Assignment 1: N-gram Language Model & Tokenizations

Instructor: Yejin Choi *CSE 517/447 Win 24*

Due at 11:59pm PT, Jan 24, 2024 (100 pt for 447 / 110 pt for 517, 14% towards the final grade)

In this assignment, you will implement and experiment with n-gram language models and two tokenizations algorithms, including the Byte-Pair Encoding (BPE) and WordPiece.

You will submit both your **code** (in Assignment 1 - code) and **write-up** (in Assignment 1 - write-up) via Gradescope. Your final grade for each question will be based on both the write-up and the submitted code. Remember to **specify your collaborators** (including AI tools like ChatGPT) and **how they contribute** to the completion of your assignment at the beginning of your write-up. If you work on the assignment independently, please specify so, too. **NOT properly acknowledging your collaborators will result in -2 % of your overall score on this assignment.** Please adhere to the assignment collaboration policy specified on the course website.

Required Deliverables

Please include all the write-ups in a **single PDF**, and assign the page for each part on Gradescope. Failing to assign the page correctly can result in a penalty. We recommend you start the write-up for each part on a new page.

Besides submitting your write-up, please submit the following required files as a single compressed file to Gradescope in a different submission portal. You should name your compressed file as **Assignment_1_UWNetID**. Here's a list of required files.

- **ngram.py** or **ngram.ipynb**: the code file for the N-gram LM part of the assignment.
- **bpe.py** or **bpe.ipynb**: the code file for the BPE tokenization part of the assignment.
- **wordpiece.py** or **wordpiece.ipynb**: the code file for the WordPiece tokenization part of the assignment. (You are also welcome to submit a single file for both BPE and WordPiece, and name them **tokenization.py** or **tokenization.ipynb** if that makes things easier for you. However, please still include all required functions in this combined file.)

1 N-gram Language Models (45%)

Download Data: The data you need for the N-gram LM part of this assignment is available here.
Write-up: Your write-up for this problem should be **no more than 3 pages**.

1.1 Dataset

We provide you with three data files (a subset of the One Billion Word Language Modeling Benchmark). Each line in each file contains a whitespace-tokenized sentence.

- **1bbenchmark.train.tokens:** data for training your language models.
- **1bbenchmark.dev.tokens:** data for debugging and choosing the best hyperparameters.
- **1bbenchmark.test.tokens:** data for evaluating your language models.

A word of caution: You will primarily use the **development/validation** dataset as the previously unseen data while (i) developing and testing your code, (ii) trying out different model and training design decisions, (iii) tuning the hyperparameters, and (iv) performing error analysis (not applicable in this assignment, but a key portion of future ones).
For scientific integrity, it is extremely important that you use the test data only once, just before you report all the final results. Otherwise, you will start overfitting on the test set indirectly. **Please don't be tempted to run the same experiment more than once on the test data.**

1.2 N-gram Language Modeling & Perplexity (28%)

Build N-gram Models: You will build **unigram**, **bigram**, and **trigram** language models. To handle out-of-vocabulary (OOV) words, convert tokens that occur less than three times in the training data into a special ⟨`unk`⟩ token during training. If you did this correctly, your language model's vocabulary (including the ⟨`unk`⟩ token and STOP, but excluding START) should have 26,602 types.

Evaluate Your Models with Perplexity: After implementing your N-gram models, you should implement two versions of the *perplexity* function to evaluate your models: (1) one without any smoothing technique; (2) one with Laplace smoothing.
Recall that Laplace smoothing adds one to each token count; hence it's alternately named add-one smoothing. Remember to also adjust the denominator too, as each token in the vocabulary was incremented by one count.

