50.040 Natural Language Processing, Fall 2024

Mini project

Due 25 October 2024, 23:59pm

This is an individual project
The project will be graded by Chen Huang

Introduction

Language models are very useful for a wide range of applications, e.g., speech recognition and machine translation. Consider a sentence consisting of words $x_1, x_2, \ldots, x_m$, where m is the length of the sentence, the goal of language modeling is to model the probability of the sentence, where $m \geq 1$, $x_i \in V$ and V is the vocabulary of the corpus:

$$p(x_1, x_2, \ldots, x_m)$$

In this project, we are going to explore both statistical language models and neural language models on the Wikitext-2 datasets.

Statistical Language Model

A simple way is to view words as independent random variables (i.e., zero-th order Markovian assumption). The joint probability can be written as:

$$p(x_1, x_2, \ldots, x_m) = \prod_{i=1}^{m} p(x_i)$$

However, this model ignores the word order information, to account for which, under the first-order Markovian assumption, the joint probability can be written as:

$$p(x_0, x_1, x_2, \ldots, x_m) = \prod_{i=1}^{m} p(x_i \mid x_{i-1})$$

Under the second-order Markovian assumption, the joint probability can be written as:

$$p(x_{-1}, x_0, x_1, x_2, \ldots, x_m) = \prod_{i=1}^{m} p(x_i \mid x_{i-2}, x_{i-1})$$

Similar to what we did in HMM, we will assume that $x_{-1} = \text{START}$, $x_0 = \text{START}$, $x_m = \text{STOP}$ in this definition, where START, STOP are special symbols referring to the start and the end of a sentence.

Parameter Estimation

Let's use $\text{count}(u)$ to denote the number of times the unigram u appears in the corpus, use $\text{count}(v, u)$ to denote the number of times the bigram (v, u) appears in the corpus, and $\text{count}(w, v, u)$ the times the trigram (w, v, u) appears in the corpus, $u \in V \cup \{\text{STOP}\}$ and $w, v \in V \cup \{\text{START}\}$. The parameters of the unigram, bigram and trigram models can be obtained using maximum likelihood estimation (MLE).

In the unigram model, the parameters can be estimated as:

$$p(u) = \frac{\text{count}(u)}{c}$$

where c is the total number of words in the corpus. In the bigram model, the parameters can be estimated as:

$$p(u \mid v) = \frac{\text{count}(v, u)}{\text{count}(v)}$$

In the trigram model, the parameters can be estimated as:

$$p(u \mid w, v) = \frac{\text{count}(w, v, u)}{\text{count}(w, v)}$$

Smoothing the Parameters

It is likely that many parameters of bigram and trigram models will be 0 because the relevant bigrams and trigrams involved do not appear in the corpus. If you don't have a way to handle these 0 probabilities, all the sentences that include such bigrams or trigrams will have probabilities of 0.

We'll use an Add-k Smoothing method to fix this problem, the smoothed parameter can be estimated as:

$$p_{\text{add-k}}(u) = \frac{\text{count}(u) + k}{c + k|V^*|}$$

$$p_{\text{add-k}}(u \mid v) = \frac{\text{count}(v, u) + k}{\text{count}(v) + k|V^*|}$$

$$p_{\text{add-k}}(u \mid w, v) = \frac{\text{count}(w, v, u) + k}{\text{count}(w, v) + k|V^*|}$$

where $k \in (0, 1)$ is the parameter of this approach, and $|V^*|$ is the size of the vocabulary V^*, where $V^* = V \cup \{\text{STOP}\}$. One way to choose the value of k is by optimizing the perplexity of the development set, namely to choose the value that minimizes the perplexity.

