$$

3. We choose an f that helps prevents common word pairs (i.e., those with large X_{ij} values) from skewing our objective too much:

$$f\left(X_{ij}\right) = \begin{cases} \left(\frac{X_{ij}}{x_{\max}}\right)^{\alpha} & \text{if } X_{ij} < x_{\max} \\ 1 & \text{otherwise.} \end{cases}$$

4. From our original cost function J, we derive gradients with respect to the relevant parameters $\vec{w}_i$, $\vec{w}_j$, b_i, and b_j. Note that $f(X_{ij})$ doesn't depend on any of these parameters. Below we use the operator $\odot$ to denote element wise vector multiplication.

$$\nabla_{\vec{w}_i} J = \sum_{j=1}^{V} f\left(X_{ij}\right) \vec{w}_j \odot \left(\vec{w}_i^T \vec{w}_j + b_i + b_j - \log X_{ij}\right)$$

$$\frac{\partial J}{\partial b_i} = \sum_{j=1}^{V} f\left(X_{ij}\right) \left(\vec{w}_i^T \vec{w}_j + b_i + b_j - \log X_{ij}\right)$$

Note that we ignore the factor of 2 because it is adjusted in the learning parameter.

5. Let us consider each parameter as θ_i, so $\vec{w}_i \Rightarrow \theta_1$, $\vec{w}_j \Rightarrow \theta_2$, $b_i \Rightarrow \theta_3$ and $b_j \Rightarrow \theta_4$. As Adagrad uses a different learning rate for every parameter θ_i at every time step t, we first show Adagrad's per-parameter update, which we then vectorize. For brevity, we use g_t to denote the gradient at time step t. $g_{t,i}$ is then the partial derivative of the objective function w.r.t. to the parameter θ_i at time step t:

$$g_{t,i} = \nabla_\theta J(\theta_{t,i})$$

6. In its update rule, Adagrad modifies the general learning rate η at each time step t for every parameter θ_i based on the past gradients that have been computed for θ_i:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

where $G_t \in R^{d \times d}$ is a diagonal matrix where each diagonal element i,i is the sum of the squares of the gradients w.r.t. θ_i up to time step t, while ε is a smoothing term that avoids division by zero (usually on the order of 10^{-8}).

7. As G_t contains the sum of the squares of the past gradients w.r.t. to all parameters θ along its diagonal, we can now vectorize our implementation by performing a matrix-vector product $\odot$ between G_t and g_t:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

---

3 - Word Similarity Comparison

Benchmark Tool

Word Embeddings Benchmark tool is used for word-similarity comparison between the pre-trained GloVe embeddings and embeddings obtained by the implementation. The Word Embedding Benchmark package is focused on providing methods for easy evaluating and reporting results on common benchmarks (analogy, similarity and categorization). The tool includes support for 18 popular word-similarity benchmark datasets, some of which are used for the comparison:

- ➔ **MEN:** The MEN Test Collection contains 3,000 English word pairs (one for training and one for testing) together with human-assigned similarity judgments. The word pairs are randomly selected from words that occur in the freely available ukWaC and Wackypedia corpora.
- ➔ **WS353:** WordSim353 is a test collection for measuring word similarity or relatedness, developed and maintained by E. Gabrilovich. The collection contains a split of the test set into two subsets, one for evaluating similarity, and the other for evaluating relatedness.
- ➔ **SimLex999:** SimLex-999 is a gold standard resource for the evaluation of models that learn the meaning of words and concepts. SimLex-999 provides a way of measuring how well models capture similarity, rather than relatedness or association.
- ➔ **MTurk:** The Amazon Mechanical Turk data [MTurk-287] consists of 287 word pairs, elaborated with the collaboration of Amazon's Mechanical Turk workers.
- ➔ **RW:** The Stanford Rare Word (RW) Similarity dataset has been regarded as a standard evaluation benchmark for rare word representation techniques. The dataset has 2034 word pairs which are selected in a way to reflect words with low occurrence frequency in Wikipedia, rated with a similarity scale [0,10].

The sample data for each dataset is illustrated below:

Benchmark Dataset	Word 1	Word 2	Human assigned score
MEN	sun	sunlight	10
WS353	love	sex	6.77
SIMLEX999	old	new	1.58
MTurk	episcopal	russia	5.5
RW	squishing	squirt	5.88

Spearman's correlation

We use Spearman's correlation to measure the word similarity between the two word embedding datasets. Spearman's correlation is what is known as a non-parametric statistic, which is a statistic whose distribution does not depend on parameters. Very often, non-parametric statistics rank the data instead of taking the original values.

To calculate Spearman's correlation we first need to map each of our data to ranked data values:

$$\mathbf{x} \rightarrow \mathbf{x}^r$$
$$\mathbf{y} \rightarrow \mathbf{y}^r$$

If the raw data are [0, -5, 4, 7], the ranked values will be [2, 1, 3, 4]. We can calculate Spearman's correlation in the following way:

$$SCORR(\mathbf{x}, \mathbf{y}) = \frac{\sum_{i=1}^{n}(x_i^r - \bar{x}^r)(y_i^r - \bar{y}^r)}{\sqrt{\sum_{i=1}^{n}(x_i^r - \bar{x}^r)^2}\sqrt{\sum_{i=1}^{n}(y_i^r - \bar{y}^r)^2}}$$

where

$$\bar{x}^r = \frac{1}{n}\sum_{i=1}^{n} x_i^r$$

Spearman's correlation benchmarks monotonic relationships, therefore it can have perfect relationships that are not linear. It can take a range of values from -1 to +1.

Interpretation: If Y tends to increase when X increases, the Spearman correlation coefficient is positive. If Y tends to decrease when X increases, the Spearman correlation coefficient is negative. A Spearman correlation of zero indicates that there is no tendency for Y to either increase or decrease when X increases. The Spearman correlation increases in magnitude as X and Y become closer to being perfectly monotone functions of each other. When X and Y are perfectly monotonically related, the Spearman correlation coefficient becomes 1.

Word similarity comparison

For missing words in the datasets, we take the mean vector to denote the embedding of the missing word.

Benchmark Dataset	Spearman’s correlation on implemented GloVe word embeddings	Spearman’s correlation on pre-trained GloVe word embeddings
MEN	0.12954	0.57734
WS353	0.20867	0.46979
SIMLEX999	0.07121	0.12221
MTurk	0.18482	0.56410
RW	0.20551	0.23074

We train the GloVe word embeddings on a vocabulary size of 76,825 words whereas the original GloVe word embeddings are trained on a vocabulary size of 400,000 which is way larger. So, we do not get the exact results as shown by the Spearman’s correlation on pre-trained embeddings. But the trend of the results is similar and is summarized as follows:

1. We do not get any negative value of the Spearman’s correlation on any benchmark dataset denoting that there is a positive correlation (or the word vectors are monotonically related) to each other as also illustrated by the pre-trained GloVe embeddings.
2. The correlation values are almost half to that of the correlation values of the pre-trained embeddings. Hence, we have built a close word-similarity relationship to that of the pre-trained GloVe word vectors. The trend also shows that if we use larger data, we can increase the correlation values to match with that of the pre-trained GloVe.