Deliverables:

1. **Code:**
   - Please put all your code for the n-gram models part in a file named ngram.py or ngram.ipynb depends on whether you run your code in raw Python files or Jupyter notebooks.
   - **(5%)** In your submitted file, you should have three functions called **unigram()**, **bigram()**, and **trigram()** that contain your implementations of these models, respectively. You should also implement two versions of the perplexity functions: (1) **perplexity()** without any smoothing; (2) **perplexity_laplace()** with the Laplace smoothing.
   - Your file can contain other necessary helper functions for experimentation, but the above functions are required and must follow our suggested function names.
   - Your submission will not be evaluated for efficiency, but we recommend keeping such issues in mind to better streamline the experiments.
2. **Write-up:**
   (a) **(5%)** Describe how you built your N-gram models. Provide graphs, tables, charts or other summary evidence to support any claims you make.
   (b) **(3%)** Describe how you computed *perplexity* without any smoothing in a detailed equation, in natural language, or with pseudo code.
   (c) **(4%)** What's the potential downside of Laplace smoothing? Back up your claim with empirical evidence.
   (d) **(4%)** Another extension of Laplace smoothing, instead of adding 1 to the count of each token, is to add k, where typically $0 < k < 1$. Try different values for k, and describe how different k change the perplexity score differently.
   (e) **(7%)** Report the *unsmoothed* and *smoothed with Laplace smoothing version* of perplexity scores of the unigram, bigram, and trigram language models for your training, development, and test sets. Briefly discuss the experimental results.

1.3 Interpolation (17%)

To make your language model work better, you will implement linear interpolation smoothing between unigram, bigram, and trigram models:

$$\theta'_{x_j|x_{j-2},x_{j-1}} = \lambda_1\theta_{x_j} + \lambda_2\theta_{x_j|x_{j-1}} + \lambda_3\theta_{x_j|x_{j-2},x_{j-1}}$$

where θ' represents the smoothed model parameters, and the hyperparameters λ_1, λ_2, λ_3 are weights on the unigram, bigram, and trigram language models, respectively. $\lambda_1 + \lambda_2 + \lambda_3 = 1$.
You should use the development data to choose the best values for the hyperparameters. Hyperparameter optimization is an active area of research; for this homework, you can simply try a few combinations to find reasonable values.

Deliverables: In this part of the assignment, report perplexity scores **without Laplace smoothing**.

1. **Code:**
   - You should put the code for all smoothing experiments into the ngram.py or ngram.ipynb files.
2. **Write-up: Fully describe your models and experimental procedure.** Provide graphs, tables, charts or other summary evidence to support any claims you make.
   (a) **(5%)** Your goal is to find reasonably good combinations of λ_1, λ_2, λ_3. Experiment and report perplexity scores on *training* and *development* sets for five sets of values of λ_1, λ_2, λ_3 that you tried, along with short descriptions of the strategies that you used to find better hyperparameters. In addition, report the training and development perplexity for the values $\lambda_1 = 0.1$, $\lambda_2 = 0.3$, $\lambda_3 = 0.6$.
   (b) **(2%)** Putting it all together, report perplexity on the *test* set, using the *best* combination of hyperparameters that you chose from the development set. Specify those hyperparameters.
   (c) **(5%)** If you use half of the training data, would it increase or decrease the perplexity of previously unseen data? Why? Provide empirical experimental evidence to support your claims.
   (d) **(5%)** If you convert all tokens that appeared less than five times to ⟨`unk`⟩ (a special symbol for out-of-vocabulary tokens), would it increase or decrease the perplexity on the previously unseen data compared to an approach that converts only those words that appeared just once to ⟨`unk`⟩? Why? Provide empirical evidence to support your claims.

2 Byte-Pair Encoding (BPE) (30%)

Download Data: The data you need for the BPE part of this assignment is available here.
Write-up: Your write-up for this problem should be **no more than 2 pages**.

2.1 Background

The rare/unknown word issue is ubiquitous in neural text generation. As we saw in Problem 1, words not appearing in the vocabulary need to be replaced with a special ⟨unk⟩ symbol.

Byte-pair encoding (BPE) is a popular tokenization technique that addresses this issue. The idea is to encode text using a set of automatically constructed types, instead of using conventional (white-space-separated) word types. A type can be a character or a subword unit, and the types are built through a iterative process, which we now walk you through.

Walking through an example: Suppose we have the following tiny training data: it unit unites.

