Assignment 1: Logistic Regression and N-Gram Language Models

Instructor: Robert Minneker *CSE 447 / CSE 517 - Wi 26*

Due at 11:59pm PT, January 29, 2026
100 points for CSE 447 (+ 5 points extra credit) / 110 points for CSE 517, 15% towards the final grade

In this assignment, you will learn how to implement logistic regression for binary and multi-class classifiers from scratch. You will also learn about N-gram language models, how do we train them, evaluate their quality, and 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.

Required Deliverables

- **Code Notebook**: The project has been divided into two parts and both of them have their associated Jupyter notebooks. You need to submit the notebooks with your solutions for both of these parts. Please download all the notebooks as ipynb files (`.ipynb`) and submit them in Gradescope. On Google Colab you can do so by `File` → `Download` → `Download .ipynb`.

- **CSV Files With Predictions and Adversarial Examples**: As you will go through the notebook for §1 (Text Classification), you would be asked to make predictions on a test dataset and save them in a csv file. **Follow the exact names for these files as specified in the notebook i.e. -** `test_data_with_binary_predictions.csv` and `test_data_with_multiclass_predictions.csv`. For CSE 517, there is a question on generating adversarial examples and storing them in a csv file called `adversarial_examples.csv`. Upload all the three csv files (or 2 if you are CSE 447 and not attempting optional question) on gradescope along with your code.

- **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 two chapters 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 Text Classification Using Logistic Regression (50 pt for 447 and 60 pt for 517)

In this project, you will implement linear text classifiers for sentiment analysis. We will be working with Stanford Sentiment Tree (SST) Bank, which contains movie reviews with sentiment labels. We will consider both the binary version (only two labels positive and negative) as well as 5 label version (very negative, negative, neutral, positive, very positive). In particular you will learn:

- How to convert text inputs to features
- How to train logistic regression models from scratch
- How to evaluate classification models
- Interpreting trained linear models
- (Optional for 447) Using insights to model's decision making process to generate Adversarial examples.

Notebook: We have designed this part with the following Python notebook: CSE447a_Project1a.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 project 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. To submit your code, download your notebook as a Python file (`CSE447a_Project1a.py`).
2. **Files containing Predictions and Adversarial Examples** You should submit csv files for test data predictions and generated adversarial examples. More details in the notebook and below.
3. **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 Converting Text To Features (8 Points)

Typical ML models work on the data described using mathematical objects like vectors and matrices, which are often referred to as features. These features can be of different types depending upon the downstream application, like for building a classifier to predict whether to give credit to a customer we might consider features like their age, income, employement status etc. In the same way to build a classifier for textual data, we need a way to describe each text example in terms of numeric features which can then be fed to the classification algorithm of our choice.

Coding Exercises (8 points). Implement your code for the following classes in the notebook:

- `LinguisticVectorizer` (2 points)
- `BOWVectorizer` (3 points)
- `BOWVectorizerWNormalizer` (3 points)

1.2 Binary Logistic Regression (27 points for 447 and 37 points for 517)

We will now start building our first text classifier! We will start with Logistic Regression for binary classification i.e. where each input can be classified into one of the two labels. Consider an input $\boldsymbol{x} \in \mathbb{R}^d$ defining the feature vector and label $y \in \{0, 1\}$. Recall from the lectures that logistic regression has the following functional form:

$$\hat{y} = P(y = 1) = \sigma(\boldsymbol{w}^T \boldsymbol{x} + b)$$

where, σ is the sigmoid function:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

and the weight vector $\boldsymbol{w} \in \mathbb{R}^d$ and the bias term $b \in \mathbb{R}$ are the learnable parameters in the model.

Binary Cross Entropy Loss To train logistic regression model. we need to first define a loss function that measures the error between the model's prediction of the label ($\hat{y}$) and the ground truth label (y). For logistic regression with binary labels, we use binary cross entropy (BCE) loss. For a given prediction, ground truth pair, the BCE loss is given as:

$$L_{\text{BCE}}(\hat{y}, y) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$$

Plugging in the value of $\hat{y}$, we get:

$$L_{\text{BCE}}(\boldsymbol{w}, b \mid \boldsymbol{x}, y) = -[y \log \sigma(\boldsymbol{w}^T \boldsymbol{x} + b) + (1 - y) \log(1 - \sigma(\boldsymbol{w}^T \boldsymbol{x} + b))]$$

To compute loss over the entire dataset $\mathcal{D}$, we simply average the loss for all examples:

$$L_{\text{BCE}}(\boldsymbol{w}, b \mid \mathcal{D}) = \frac{1}{m} \sum_{i=1}^{m} L_{\text{BCE}}(\hat{y}_i, y_i)$$

where, m is the number of examples in the dataset.

