CS-565 | INTELLIGENT SYSTEMS AND INTERFACES
ASSIGNMENT-2
VAKUL GUPTA | 170101076

Link to Google Colab Notebook - **Click Here**

NOTE - Entire processing has been performed on full corpus for N-gram Language Model while 10% of corpus has been taken for Vector Semantics - GloVe implementation (due to RAM issues)

TASK 1 - N-gram Language Model

1.0 - Preprocessing - Sentence Segmentation, Word Tokenization and Splitting

Sentence Segmentation

- Sentence segmentation of the corpus is done using the **nltk.sent_tokenize** which uses the PunktSentenceTokenizer. It uses an unsupervised algorithm to build a model for collocations, and words that start sentences; and then uses that model to find sentence boundaries.
- **Number of Sentences** - `761582`

Word Tokenization

- Word tokenization of the sentences is done using the **nltk.word_tokenize** which uses an improved TreebankWordTokenizer which uses regular expressions to tokenize text.
- **Number of Tokens -** 19602594

Data Splitting

- After sentence segmentation and word tokenization, we randomly shuffle the data using **random.shuffle**. The data is split into a **training set** and **fixed test set** with **90%** and **10%** sentences respectively.

1.1 - Trigram Language Model

Smoothing

- For large vocabularies, the number of possible trigrams are very large, and many of them are unlikely to occur in the training set.
- If all these are assigned zero probability, this precludes recognition of words with corresponding histories.
- 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 Method

- For any trigram (u,v,w), we first define the discounted count by the following expression where **β** is the **discounting value** between 0 and 1:

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

- By doing this we'll ensure 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.
- Now due to discounting, we will have some missing probability mass, and now we have to divide this "missing mass" between the words which are not present in the training corpus.

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

- Here **α(u,v)** is the **"missing" probability mass**.
- **Calculating the value of β**: We will take help of the log-likelihood test and increment **β** from 0 to 1 gradually and choose that **β** that maximises the log-likelihood value. This process would give us the optimal value of **β**.

Interpolation Method - Linear

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

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

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

- In this method, we estimate our probability from a linear combination of a trigram, bigram and unigram relative frequencies.
- If the vocabulary is confined to words in training sets or those words plus a catch-all unknown words, then this formulation avoids zero probabilities
- Here the context of the previous two words has been completely ignored in the unigram estimate, but its value will always be greater than 0.
- In contrast, trigram estimate does make use of context but has a problem of many of its count being 0. The bigram estimate falls between these two extremes.
- **Calculating the value of λ_i**: 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.

1.2 - Splitting training dataset

We split the original training set after shuffling into **five sets** of **training** and **development** sets each with 90% and 10% sentences of the original training set with the help of **random.shuffle**.

1.3 - Model's performance on the Development set

Discounting Method

Iteration No	Optimal Beta	Perplexity Value after Discounting
1	0.67	402.0237
2	0.62	388.7980
3	0.60	383.4649
4	0.64	368.5363
5	0.67	376.5628

Interpolation Method

Iteration No	Optimal Lambda Set ($\lambda_1, \lambda_2, \lambda_3$)	Perplexity Value after Interpolation
1	[0.3216, 0.4108, 0.2675]	83.3342
2	[0.3218, 0.4110, 0.2671]	83.4726
3	[0.3223, 0.4097, 0.2679]	83.5113
4	[0.3213, 0.4107, 0.2678]	83.1066
5	[0.3209, 0.4104, 0.2686]	82.6550

1.4 - Model's performance on the Test set

Discounting Method

Iteration No	Optimal Beta	Perplexity Value after Discounting
1	0.67	372.2246
2	0.65	371.5809
3	0.69	377.7207
4	0.68	376.5780
5	0.67	374.9597

Interpolation Method

Iteration No	Optimal Lambda Set ($\lambda_1, \lambda_2, \lambda_3$)	Perplexity Value after Interpolation

1	[0.3211, 0.4095, 0.2693]	82.8701
2	[0.3211, 0.4094, 0.2694]	82.8558
3	[0.3211, 0.4094, 0.2693]	82.8797
4	[0.3211, 0.4095, 0.2692]	82.9154
5	[0.3209, 0.4094, 0.2695]	82.8296

Laplace Smoothing Method

Iteration No.	Perplexity Value after Laplace Smoothing
1	668.4223
2	667.7097
3	666.1954
4	666.4257
5	665.7075

1.5 - Laplace Smoothing and Summary

Note - All the readings below are taken on the test data.

	Average	Variance
Perplexity after Discounting	374.612	5.7137
Perplexity after Interpolation	82.87012	0.00079
Perplexity after Laplace	666.89212	1.0232
Optimal λ_1	0.32106	$6.4 * 10^{-9}$
Optimal λ_2	0.40944	$2.4 * 10^{-9}$
Optimal λ_3	0.26934	$1.04 * 10^{-8}$
Optimal Beta	0.672	0.000176

OBSERVATIONS -

We observe that the highest perplexity value is achieved after laplace smoothing. Among discounting and interpolation smoothing methods, lower perplexity has been achieved with interpolation. So, in terms of **performance evaluation** the following trend - **Interpolation > Discounting > Laplace** is observed (as smaller the perplexity value, better the performance). In case of variance, maximum variance is achieved after discounting and minimum variance is achieved after interpolation. So the

trend in terms of **variance of perplexity** is **Discounting > Laplace > Interpolation**. While calculating the variance for the optimal set of parameters, we observe that they have variance very close to zero. This implies that the different training sets used in each iteration does not alter the optimal parameters required for best performance.