Training the Tokenizer:

- **First, we segment the data into characters.** Our initial types are the characters and the special beginning-of-word symbol: { i, t, u, n, e, s, ⟨*s*⟩ }. Using the initial type set, the encoding of the training data is: i t ⟨*s*⟩ u n i t ⟨*s*⟩ u n i t e s. (Note that the beginning-of-word symbol here plays the same rule as the space token used in the lecture, and won't be prepended to a word at the beginning of a line)
- **In each training iteration, the most frequent type bigram is merged into a new symbol and then added to the type vocabulary.** Ties can be broken at random. The bigram count for our initial data is: (i, t) : 3, (⟨*s*⟩, u): 2, (u, n) : 2, (n, i) : 2, (t, e) : 1, (e, s) : 1. Note that we don't consider merges across white-space-separated words.
  - **Iteration 1:**
    - Bigram to merge (with frequency 3): i t
    - Updated data: it ⟨*s*⟩ u n it ⟨*s*⟩ u n it e s
  - **Iteration 2:**
    - Bigram to merge (with frequency 2): ⟨*s*⟩ u
    - Updated data: it ⟨*s*⟩u n it ⟨*s*⟩u n it e s
  - **Iteration 3:**
    - Bigram to merge (with frequency 2): ⟨*s*⟩u n
    - Updated data: it ⟨*s*⟩un it ⟨*s*⟩un it e s
- In this example, we end up with the type vocabulary {i, t, u, n, e, s, ⟨*s*⟩, it, ⟨*s*⟩u, ⟨*s*⟩un}. The stopping criterion can be defined through a target vocabulary size.

Applying to New Words:

- At inference-time, we encode text with BPE by first splitting the word into characters, and then iteratively applying the merge rules in the same order as learned in training. Suppose we want to encode text **itunes unite**. This text is first split into characters, **i t u n e s** ⟨*s*⟩ **u n i t e**.
- Then,
  - **Iteration 1:**

    - Bigram to merge: i t
    - Encoded word: it u n e s ⟨s⟩ u n it e
  - **Iteration 2:**
    - Bigram to merge: ⟨s⟩ u
    - Encoded word: it u n e s ⟨s⟩u n it e
  - **Iteration 3:**
    - Bigram to merge: ⟨s⟩u n
    - Encoded word: it u n e s ⟨s⟩un it e
  - The above procedure is repeated until no merge rule can be applied. In this example we get the encoding: it u n e s ⟨s⟩un it e

2.2 Your Implementation of BPE

Now train a BPE tokenizer on the provided text file. Use the **first 4000 lines** as the training set. **Run the algorithm until the frequency of the most frequent type bigram is two.**

Note that the above procedure isn't exactly how people use BPE in real-life. There are other implementation details like how to handle punctuation that aren't covered. If you would like to see a more comprehensive description of how BPE tokenizer works, please see the tutorial from Huggingface. It can also be helpful for implementing the algorithm, **but please refrain from copy-pasting code directly.**

Deliverables:

1. **Code:**
   - You should put the code for the BPE part of the assignment into the bpe.py or bpe.ipynb files.
   - **(5%)** In your file, please include two functions **bpe_train()** and **bpe_apply()** that train and apply your implementation of the BPE tokenizer, respectively.
2. **Write-up:** After training your BPE tokenizer, answer the following questions in your write-up:
   (a) **(15%)** Please produce a scatterplot showing points (x,y), each corresponding to an iteration of the algorithm, with x the current size of the type vocabulary (including the base vocabulary), and y the length of the training corpus (in tokens) under that vocabulary's types. How many types do you end up with? What is the length of the training data under your final type vocabulary?
   (b) **(5%)** Another way of overcoming the rare word issue is to encode text as a sequence of characters. Discuss the advantages and potential issues of character-level encoding compared to BPE.
   (c) **(5%)** Applying your tokenizer on the last 1000 lines of the data, how many tokens do you end up with? Do you encounter issues with words that didn't appear in the training set? More generally, when would you expect your tokenizer to fail when applying to an unseen text.

3 WordPiece (25%)

Write-up: Your write-up for this problem should be **no more than 2 pages**.