Gradient Descent for Logistic Regression The next step is find the values of the weight vector and bias term that minimizes the binary cross entropy loss. One of the most commonly used optimization algorithms used in machine (and deep) learning is gradient descent. Recall from lectures that in gradient descent we iteratively update the parameters of a model in the opposite direction of the gradient of loss function w.r.t. to the parameters, i.e.,

$$\theta^{t+1} = \theta^t - \frac{\eta}{m} \nabla_\theta \sum_{i=1}^{m} L(f(x_i; \theta), y_i)$$

Note that for logistic regression, $L(f(x; \theta), y) = L_{\text{BCE}}(\hat{y}, y)$. θ here denotes the parameters of the model, which for us is simply the weights $\boldsymbol{w}$ and bias b. η is also called learning rate, which determines the strength of every the update (too low then the convergence will be slow and too high we might overshoot the local minima).

The gradient of the BCE loss w.r.t $\boldsymbol{w}$ is given by:

$$\nabla_w L_{\text{BCE}}(\hat{y}_i, y_i) = [\frac{\partial L_{\text{BCE}}(\hat{y}_i, y_i)}{\partial w_1}, \cdots, \frac{\partial L_{\text{BCE}}(\hat{y}_i, y_i)}{\partial w_d}]^T$$

$$\frac{\partial L_{\text{BCE}}(\hat{y}_i, y_i)}{\partial w_j} = -(y_i - \hat{y}_i) x_{ij}$$

Note that x_{ij} refers to the j^{th} feature of i^{th} input. Similarly, for the bias term we get:

$$\nabla_b L_{\mathrm{BCE}}(\hat{y}_i, y_i) = \frac{\partial L_{\mathrm{BCE}}(\hat{y}_i, y_i)}{\partial b} = -(y_i - \hat{y}_i)$$

Plugging these into our gradient descent equation we get:

$$w_j^{t+1} = w_j^t - \frac{\eta}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)x_{ij}$$

$$b^{t+1} = b^t - \frac{\eta}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)$$

Before we begin our implementation, there are two points to note. First you must have noticed that we sum over m examples in our update equation. These are the all the examples in our training data. In practice, it can get very expensive to compute gradients w.r.t all examples, specially when we are dealing with huge datasets (millions or even billions of examples) and deep neural networks. In such cases, it is common to use stochastic or mini batch gradient descent, where for each update we only use a small batch of training data to update the weights (we use a different batch for every update till we exaust all the training data). An extreme case of this is where we only use one example at a time to update the weights:

$$w_j^{t+1} = w_j^t - \eta(\hat{y}_i - y_i)x_{ij}$$

$$b^{t+1} = b^t - \eta(\hat{y}_i - y_i)$$

Second, we wrote the above equations in terms of scalar variables and their summations. In practice, it can be much more efficient to vectorize these equations and use matrix operations which can make use of parallel computation. The vectorized version of our update rule will look like:

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \frac{\eta}{m}\boldsymbol{X}^T(\hat{\boldsymbol{y}} - \boldsymbol{y})$$

$$b^{t+1} = b^t - \frac{\eta}{m}\mathbf{1}_d^T(\hat{\boldsymbol{y}} - \boldsymbol{y})$$

Here, $X \in \mathbb{R}^{m\times d}$ is the input matrix where each row is a feature vector for an input example. Similarly, $\boldsymbol{y} \in \{0,1\}^m$ is the vector containing labels for each example and $\hat{\boldsymbol{y}} \in \mathbb{R}^m$ is the vector of predicted labels. $\mathbf{1}_d \in 1^d$ is a vector of all ones.