Perplexity

Given a test set D' consisting of sentences $X^{(1)}, X^{(2)}, \ldots, X^{(|D'|)}$, each sentence $X^{(j)}$ consists of words $x_1^{(j)}, x_2^{(j)}, \ldots, x_{n_j}^{(j)}$, we can measure the probability of each sentence s_i, and the quality of the language model would be the probability it assigns to the entire set of test sentences, namely:

$$\prod_j p(X^{(j)})$$

Let's define average $\log_2$ probability as:

$$l = \frac{1}{c'} \sum_{j=1}^{|D'|} \log_2 p(X^{(j)})$$

where c' is the total number of words in the test set, and $|D'|$ is the number of sentences. The perplexity is defined as:

$$\text{perplexity} = 2^{-l}$$

The lower the perplexity, the better the language model.

Questions (40 Points in total)

Question 1.1 [code] (4 points)

Implement the function *compute_ngram* that computes n-grams in the corpus.(Do not take the START and STOP symbols into consideration.)

Question 1.2 [code] (2 points)

List 5 most frequent unigrams, bigrams and trigrams as well as their counts.(Hint: use the built-in function *.most_common* in Counter class)

Question 2 [code] (4 points)

Now, we take the START and STOP symbols into consideration. So we need to pad the *train_sents* as described in "Statistical Language Model" before we apply *compute_ngram* function. For example, given a sentence "I like NLP", in a bigram model, we need to pad it as "START I like NLP STOP", in a trigram model, we need to pad it as "START START I like NLP STOP".

- Implement the *pad_sents* function.
- Pad *train_sents*.
- Apply *compute_ngram* function to these padded sents.
- Implement *ngram_prob* function. Compute the probability for each n-gram in the variable **ngrams** according equations in "Parameter estimation". List down the n-grams that have 0 probability.

Question 3 [code] (4 points)

- Implement the *add_k_smoothing_ngram* function to estimate n-gram probability with the *add-k* smoothing technique.
- Implement the *interpolation_ngram* function to estimate n-gram probability with the *interpolation* smoothing technique.
- Implement the *perplexity* function to compute the perplexity of the corpus *valid_sents* according to the **Perplexity** section. The computation of $p(X^{(j)})$ depends on the n-gram model you choose.

Question 4 [code][written] (4 points)

- Based on the add-k smoothing method, try out different $k \in [0.0001, 0.001, 0.01, 0.1, 0.5]$ and different n-gram models (unigram, bigram, and trigram). Find the model and k that gives the best perplexity on *valid_sents* (smaller is better).
- Based on the interpolation method, try out different λ where $\lambda_1 = \lambda_2$ and $\lambda_3 \in [0.1, 0.2, 0.4, 0.6, 0.8]$. Find the λ that gives the best perplexity on *valid_sents* (smaller is better).
- Based on the methods and parameters provided, choose the method that performs best on the validation data.

Question 5 [code] (4 points)

Evaluate the perplexity of the test data *test_sents* based on the best model you choose in **Question 4**.

Neural Language Model

Using the chain rule, the probability of a sentence consisting of words $x_1, x_2, \ldots, x_n$ can be represented as:

$$p(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} p(x_t \mid x_{t-1}, \ldots, x_1)$$

Assume that we can use a hidden vector $h_t \in \mathbb{R}^d$ of a recurrent neural network (RNN) to record the history information of words:

