SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

50.040 Natural Language Processing, Summer 2020
Mini Project

Due 19 June 2020, 5pm

Mini Project will be graded by Li Haoran

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 model and neural language model on the Wikitext-2 datasets.

Requirements torch/torchtext/nltk/numpy

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:

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

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:

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

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

$$p(x_{-1}, x_0, x_1, x_2, \ldots, x_m) = \prod_{i=1}^{m} p(x_i \mid x_{i-2}, x_{i-1}) \tag{3}$$

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

Parameter estimation Let's use $count(u)$ to denote the number of times the unigram u appears in the corpus, use $count(v, u)$ to denote the number of times the bigram v, u appears in the corpus, and $count(w, v, u)$ the times the trigram w, v, u appears in the corpus, $u \in V \cup STOP$ and $w, v \in V \cup START$. And 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:

$$p(u) = \frac{count(u)}{c} \tag{4}$$

, where c is the total number of words in the corpus.

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

$$p(u \mid v) = \frac{count(v, u)}{count(v)} \tag{5}$$

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

$$p(u \mid w, v) = \frac{count(w, v, u)}{count(w, v)} \tag{6}$$

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 a Add-k Smoothing method to fix this problem, the smoothed parameter can be estimated as:

$$p_{add-k}(u) = \frac{count(u) + k}{c + k|V^*|} \tag{7}$$

$$p_{add-k}(u \mid v) = \frac{count(v, u) + k}{count(v) + k|V^*|} \tag{8}$$

$$p_{add-k}(u \mid w, v) = \frac{count(w, v, u) + k}{count(w, v) + k|V^*|} \tag{9}$$

where $k \in (0, 1)$ is the parameter of this approach, and $|V^*|$ is the size of the vocabulary V^*,here $V^* = V \cup STOP$. 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 $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(|D'|)}$, each sentence $\mathbf{x}^{(j)}$ 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:

$$\prod_{j}^{D'} p(\mathbf{x}^{(j)}) \tag{10}$$

Let's define average log2 probability as:

$$l = \frac{1}{c'} \sum_{j=1}^{|D'|} log_2 p(\mathbf{x}^{(j)}) \tag{11}$$

c' is the total number of words in the test set, D' is the number of sentences. And the perplexity is defined as:

$$perplexity = 2^{-l} \tag{12}$$

(The lower the perplexity, the better the language model)

Statistical Language Model (30 points)

Question 1 [code][written] (3+1 points)

1. Implement the function *compute_ngram* that computes n-grams in the corpus. (Do not take the START and STOP symbols into consideration for now.) For n=1,2,3, the number of unique n-grams should be 28910/577343/1344047, respectively.
2. List 3 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][written] (3+1+1+5 points) In this part, 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".

1. Implement the *pad_sents* function.
2. Pad "train_sents".
3. Apply *compute_ngram* function to these padded sents. For n=1,2,3, the number of unique n-grams should be 28910/580825/1363266, respectively.
4. Implement *ngram_prob* function. Compute the probability for each n-gram in the variable "ngrams" according to Eq.(7),(8),(9) in "smoothing the parameters". List down the n grams that have zero probability.

Question 3 [code][written] (5+7+3 points)

1. Implement *smooth_ngram_prob* function to estimate ngram probability with "add-k" smoothing technique. Compute the smoothed probabilities of each n-gram in the variable "ngrams" according to Eq.(7)(8)(9) in "smoothing the parameters".

2. Implement *perplexity* function to compute the perplexity of the validation data “valid_sents” according to the Equations (10),(11),(12) in “perplexity” section. The computation of $p(\mathbf{x}^{(j)})$ depends on the n-gram model you choose. If you choose 2-gram model, then you need to calculate $p(\mathbf{x}^{(j)})$ based on Eq.(8) in “smoothing the parameter” section. Hint: convert probability to log probability.

3. Try out different $k \in [0.1, 0.3, 0.5, 0.7, 0.9]$ and different n-gram ($n \in [1, 2, 3]$) models. Find the n-gram model and k that gives the best perplexity (smaller is better) on validation data “valid_sents”.

Question 4 [code] (1 points) Evaluate the perplexity of the test data “test_sents” based on the best n-gram model and k value you have found on the validation data.

Neural Language Model (RNN) (20+10 points) Suppose we have a training set $(\mathbf{x}^{(j)}, \mathbf{y}^{(j)}) \in \mathcal{D}, j \in \{1, 2, ..., |\mathcal{D}|\}$, where $\mathbf{x}^{(j)} = (x_0^{(j)}, x_1^{(j)}, ..., x_n^{(j)})$ is the input sentence and $\mathbf{y}^{(j)} = (x_1^{(j)}, x_2^{(j)}, ..., x_{n+1}^{(j)})$ is the sentence our model will predict. Typically, $x_i^{(j)} \in \mathbb{R}^{|V|}$ is a one-hot vector, where $|V|$ is the size of the vocabulary. $x_0^{(j)}$ is a special “START” token and $x_{n+1}^{(j)}$ is a special “END” token. A neural language model (RNN) will be trained to predict the next word x_{t+1} given the current word x_t and history information of words h_{t-1}.

$$p(x_1, x_2, ..., x_{n+1} \mid x_0) = \prod_{t=0}^{n} p(x_{t+1} \mid x_t, ..., x_0) = \prod_{t=0}^{n} p(x_{t+1} \mid h_t) \tag{13}$$

where $h_t = \text{RNN}(x_t, h_{t-1})$ denotes the history information of words. h_0 is usually set to be a zero vector.