1.2.1 Coding Exercises (20 points).

Implement the following classes and functions in the notebook:

- class `LogisticRegression` (2 points)
- function `bce_loss` (1 points)
- function `gradient_descent_update_vanilla` (2 points)
- function `gradient_descent_update_vectorized` (2 points)
- function `train_logistic_regression` (5 points)
- functions `get_accuracy`, `get_precision`, `get_recall`, and `get_f1_score` (2 points)
- function `evaluate_logistic_regression` (3 points)
- function `interpret_logistic_regression` (3 points)

1.2.2 Write-Up Questions (7 points + 5 points extra credits for 447 and 17 points for 517).

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

1. **Difference between Training and Dev Losses (1 points)** Training logistic regression model with BOW features result in a very low train loss but a high dev loss. Training with linguistic features we had similar loss values for both train and dev sets (albeit higher than what we get with BOW features). Can you think of reasons why there is a big gap between train and dev losses when we use BOW features but not the case for linguistic features? Answer in no more than 2-3 lines.

2. **Ablations on Normalization Methods (3 points)** The BOW model was trained by applying 4 normalization techniques on the data before transforming the text into vector features. Whenever we propose a new model for solving an NLP task, it is important to understand the role each decision had to play on the final model performance. Try disabling one normalization technique while keeping the other 3 enabled (e.g. setting `lower_case=False`, while keeping `replace_rare_words_wth_unks`, `remove_punctuation`, and `remove_stopwords` as True) and record the evaluation metrics. You should report the results in a table with following format:

Model	**Dev Accuracy**	**Dev Precision**	**Dev Recall**	**Dev F1-score**
Logistic Regression BOW				
Logistic Regression BOW - `lower_case=True`				
Logistic Regression BOW - `replace_rare_words_wth_unks=True`				
Logistic Regression BOW - `remove_punctuation=True`				
Logistic Regression BOW - `remove_stopwords=True`				

Table 1: Comparing the effect of different normalization schemes on model performance.

3. **Hyperparameter Tuning (3 points).** Tuning hyperparameters is an important part of building any Machine Learning model. For our logistic regression model, we have the following hyperparameters: learning rate, number of epochs, batch size, and normalization methods. You can either perform a grid search on the hyperparameters or a random search (Read more here). Report the hyperparameter values that you try, the search method, and the number of trials, along with the best performing hyperparameter setting based on the F1-score on dev data (also report the score).

   Along with your writeup, also provide predictions on the test set (we only provide inputs for the test data), which we will use to evaluate your submission. You can load the test data in your notebook by running the following command:

   ```
   test_df = pd.read_csv(f"{data_dir}/sst_test_release.csv")
   ```

   Add a column called `pred_label` in the test data frame above and fill it with the predictions from your best model. After you add your predictions to the dataframe, save the dataframe as a CSV file, **strictly use the name of the file as** `test_data_with_binary_predictions.csv` and submit it with your code.

   ```
   test_df.to_csv("test_data_with_binary_predictions.csv")
   ```

   You don't need to perform very exhaustive hyperparameter tuning. Our objective is for you to familiarize yourself with how to set hyperparameters for training ML models. Your submission will not be graded based on whether you find the best set of hyperparameters. We will accept all submissions with better test accuracy than the one with the default hyperparameters.

4. **Adversarial Examples (10 points for CSE 517 and bonus 5 points for CSE 447)**. Note that this question is only madatory for CSE 517 students. For CSE 447 students it is optional, but a successful solution to the problem will fetch 5 points extra credit on this homework.

Interpreting a model reveals insights into its decision-making process. However, it also opens up the model to adversarial attacks. Adversarial examples are inputs specifically designed to fool the model into making incorrect predictions. For example, in the context of sentiment analysis, an adversarial example could be a sentence that is clearly positive, but the model predicts it as negative.

By looking at the top words contributing to positive and negative sentiment, can you create a dataset of adversarial examples that would fool the model you trained above? Instead of handcrafting each example, you can use a template-based approach, where you create templates for positive examples that trick the model into labeling them as negative, and similarly for negative examples. For example, you can create a template like "I thought the movie was [word1] and [word2]" and then fill in the words to create adversarial examples. You can create multiple templates and fill in different words to create a dataset of adversarial examples. We expect you to create at least 100 adversarial examples, with at least 50 examples for each class. You can use the `interpret_logistic_regression` function to get the top words contributing to positive and negative sentiment. You can refer to Ribeiro et al. (2020) to learn about templating test examples for behavioral testing of models.