3.1 Background

Another popular tokenization algorithm is WordPiece, which is used by a few BERT-based models. The idea of WordPiece is similar to BPE: if two tokens frequently appear together, then merge them into a new token. However, WordPiece normalizes the bigram frequency by the two tokens' individual frequency. In short, at each iteration, BPE merges the pair (a, b) that maximizes $\text{freq}(ab)$, whereas WordPiece merges the pair (a, b) that maximizes $\text{freq}(ab)/(\text{freq}(a) \times \text{freq}(b))$.

Applying the tokenizer at inference-time is also different in WordPiece. In BPE, we apply each merge iteratively following the order during training. However, in WordPiece, we simply save the final vocabulary and greedily find the longest matching subword until the text is fully encoded. For example, given the vocabulary {u, n, d, o, do, un, und}, the word undo will be tokenized into **und o** in Wordpiece whereas in BPE it will be **un do** as the merge u n must be learned before un d.

3.2 Your Implementation of WordPiece

Now implement WordPiece algorithm on the training set in the previous problem. Again, feel free to refer to the Huggingface tutorial for a more detailed walkthrough, and note that the pre-tokenization procedure you are going to apply in this homework is slightly different from the standard approach. **Update: Run the algorithm until 4000 new merge rules are learned. (i.e.: the vocabulary has size 4000 excluding the base vocabulary)**

Deliverables:

1. **Code:**
   - You should put the code for the WordPiece part of the assignment into the wordpiece.py or wordpiece.ipynb files.
   - **(5%)** In your file, please include two functions **wordpiece_train()** and **wordpiece_apply()** that train and apply your implementation of the WordPiece tokenizer, respectively.
2. **Write-up:** After training your WordPiece tokenizer, answer the following questions:
   (a) **(10%)** Please produce a scatterplot showing points (x,y), each corresponding to an iteration of the algorithm, with x the current size of the type vocabulary (including the base vocabulary), and y the length of the training corpus (in tokens) under that vocabulary's types. How many types do you end up with? What is the length of the training data under your final type vocabulary?
   (b) **(5%)** Applying your tokenizer on the last 1000 lines of the data, report the length of the tokenized data. Also, include the tokenized sequences for the following two sentences:
      i. *"Analysts were expecting the opposite, a deepening of the deficit."*
      ii. *"Five minutes later, a second person arrived, aged around thirty, with knife wounds."*
   (c) **(5%)** In terms of efficiency and performance, what's the advantages and disadvantages of WordPiece compared with BPE? There is no single correct answer here, just provide your thoughts and rationales, supported by empirical evidences.

4 Open-ended Exploration for Tokenization (10% for 517, extra credit for 447)

Write-up: Your write-up for this problem should be **no more than 2 pages**.

Modern language models predominantly employ byte-level Byte Pair Encoding (BPE) tokenization. However, the uneven distribution of text from different languages in the training corpus for tokenizers can lead to imbalanced fragmentation rates, meaning that the same sentence will require more tokens to encode in one language than another. To explore this issue, consider the following questions:

- How does the fragmentation rate of different languages, i.e., the number of tokens required to represent a sentence, differ under the same tokenizer? Qualitatively illustrate your observations by comparing some long sentences in English (∼30 words) with their translations in another language. Use https://huggingface.co/spaces/Xenova/the-tokenizer-playground for exploration and experiment with at least two different tokenizers. You may also use the Huggingface `AutoTokenizer` library for additional exploration. Please document your choice of library and tokenizer, and comment on how you obtained the translations and their quality (as far as you know).
- Aside from the different amounts of data for different languages in the training corpus, what other factors might contribute to varying fragmentation rates across different languages? Provide your rationale.
- Why does a disparity in fragmentation rates for different languages matter? Discuss some implications you foresee, perhaps in terms of efficiency, language modeling quality, or something else.
- Beyond balancing the training corpus, suggest at least one strategy to make tokenizers more equitable across different languages. Describe your idea at a high-level and discuss some potential implications of training a language model with such a tokenizer.

Acknowledgement

This assignment is primarily designed by Muru Zhang, with invaluable feedback from Liwei Jiang, Alisa Liu, Orevaoghene Ahia, Khushi Khandelwal, and Yejin Choi. This homework built off homeworks from previous offerings of NLP, and we thank Noah Smith for sharing his course materials.