Assignment 4: N-Gram Language Models

Instructor: Noah Smith *CSE P517 – Spring 2025*

Due at 11:59pm PT, May 19, 2025
50 points

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

You will submit both your **code** and **writeup** (as PDF) via Gradescope. 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 writeup. If you work on the assignment independently, please specify so, too.

Required Deliverables

- **Code Notebook**: The assignment is associated with a Jupyter notebook. (CSE447_Assignment4.ipynb) Please download the notebook as Jupyter notebook files (`.ipynb`) and submit them in Gradescope.
- **Write-up**: For written answers and open-ended reports, produce a single PDF for §1-3 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 and 5 of Jurafsky and Martin. We provide all the details necessary to solve the homework in this handout and the notebooks, so it is not required to read the chapter to solve the exercises. However, we recommend going through these chapters if you are confused about any concepts that are covered in the homework.

Acknowledgement

This assignment is designed 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 (50 Points)

In this assignment, you will implement and experiment with n-gram language models. N-gram language models are the simplest kind of language model. They make a simplifying assumption that the probability of a word in a sequence only depends on the past $n-1$ 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
- How to sample text from an n-gram language model
- How to implement Laplace smoothing
- How to implement interpolation

We will be working with Shakespeare plays data from Andrej Karpathy's blog post on recurrent neural networks.

Notebook: We have designed this part with the following Python notebook: CSE447_Assignment4.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:** Your report for §1 should be **no more than four pages**. However, you will most likely be able to answer all questions within three pages. 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.

1.1 Unigram Language Models (11 points)

We start by implementing unigram language models, the simplest variant of n-gram models – simply learn the distribution over words in the corpus. Recall from the lectures that for a text sequence of terms $w_1, w_2, \cdots, w_n$, 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 corresponds to calculating the relative 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.

Recall from lecture that to ensure the probabilities over sequences sum to one, you need to include a special "STOP" symbol at the end of every training sequence. This means that $C(\text{STOP})$ will be equal to the number of sequences in the training data, and some probability mass will be held for stopping (ending a sequence).

Evaluating Unigram Language Models using Perplexity. Now that we have trained our first language model, our next job is to evaluate how well it fits the training text and 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 $w_1 w_2 \cdots w_n$. Remember that the set sequence needs to include a stop symbol at the end of each sequence, and factor in its probability (and it also counts toward "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, if m is the number of sentences, we can write:

$$\text{perplexity}(W_{1:m}) = \exp\left(-\frac{\sum_{i=1}^{m} \log P(w_1^i w_2^i \cdots w_{n_i}^i)}{\sum_{i=1}^{m} n_i}\right)$$

where W_i is a sentence in the corpus with words $w_1^i, w_2^i, \ldots, w_{n_i}^i$ and n_i is the number of words in W_i. Note that $w_{n_i}^i$ will always be the stop symbol.

Note that, just like assuming words are independent is a mismatch with reality, sentences are not (in general) independent of each other. In future homework assignments, we will drop this assumption as we build more powerful models.

Sampling from a 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 the next token from the n-gram probability distribution given the previous words, i.e.,

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

For a unigram language model, the above equation simplifies to:

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

1.1.1 Coding Exercises (8 points).

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

- function `add_eos`
- function `train_word_unigram` (2 points)
- function `eval_ppl_word_unigram` (2 points)
- functions `replace_rare_words_wth_unks` and `eval_ppl_word_unigram_wth_unks` (2 points)
- function `sample_from_word_unigram` (2 points)

1.1.2 Write-Up Questions (3 points)

Check section 1 of CSEP517_Assignment4_Writeup.pdf for detail.

1.2 Longer N–Gram Language Models (12 Points)

We will now implement language models that make use of the preceding text to model the distribution of each word. 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

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

For example, for a bigram model ($N = 2$), the expression becomes

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

In general, the distribution of a token depends on the immediately preceding $N - 1$ tokens in the sequence.