$$h_t = \text{RNN}(x_t, h_{t-1})$$

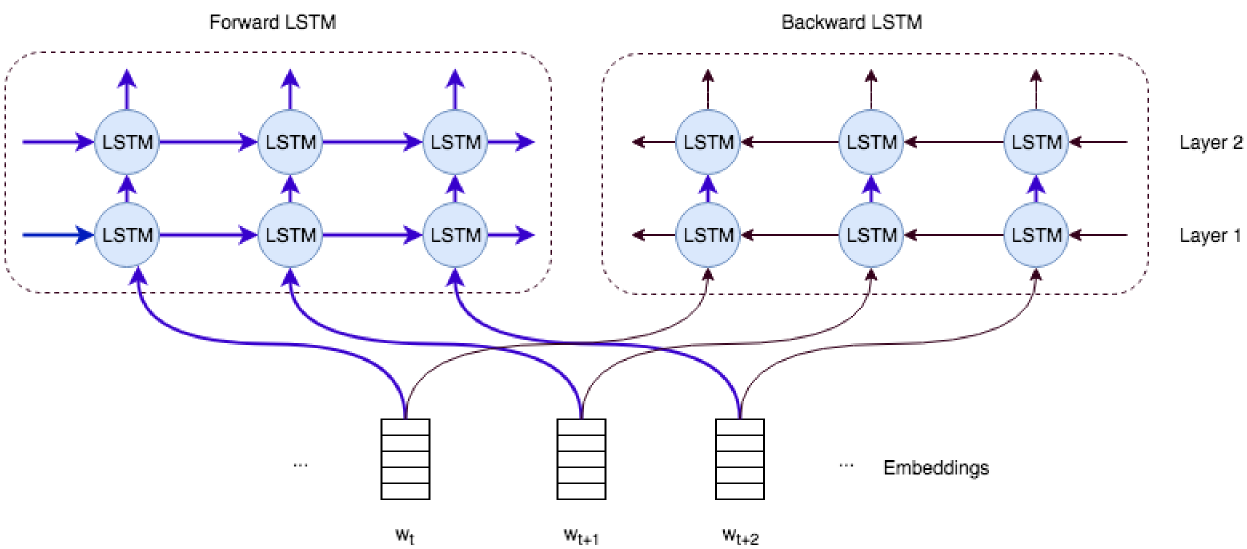

Figure 1: 2-layer Bidirectional LSTM Language Model Architecture

The conditional probability of word x_{t+1} can be parameterized as:

$$p(x_{t+1} \mid x_t, x_{t-1}, \ldots, x_1) \propto \exp(f(w_{x_{t+1}} h_t))$$

where d is the dimension size of the hidden layer, $|V|$ is the size of the vocabulary, f is a fully-connected layer, where $w \in \mathbb{R}^{|V| \times d}$ are the parameters, $w_{x_{t+1}}$ is the parameter in the row that corresponds to the index of x_{t+1} in the vocabulary (bias omitted).

Question 6 [code] (10 points)

We will create a LSTM language model, and train it on the Wikitext-2 dataset. The data generators (train_iter, valid_iter, test_iter) have been provided. The word embeddings together with the parameters in the LSTM model will be learned from scratch.

Pytorch and **torchtext** are required in this part. Do not make any changes to the provided code unless you are requested to do so.

- Implement the *__init__* function in the `LangModel` class.
- Implement the *forward* function in the `LangModel` class.
- Complete the training code in the *train* function. Then complete the testing code in the *test* function and compute the perplexity of the test data `test_iter`. The test perplexity should be below 150.

Question 7 [code][written] (8 points)

We will use the hidden vectors (the working memory) of LSTM as the contextual embeddings.

- Implement the *contextual_embedding* function.
- Use the *contextual_embedding* function to get the contextual embeddings of the word "play" in three sequences: "to play", "dance play", and "sing play". Then calculate the cosine similarity of "play" from each pair of sequences: "to play", "dance play", and "sing play". Assume that $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ are embeddings of "play" in the sequences "to play" and "dance play" respectively. The cosine similarity can be calculated as

$$\text{similarity} = \cos(\theta) = \frac{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{w}_2}{\|\boldsymbol{w}_1\|_2\|\boldsymbol{w}_2\|_2} \tag{1}$$

  Give an explanation of the results.

Requirements

- This is an individual project.
- Write down names and IDs of students with whom you have discussed (if any).
- You should **NOT** copy other's answer, once discovered, the person will get **0** in this mini project.
- Complete answers and Python code in the "mini_project.ipynb" file.
- Follow the honor code strictly.
- Submit the file before the due on eDimension system.