Assignment 2: N-Gram Language Models

Instructor: Luke Zettlemoyer *CSED503 - Sp 26*

Due at 11:59pm PT, May 4th 2026

In this assignment, you will learn about N-gram language models: how to train them, evaluate their quality, and generate text using them.

You will submit both your **code** and **writeup** (as PDF) via Gradescope.

Required Deliverables

- **Code Notebook**: Please download the notebook with your solution and submit it in Gradescope. On Google Colab you can do so by `File` → `Download` → `Download .ipynb`.
- **Write-up**: For written answers and open-ended reports, produce a single PDF for and submit it in Gradescope. We recommend using Overleaf to typeset your answers in LaTeX, but other legible typed formats are acceptable. We do not accept hand-written solutions because grading hand-written reports is incredibly challenging.

Recommended Reading

The homework is based on chapter 3 of Jurafsky and Martin. We provide all the details necessary to solve the homework in this handout and the notebook, so it is not required to read the chapter to solve the exercises. However, we recommend going through it if you are confused about any concepts that are covered in the homework.

Acknowledgement

This assignment is adapted by Hamish Ivison from work by Kabir Ahuja with invaluable feedback from Riva Gore, Khushi Khandelwal, Melissa Mitchell, and Kavel Rao. Kavel Rao also helped design autograder for the homework.

1 N-Gram Language Models

In this assignment, you will implement and experiment with N-gram language models. N-gram language models are the simplest versions of a language model, which make a simplifying assumption that the probability of predicting a word in a sentence only depends on the past N words in the sentence. In this assignment, you will learn:

- How to train word-level unigram and N-gram language models on text data
- Evaluating the quality of a language model by computing perplexity
- Sampling text from an N-gram language model
- Implementing Laplace Smoothing
- Implement Interpolation for N-gram Language Models

We will be working with Shakespeare plays data from Andrej Karpathy's blog post on Recurrent Neural Networks. We also recommend going through chapter 3 of Jurafsky and Martin on N-gram models, especially if you are not familiar with them yet.

Notebook: We have designed this part with the following Python notebook: CSED503_Assignment2.ipynb. Please make a copy for yourself by navigating to `File` → `Save a copy in Drive`. Alternatively, when attempting to save, Google Colab will prompt you to save a copy in your own drive. Make your way through the notebook and implement the classes and functions as specified in the instructions. All the data necessary for this assignment can be downloaded within the notebook itself.

Deliverables:

1. **Coding Exercises:** You should complete the code blocks denoted by `YOUR CODE HERE:` in the Python notebook. Do not forget to remove `raise NotImplementedError()` from the code blocks.
2. **Write-up:** Note that the notebook also lists the same write-up questions which we do below, but those should be answered in the write-up pdf only and not in the notebook.

Note on special tokens used in this assignment Please note the following when dealing with special tokens:

- `<eos>` (end-of-sentence): appended to every sentence by `add_eos`. It is counted in the denominator n of perplexity and can be generated during sampling (at which point generation terminates).
- `<sos>` (start-of-sentence): $N-1$ copies are prepended to every sentence by `process_text_for_Ngram` so conditional probabilities are well-defined at the sentence start. It is *never* counted in n and should not be generated during sampling.
- `<unk>` (unknown word): used to stand in for rare/unseen words. After `<unk>` replacement on training data it behaves like any other vocabulary token.
- Your **vocabulary** (and the V in the Add-k denominator) should include `<sos>`, `<eos>`, and `<unk>`. The reference perplexity values assume this convention.

1.1 Unigram Language Models (22 points)

We will start by implementing unigram language models, which constitutes the simplest variant of N-gram models – simply learn the distribution of each unigram (here a word) in the corpus. Recall from the lectures, that for a text sequence with unigrams $w_1, w_2, \cdots, w_n$, unigram language models, the probability of the sequence is given as:

$$P(w_1, w_2, \cdots, w_n) = P(w_1)P(w_2)\cdots P(w_n)$$

where $P(w_i)$ is simply the frequency of the word w_i in the training corpus.

Training a unigram model simply corresponds to calculating the frequencies of each word in the corpus, i.e.,

$$p(w_i) = \frac{C(w_i)}{n}$$

where $C(w_i)$ is the count of word w_i in the training data and n is the total number of words in the training dataset.

Evaluating Unigram Language Models using Perplexity. Now that we have trained our first (albeit very basic) language model, our next job is to evaluate how well it models the training text as well as how well it generalizes to unseen text. The most commonly used metric for evaluating the quality of a language model is Perplexity. Recall from the lecture that the perplexity of a language model on a test dataset measures the (inverse) probability assigned by the language model to the test dataset, normalized by the number of words (or tokens). A lower perplexity indicates a higher probability assigned to the text in the test dataset, and hence better quality.

$$\text{perplexity}(W) = P(w_1 w_2 \cdots w_n)^{\frac{-1}{n}} = \sqrt[n]{\frac{1}{P(w_1 w_2 \cdots w_n)}}$$

where W is a test set with n words $w_1 w_2 \cdots w_n$.