In a neural language model, we first map a sequence of words $x_1^{(j)}, ..., x_n^{(j)}$ to word embeddings (we don’t consider special token for simplicity), $e_1^{(j)}, ..., e_n^{(j)}$, where $e_i^{(j)} \in \mathbb{R}^k$. Then, we feed those embeddings into a RNN model (a LSTM in the project), obtaining a sequence of hidden states $h_1^{(j)}, h_2^{(j)}, ..., h_n^{(j)}$, where $h_i^{(j)} \in \mathbb{R}^d$. Next, we apply a linear transformation $W \in \mathbb{R}^{|V| \times d}$ to those hidden states, yielding $\hat{h}_1^{(j)}, \hat{h}_2^{(j)}, ..., \hat{h}_n^{(j)}$, where $\hat{h}_i^{(j)} \in \mathbb{R}^{|V|}$. We apply “softmax” function to each $\hat{h}_i^{(j)}$, yielding a probability distribution $\hat{y}_i^{(j)} \in \mathbb{R}^{|V|}, i \in [1, 2, ..., n]$ over the vocabulary V. $\hat{y}_i^{(j)}$ represents the probability of a certain word at position i. Finally, we compute the “Negative Log Likelihood Loss” between model’s predictions $\hat{y}_i^{(j)}, i \in \{1, 2, ..., n\}$ and gold targets $\mathbf{y}^{(j)}$.

Question 5 [code] (3+4+3 points) We will create a LSTM language model as shown in figure 1 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.

1. Implement the “__init__” function in “LangModel” class.

2. Implement the “forward” function in “LangModel” class.

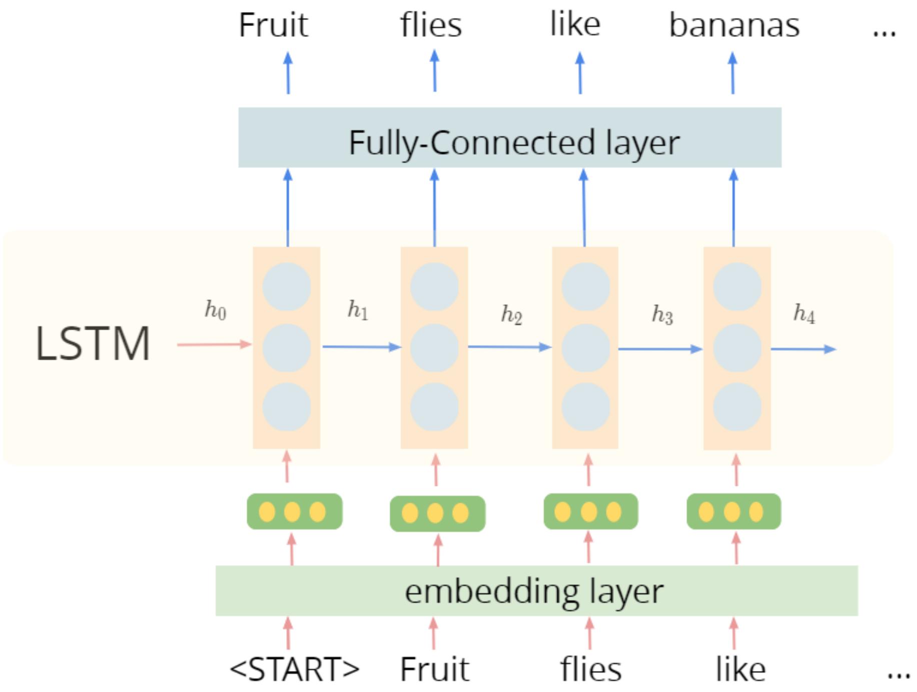

Figure 1: LSTM Language Model

3. Complete the training code in *train* function, tune hyperparameters on the validation data "valid_iter". Then complete the testing code in *test* function and compute the perplexity of the test data "test_iter". The test perplexity should be below *150*.

Question 6 [code] (5+10 points)

1. Implement the "word_greedy_search" function. This function takes an arbitrary token as well as a model as input and returns a sentence that starts with this input token.

2. **[optional]** Implement the "word_beam_search" function. This function takes an arbitrary token, a model and an integer (*beam_size*) as input. It returns the top *beam_size* sentences that have the highest probabilities and all of them start with the given input token. When $beam\_size = 1$, this function is equivalent to "greedy_search".

Question 7 [code] (1+4 points) A RNN language model usually is usually built on word tokens, like the one you just implemented. However, we can also build a RNN language model simply based on characters. For example, a sentence "START, fruit flies like ..." will be tokenized as "START, 'f', 'r', 'u', 'i', 't', ' ', 'f', 'l'... ". Then, we use this character sequence to train our language model.

1. Implement the "char_tokenizer" function which takes a string as input and returns a list of characters.

2. Implement “CharLangModel”, “char_train”, “char_test”, “char_greedy_search” function according to your code in previous sections.

How to submit

1. Fill up your student ID and name in the Jupyter Notebook.

2. Click the Save button at the top of the Jupyter Notebook.

3. Select Cell - All Output - Clear. This will clear all the outputs from all cells (but will keep the content of all cells).

4. Select Cell Run All. This will run all the cells in order and will take several minutes.

5. Once you’ve rerun everything, select File – Download as – html and then export html to PDF, or you can directly print html and save as PDF.

6. Look at the PDF file and make sure all your solutions are there, displayed correctly. The PDF is the only thing your graders will see! Submit your PDF on eDimension.