TASK 2 - Vector Semantics: GloVe implementation

2.1 - Implementation

- First, we build vocabulary of the given corpus containing the frequency of each word.
- Then we build the co-occurrence matrix which is being represented by a list of tuples containing **{centre_word ID: i, context_word ID: j, co-occurrence: Xij}**. This is done to optimize the use of memory.
- 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)**.
- Finally, GloVe word embeddings are formed by adding the centre word vector and context word vector for each word.

Optimization method used : Adaptive Gradient Descent (AdaGrad)

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.

Vocabulary size: 76825 (10% of corpus taken due to RAM issues)
The following parameters have been used for GloVe implementation :

Parameter	Value	Explanation
DIM_SIZE	100	Dimensions of the vector for word embeddings
ALPHA	0.75	Value of alpha used in the weighting function
X_MAX	100	Value of x_max used in the weighting function
CONTEXT_WINDOW	10	Size of the window taken during the co-occurrence matrix.
ITERATIONS	45	Number of times to run AdaGrad for training
LEARNING_RATE	0.05	The initial learning rate for AdaGrad optimization method

2.2 - Comparison of embeddings and Word-similarity benchmark

WORD EMBEDDINGS BENCHMARK - Link to GitHub - [Click Here]

This tool is used for **word-similarity comparison** between the custom embeddings obtained by the implementation and the pre-trained GloVe embeddings. It is focused on providing methods for easy evaluating and reporting results on common benchmarks (analogy, similarity and categorization). Some of the popular word-similarity benchmark datasets used for the comparison are -

- **MTurk:** This is developed by Amazon and consists of **287**-word pairs.
- **RW:** The Stanford Rare Word (RW) data is one of the highly used benchmarks for rare word representation techniques. The dataset contains **2034** word pairs which are selected in a way to reflect words with low occurring frequency from Wikipedia.
- **WS353:** This is a test collection for measuring word similarly and contains a split of the test set into two subsets, one for calculating similarity, and other for evaluation relatedness.
- **MEN:** Contains around **3000** English word pairs together with human-assigned similarity judgements. The word pairs are randomly selected from words that occur in freely available corpora.
- **SimLex999:** Provides a way of measuring how well models capture similarity, rather than relatedness or association.

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.
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 [-1, -6, 3, 11], the ranked values will be [2, 1, 3, 4]. We can calculate Spearman's correlation by the following expression:

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

where

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

It can take a **range of values from -1 to +1**. If there are no repeated data values, a perfect Spearman correlation of +1 or −1 occurs when each of the variables is a perfect monotone function of the other.

The Spearman correlation between two variables will be high when observations have a similar (or identical for a correlation of 1) rank between the two variables and low when observations have a dissimilar rank between the two variables.

WORD SIMILARITY COMPARISONS -

NOTE - The mean vector is used to denote the embedding of the missing word in the dataset.

Benchmark Dataset	Spearman's correlation on implemented GloVe word embeddings	Spearman's correlation on pre-trained GloVe word embeddings
MTurk	0.1714	0.5641
RW	0.2072	0.2307
WS353	0.1900	0.4697
MEN	0.1244	0.5773
SimLex999	0.0687	0.1222

We train the GloVe word embeddings on a vocabulary size of 76,825 words while 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:

- The correlation values are almost half to that of the correlation values of the pre-trained embeddings. Hence, we are able to achieve a close word-similarity relationship to that of the pre-trained GloVe word vectors. The trend also shows that if we use larger data for training, we can increase the correlation values to match with that of the pre-trained GloVe.
- We do not get any negative value of the Spearman's correlation on any benchmark dataset denoting that there is a positive correlation to each other as also illustrated by the pre-trained GloVe embeddings.

2.3 - Necessary expression of Derivatives in AdaGrad optimisation method

NOTE - For the study of the following expressions, reference is taken from Link1 and Link2

- 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.

- We want to build word vectors that retain some useful information about how every pair of words co-occur. We do this by **minimizing objective function J** which evaluates the sum of all squared errors.
- We choose an **f that helps prevent common word pairs from skewing** our objective function two much. When we encounter extremely common word pairs, the function will cut off its normal output and simply return 1. And for all other word pairs, we return some weight in range (0,1) decided by some factor α.
- 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 ⊙ to denote element wise vector multiplication.

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

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

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

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

- Let us consider each parameter as θ_i, so parameter as 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:

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

- In its update rule, Adagrad modifies the general learning rate at each time step t for every parameter based on the past gradients that have been computed for θi.
- 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 ⊙ between G_t and g_t:

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