It is useful to calculate perplexity in log space to avoid numerical issues:

$$\text{perplexity}(W) = \exp\left(-\frac{\log P(w_1 w_2 \cdots w_n)}{n}\right)$$

When we have multiple sentences in the corpus and assume sentences to be independent, we can write:

$$\text{perplexity}(W) = \exp\left(-\frac{\sum_{S \in W} \log P(s_1 s_2 \cdots s_{n_s})}{n}\right)$$

where S is a sentence in the corpus W with words $s_1, s_2, \ldots, s_{n_s}$ and n_s is the number of words in S.

Note that assuming sentences to be independent is not actually true in practice. However, since N-gram language models are already limited in their context, it is not a significant loss to remove dependencies between sentences. In future homework assignments, we will drop this assumption as we build more powerful models.

Sampling from Unigram Language Model Now that we have trained and evaluated our unigram LM, we are ready to generate some text from it. To sample text from an N-gram language model given prefix words $w_1, w_2, \cdots, w_n$, we sequentially sample next tokens from the N-gram probability distribution given the previous words, i.e.,

$$w_{n+1} \sim P(w_{n+1}|w_1, \cdots, w_n)$$

For a unigram language model, since $P(w_1, \ldots, w_n) = P(w_1) \cdots P(w_n)$ i.e. all words all distributed independently and the next token is sampled independent of previous tokens, the above equation simplifies to:

$$w_{n+1} \sim P(w_{n+1})$$

1.1.1 Coding Exercises (16 points).

Implement the following functions in Part 1 (Word-level unigram language models) of the notebook.

- function `add_eos`
- function `train_word_unigram` (4 points)
- function `eval_ppl_word_unigram` (4 points)
- functions `replace_rare_words_wth_unks` and `eval_ppl_word_unigram_wth_unks` (4 points)
- function `sample_from_word_unigram` (4 points)

1.1.2 Write-Up Questions (6 points)

We recommend answering the write-up questions once you have finished the coding exercises in this section.

1. **Effect of `<unk>` tokens on perplexity and generation quality (4 points).** What effect do you think the `<unk>` tokens will have on the perplexity of the model? Try out different thresholds for replacing rare words with the `<unk>` token and report the perplexity on the training and development sets. What do you observe? Does the perplexity decrease with increasing threshold? Do you think that improves generation quality?

   What to submit:

   - A table with the perplexity of the unigram model on the training and development sets for different thresholds of replacing rare words with the `<unk>` token. You can choose the thresholds as 1, 3, 5, 7, 9, 10.
   - Example generations at each threshold.
   - A maximum of 3-4 lines discussing the trends you observe in the perplexity and generation quality with increasing threshold.

2. **Alternatives to Random Sampling (2 Points)** An alternate algorithm to generate text from a language model is greedy decoding, i.e., where we generate the most likely token at each step of decoding, i.e.

   $$w_{n+1} = \texttt{argmax}_w P(w \mid w_1, \cdots, w_n)$$

   Can you explain why or why not that will be a good idea for unigram language models? Explain in no more than 3 lines.

1.2 N(>1)-Gram Word-Level Language Models (24 points)

We will now implement much more sophisticated language models, which make use of the surrounding text to model the distribution of text. Recall from the lectures for an N-gram language model with $n > 1$, the distribution of a sequence of tokens $w_1, w_2, \cdots, w_n$ is given as:

$$P(w_1, w_2, \cdots, w_n) = \prod_{k=1}^{n} P(w_k \mid w_{k-N-1}, \cdots, w_{k-1})$$

E.g., for a bigram model i.e. $N = 2$, the expression becomes:

$$P(w_1, w_2, \cdots, w_n) = \prod_{k=1}^{n} P(w_k \mid w_{k-1})$$

i.e. the distribution of a token depends solely on the past $N-1$ tokens in the sequence.

At the heart of training an N-gram language model is to estimate the conditional distributions $P(w_n \mid w_{n-N-1}, \cdots, w_{n-1})$. Recall from the lectures that the conditional distributions can be estimated as:

$$P(w_n \mid w_{n-N-1}, \cdots, w_{n-1}) = \frac{C(w_{n-N-1} \cdots w_{n-1} w_n)}{\sum_{w \in \mathcal{W}} C(w_{n-N-1} \cdots w_{n-1} w)} = \frac{C(w_{n-N-1} \cdots w_{n-1} w_n)}{C(w_{n-N-1} \cdots w_{n-1})}$$

where $C(w_{n-N-1} \cdots w_{n-1} w)$ is the number of times the token sequence $w_{n-N-1} \cdots w_{n-1} w$ appears in the corpus, and $\mathcal{W}$ is the vocabulary of the N-gram model.

Note that for perplexity computation and sampling text, you can just refer to the general expressions we provide in the §1.1 and we do not repeat them here.

1.2.1 Coding Exercises (20 points).

Implement the following functions and classes in Part 2 (N(>1)-Gram Word-Level Language Models) of the notebook.