Note that such types of adversarial attacks are *White Box* attacks, where the attacker has full knowledge of the model. In practice, adversarial attacks can also be *Black Box*, where the attacker does not have access to the model's parameters. For other examples of adversarial attacks on NLP models, you can check Wallace et al. 2019 and Nasr et al. 2023.

What you need to submit:

- Describe your approach for creating adversarial examples in 4-5 lines. Share the templates and the words filling those templates that you used to create the adversarial examples.
- Evaluate the performance of the model on the adversarial examples you created. Report accuracy on the adversarial examples.
- Share the adversarial examples you created in a CSV file. The CSV file should have two columns: `text` and `label`. The `text` column should contain the adversarial examples, and the `label` column should contain the true label of the adversarial examples. Save the CSV file as `adversarial_examples.csv` and upload it with your code.

How your submission will be graded:

- We will run the logistic regression model trained with default hyperparameters on the adversarial examples you created. The model should perform worse than chance on these examples, i.e., should have accuracy way less than 50% (we will accept all submissions with accuracies below 40%).
- We will also check the quality of your adversarial examples, i.e., whether the true label that you assigned is indeed the correct label for the adversarial example. For example, creating an adversarial example like "I thought the movie was good and bad" and assigning it a positive label would be incorrect. The examples should be such that they are clearly positive or negative, but the model is tricked into predicting the opposite label.

1.3 Multinomial Logistic Regression (15 points)

We will now move to the multi-class case i.e. where the text is to be classified into more than 2 classes. We will be working with the 5-label version of the SST dataset where the labels are: very negative, negative, neutral, positive, very positive.

To extend logistic regression to multi-class classification, we can use the softmax function. For a vector $\boldsymbol{z} = [z_1, z_2, \cdots, z_K]$ of K arbitrary real numbers, the softmax function maps them to a probability distribution, with each value between 0 and 1 and summing to 1. The softmax function is defined as:

$$\texttt{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)} \quad 1 \le i \le K$$

In multinomial logistic regression, we consider a weight vector $\boldsymbol{w}_k$ for each of the K classes, take the dot product with input features (also adding a bias term unique for each class) to obtain scores or *logits* for each of the classes. We then apply the softmax function to get the probability distribution over the classes. Formally, this is given by:

$$\hat{y}_k = P(y_k = 1 \mid \boldsymbol{x}) = \frac{\exp(\boldsymbol{w}_k^T \boldsymbol{x} + b_k)}{\sum_{j=1}^{K} \exp(\boldsymbol{w}_j^T \boldsymbol{x} + b_j)}$$

where $P(y_k = 1 \mid \boldsymbol{x})$ is the probability of the input $\boldsymbol{x}$ belonging to class k, $\boldsymbol{w}_k$ is the weight vector for class k, and b_k is the bias term for class k.

Note: In practice, we often use a trick to make the computation of softmax more numerically stable. We subtract the maximum value from the logits before applying softmax. This does not change the output of softmax but can prevent numerical overflow. The softmax function with this trick is given by:

$$\texttt{softmax}(z_i) = \frac{\exp(z_i - \max(\boldsymbol{z}))}{\sum_{j=1}^{K} \exp(z_j - \max(\boldsymbol{z}))} \quad 1 \le i \le K$$

Cross Entropy Loss for Multinomial Logistic Regression. The loss function for multinomial logistic regression is the cross entropy loss. For a given input $\boldsymbol{x}$ and ground truth label $\boldsymbol{y}$, the cross entropy loss is given by:

$$L_{\text{CE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

$$L_{\text{CE}}(\boldsymbol{w}, \boldsymbol{b} \mid \boldsymbol{x}, \boldsymbol{y}) = -\sum_{k=1}^{K} y_k \log \frac{\exp(\boldsymbol{w}_k^T \boldsymbol{x} + b_k)}{\sum_{j=1}^{K} \exp(\boldsymbol{w}_j^T \boldsymbol{x} + b_j)}$$

where $\hat{\boldsymbol{y}}$ is the predicted probability distribution over the classes and $\boldsymbol{y}$ is the one-hot encoded ground truth label, such that $y_k = 1$ if the input belongs to class k and 0 otherwise. We can compute the loss over the entire dataset $\mathcal{D}$ by averaging the loss over all examples:

$$L_{\text{CE}}(\boldsymbol{w}, \boldsymbol{b} \mid \mathcal{D}) = \frac{1}{m} \sum_{i=1}^{m} L_{\text{CE}}(\hat{\boldsymbol{y}}_{\boldsymbol{i}}, \boldsymbol{y}_{\boldsymbol{i}})$$