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

$$P(w_k \mid w_{k-N-1}, \cdots, w_{k-1}) = \frac{C(w_{k-N-1} \cdots w_{k-1} w_k)}{\sum_{w \in \mathcal{V}} C(w_{k-N-1} \cdots w_{k-1} w)} = \frac{C(w_{k-N-1} \cdots w_{k-1} w_k)}{C(w_{k-N-1} \cdots w_{k-1})}$$

where $C(w_{k-N-1} \cdots w_{k-1} w)$ is the number of times the token sequence $w_{k-N-1} \cdots w_{k-1} w$ appears in the corpus, and $\mathcal{V}$ is the vocabulary of the n-gram model (fixed in advance). As before, w_k will sometimes be the stop symbol. For $N \geq 2$, we also need to include special start symbols in the preceding token context. One way to accomplish this is to apprend $N - 1$ start symbols before every sequence, before tallying the counts. For example, if we are building a trigram model, the training sequence

We will now implement models that use the preceding text to model the distribution of each word .

should be treated as

START START *We will now implement models that use the preceding text to model the distribution of each word .* STOP

and there are 19 different trigrams whose counts will be increased by this sequence; you don't need to model $p(\text{START} \mid \text{context})$. (We assume in this example that the period at the end is a separate token.)

1.2.1 Coding Exercises (10 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` (10 points)

1.2.2 Write-Up Questions (2 points).

Check section 2 of CSEP517_Assignment4_Writeup.pdf for detail.

1.3 Smoothing and Interpolation in N-Gram LMs (27 points)

One issue with using n-gram language models is that any finite training corpus is bound to miss some n-grams that will appear in the test set. If we use maximum likelihood estimation (as we hae done so far in this assignment), the model assigns 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. Research over the past thirty years has produced many moothing algorithms for n-gram language models For this assignment we will focus on (i) Laplace smoothing and (ii) interpolation.

Laplace and Add-λ Smoothing. The simplest smoothing algorithm is Laplace smoothing. It adds one to the count of each possible 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

$$P_{\text{Laplace}}(w_k \mid w_{k-1}) = \frac{C(w_{k-1}w_k) + 1}{\sum_{w \in \mathcal{V}}(C(w_{k-1}w_k) + 1)} = \frac{C(w_{k-1}w_k) + 1}{C(w_{k-1}) + |\mathcal{V}|}$$

The expressions are similar for other values of N.

Laplace smoothing is also called "add-one" smoothing. A generalization of Laplace smoothing is "add-λ" smoothing with $\lambda > 0$. The expression for the add-λ-smoothed distribution for the bigram language model is given by:

$$P_{\text{add-}\lambda}(w_k \mid w_{k-1}) = \frac{C(w_{k-1}w_k) + \lambda}{\sum_{w \in \mathcal{V}}(C(w_{k-1}w) + \lambda)} = \frac{C(w_{k-1}w_k) + \lambda}{C(w_{k-1}) + \lambda|\mathcal{V}|}$$

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 $P(w_n)$. A simple way to make sure that the trigram gets some probability is to interpolate between the three distributions:

$$\hat{p}(w_k \mid w_{k-2}w_{k-1}) = \lambda_1 p(w_k) + \lambda_2 p(w_k \mid w_{k-1}) + \lambda_3 p(w_k \mid w_{k-2}w_{k-1})$$

where $\lambda_1 + \lambda_2 + \lambda_3 = 1$ (and each λ_i is nonnegative). 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 development set, very similar to tuning hyperparameters for a machine learning model. (Tuning them on the *training* set, i.e., only on trigrams seen in the training set, would lead to an optimal value of $\lambda_3 = 1$, which defeats the purpose of smoothing. Tuning them on the *test* set would be cheating!)

1.3.1 Coding Exercises (20 points).

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

- class `WordNGramLMWithAddKSmoothing` (10 points)
- class `WordNGramLMWithInterpolation` (10 points)

1.3.2 Write-Up Questions (7 points).

Check section 3 of CSEP517_Assignment4_Writeup.pdf for detail.