- function `process_text_for_Ngram`
- class `WordNGramLM` (20 points)

1.2.2 Write-Up Questions (4 points).

1. **Probability distribution of sentence modeled by an N-gram LM (4 points)** Can you show why for an N-gram language model the following expression holds?

$$P(w_1, w_2, \cdots, w_n) = \prod_{k=1}^{n} P(w_k \mid w_{k-N-1}, \cdots, w_{k-1})$$

   Lay down the assumption that is required to derive this expression and show all the steps in your derivation.

1.3 Smoothing and Interpolation in N-Gram LMs (54 points)

The issue with using N-gram language models is that any finite training corpus is bound to miss some N-grams that appear in the test set. The models hence assign zero probability to such N-grams, leading to probability of the entire test set to be zero and hence infinite perplexity values that we observed in the previous exercise.

The standard way to deal with zero-probability N-gram tokens is to use smoothing algorithms. Smoothing algorithms shave off a bit of probability mass from some more frequent events and give it to unseen

events. Smoothing algorithms for N-gram language models is a well studied area of research with numerous algorithms. For this assignment we will focus on Laplace Smoothing and Interpolation.

Laplace and Add-k Smoothing. Perhaps the simplest smoothing algorithm that exists is Laplace smoothing. It merely adds one to the count of each N-gram, so that there is no zero-probability N-gram in the test data. For a bigram model, the expression for the Laplace-smoothed distribution is given by:

$$P_{\text{Laplace}}(w_n \mid w_{n-1}) = \frac{C(w_{n-1}w_n) + 1}{\sum_{w \in \mathcal{W}}(C(w_n w) + 1)} = \frac{C(w_{n-1}w_n) + 1}{C(w_{n-1}) + V}$$

We can similarly write expressions for other N-gram models.

Laplace smoothing is also called "Add-one" smoothing. A generalization of Laplace smoothing is "Add-k" smoothing with $k < 1$, where we move a bit less of the probability mass from seen to unseen N-grams. The expression for the Add-k smoothed distribution for the bigram language model is given by:

$$P_{\text{Add-k}}(w_n \mid w_{n-1}) = \frac{C(w_{n-1}w_n) + k}{\sum_{w \in \mathcal{W}}(C(w_n w) + k)} = \frac{C(w_{n-1}w_n) + k}{C(w_{n-1}) + kV}$$

Language Model Interpolation. An alternate to smoothing that often works well in practice is interpolating between different language models. Let's say we are trying to compute $P(w_n \mid w_{n-2}w_{n-1})$, but we have no examples of the particular trigram $w_{n-2}w_{n-1}w_n$ in the training corpus, we can instead estimate its probability by using the bigram probability $P(w_n \mid w_{n-1})$. If there are no examples of the bigram $w_{n-1}w_n$ in the training data either, we use the unigram probability $P(w_n)$. Formally, the trigram probability by mixing the three distributions is given by:

$$\hat{P}(w_n \mid w_{n-2}w_{n-1}) = \lambda_1 P(w_n) + \lambda_2 P(w_n \mid w_{n-1}) + \lambda_3 P(w_n \mid w_{n-1}w_{n-2})$$

where $\lambda_1 + \lambda_2 + \lambda_3 = 1$ (and each λ is non-negative), making the above equation a form of weighted averaging. We can similarly write expressions for other N-gram LMs.

But how do we choose the values of different λ_i? We choose these values by tuning them on a held out data i.e. the dev set, very similar to tuning hyperparameters for a machine learning model.

1.3.1 Coding Exercises (40 points).

Implement the following functions and classes in Part 3 (Smoothing and Interpolation) of the notebook.

- class `WordNGramLMWithAddKSmoothing` (20 points)
- class `WordNGramLMWithInterpolation` (20 points)

1.3.2 Write-Up Questions (14 points).

We recommend answering the write-up questions once you have finished the coding exercises in this section.

1. **Expression for N-gram language models with Laplace Smoothing (2 points).** Write expressions for the joint probability distribution $p(w_1 \cdots w_n)$ for unigram, trigram, 4-gram, and 5-gram LMs with Laplace smoothing.

2. **Effect of k on Perplexities (6 points).** Plot how the train and dev perplexities of bigram, trigram, 4-gram, and 5-gram LMs with Add-k smoothing from different values of k i.e. $\{1e{-}8, 1e{-}7, \cdots, 1e{-}1, 1\}$. You should have perplexity on the y-axis and k on the x axis. Use log-scaling for the x axis when plotting. Explain the trend that you see in 3 lines. Finally, report the best setup i.e. values of N and k which achieve the best dev perplexity.

3. **Effect of λ_is for Interpolation (6 points).** For bigram, trigram, and 4-gram models with interpolation, train models with different values of λ_is and evaluate perplexities on train and dev datasets. You should try at least 5 sets of values for each model. Report the λ_is values that you experiment with and train and dev perplexities for each setting and N-gram model.