Gradient Descent for Multinomial Logistic Regression The gradient of the cross entropy loss w.r.t the weights and biases can be computed as follows:

$$\nabla_{\boldsymbol{w}_k} L_{\text{CE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = (\hat{\boldsymbol{y}} - \boldsymbol{y})^T \boldsymbol{x}$$

$$\nabla_{b_k} L_{\text{CE}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \hat{\boldsymbol{y}} - \boldsymbol{y}$$

The gradient descent update rule for the weights and biases is given by:

$$\boldsymbol{w}_k^{t+1} = \boldsymbol{w}_k^t - \frac{\eta}{m} \sum_{i=1}^{m} (\hat{\boldsymbol{y}}_{\boldsymbol{i}} - \boldsymbol{y}_{\boldsymbol{i}})^T \boldsymbol{x}_{\boldsymbol{i}}$$

$$b_k^{t+1} = b_k^t - \frac{\eta}{m} \sum_{i=1}^{m} (\hat{\boldsymbol{y}}_{\boldsymbol{i}} - \boldsymbol{y}_{\boldsymbol{i}})$$

1.3.1 Coding Exercises (11 points).

Implement the following functions and classes in Part 3 (Multinomial Logistic Regression) of the notebook:

- class `MultinomialLogisticRegression` (2 points)
- function `ce_loss` (1 points)
- function `gradient_descent_update_multiclass` (3 points)
- function `train_multinomial_logistic_regression` (2 points)
- functions `get_precision_multiclass`, `get_recall_muticlass`, `get_f1_score_multiclass`, and `get_confusion_matrix` (2 points)
- function `evaluate_multiclass_logistic_regression` (1 points)

1.3.2 Write-Up Questions (4 points)

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

1. **Numerical Stability of Softmax (1 points)** Can you show why subtracting the maximum value from the logits before applying softmax doesn't change the output of softmax? Show detailed steps in your answer. Also explain in not more than 3 lines, the cause of numerical overflow in softmax and how does this trick help in preventing it.

2. **Hyperparameter Tuning for Multinomial Logistic Regression (3 points)** Similar to the binary case, you will perform hyperparameter tuning for the multinomial logistic regression model. You can either perform a grid search on the hyperparameters or a random search. Report the hyperparameter values that you try, the search method, and the number of trials, along with the best performing hyperparameter setting. Along with your writeup, also provide predictions on the test set (we only provide inputs for the test data), which we will use to evaluate your submission.

   Like before, add a column called `pred_label` in the test data frame above and fill it with the predictions from your best model. After you add your predictions to the dataframe, save the dataframe as a CSV file (strictly use the name of the file as `test_data_with_multiclass_predictions.csv`) and upload it with your code.

   ```
   test_df.to_csv("test_data_with_multiclass_predictions.csv")
   ```

2 N-Gram Language Models (50 Points for both CSE 447 and CSE 517)

In this project, 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 project, 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: CSE447a_Project1b.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 project 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. To submit your code, download your notebook as a Python file (`CSE447b_Project1b.py`).
2. **Write-up:** Your report for §2 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.

2.1 Unigram Language Models (11 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:

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

2.1.1 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) [**Note: We will grade this function manually**]

2.1.2 Write-Up Questions (3 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 (2 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 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 (1 Point)** 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.

2.2 N(>1)-Gram Word-Level Language Models (12 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 §2.1 and we do not repeat them here.

2.2.1 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)

2.2.2 Write-Up Questions (2 points).

1. **Probability distribution of sentence modeled by an N-gram LM (2 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.

2.3 Smoothing and Interpolation in N-Gram LMs (27 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 project 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.

2.3.1 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)

2.3.2 Write-Up Questions (7 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 (1 points).** Write expressions for the joint proabbiility 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 (3 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 (3 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.