Homework 3

Start Assignment

- Due Monday by 11:59pm
- Points 56
- Submitting a file upload
- File Types pdf, py, and txt
- Available Feb 26 at 4:30pm - Apr 30 at 11:59pm

Language modeling is the task of predicting the next word in a sequence given the previous words. In this assignment, we will focus on the related problem of predicting the next *character* in a sequence given the previous characters. You will build character-level n-gram language models as well as train an LLM (GPT-2) to do character-level language modeling using Hugging Face. You will generate text from models you create, as well as use perplexity to measure the fit of various language models on test data related and unrelated to the training data.

Learning objectives (https://michaelmilleryoder.github.io/cs2731_fall2024/hw3#le objectives)

After completing this assignment, students will be able to:

- Understand how to compute n-gram language model probabilities using maximum likelihood estimation.
- Use n-gram and transformer-based language models to probabilistically generate texts.
- Intuitively understanding of how perplexity will estimate language model performance on unseen texts.
- Gain familiarity with training LLMs using Hugging Face

Data

Download the following datasets for this assignment:

- **Shakespeare training data (https://drive.google.com/file/d/15Yq_WS6tVNkSP2ux0lHorzj1pdJ7FqrT/view?usp=sharing)**
- **Test data (https://drive.google.com/file/d/14kF_-Pk12hXybS8lu1H300plonCP6aCH/view?usp=sharing)** of *New York Times* articles and several of Shakespeare's sonnets

Part 1: Train character-level n-gram language models

In this section, you will fill in the following skeleton Python script:

- **N-gram skeleton script (https://drive.google.com/file/d/1HIAF4b57msyytlW-vAqs6uJZDqezTIQO/view?usp=sharing)**

1.1 Extract character n-grams (https://michaelmilleryoder.github.io/cs2731_fall2024/hw3#extract-character-n-grams)

First, fill out the `ngrams(c, text)` function that produces a list of all n-grams of that use `c` elements of context from the input text. Each n-gram should consist of a 2-element tuple `(context, char)`, where the context is itself a `c`-length string comprised of the `c` characters preceding the current character. If `c == 1`, then produce bigrams, if `c == 2`, trigrams. The sentence should be padded with `c` `~` characters at the beginning (we've provided you with `start_pad(c)` for this purpose). If `c == 0`, all contexts should be empty strings. You may assume that `c ≥ 0`. You are allowed to use any resources or packages to extract the character ngrams from text, such as scikit-learn or NLTK. Here is some example output from such a function:

```
>>> ngrams(1, 'abc')
[('~', 'a'), ('a', 'b'), ('b', 'c')]

>>> ngrams(2, 'abc')
[('~~', 'a'), ('~a', 'b'), ('ab', 'c')]
```

We've also given you the function `create_ngram_model(model_class, path, c, k)` that will create and return an n-gram model trained on the entire file path provided.

1.2 Build n-gram language models (https://michaelmilleryoder.github.io/cs2731_fall2024/hw3#build-n-gram-language-models)

In this section, you will build a simple n-gram language model that can be used to generate random text resembling a source document.

In the `NgramModel class`, write an initialization method `__init__(self, c, k)` which stores the context length `c` of the model and initializes any necessary internal variables. Then write a method `get_vocab(self)` that returns the vocab (this is the set of all characters used by this model).

Write a method `update(self, text)` which computes the n-grams for the input sentence and updates the internal counts. Also, write a method `prob(self, context, char)` which accepts a `c`-length string representing a context and a character, and returns the probability of that character occurring, given the preceding context. Characters that have never been seen before in a certain context would be assigned a 0 probability. If you encounter a novel context (one that has never been seen before in training data), the probability of any given character should be $1/V$ where V is the size of the vocabulary. See Chapter 3 of the Jurafsky and Martin textbook and Equation 3.12 for calculating probabilities based on observed counts. You may not use any package to directly train/compute language model probabilities; that portion of the program should be from scratch.

```
>>> m = NgramModel(1, 0)
>>> m.update('abab')
>>> m.get_vocab()
{'a', 'b'}
>>> m.update('abcd')
>>> m.get_vocab()
{'a', 'b', 'c', 'd'}
>>> m.prob('a', 'b')
1.0
>>> m.prob('~', 'c')
0.0
>>> m.prob('b', 'c')
0.5
```

Write a method `random_char(self, context)` which returns a random character according to the probability distribution determined by the given context. Just like the `prob` function, in a novel context assign a probability of any given character $1/V$, where V is the size of the vocabulary.

Here is some example output. Even with setting the random seed, **your output does not need to perfectly match the example output** as there are multiple functions that can perform this task.

```
    >>> m = NgramModel(0, 0)
    >>> m.update('abab')
    >>> m.update('abcd')
    >>> random.seed(1)
    >>> [m.random_char('') for i in range(10)]
    ['a', 'c', 'c', 'a', 'b', 'b', 'b', 'c', 'a', 'a']
```

In the NgramModel class, write a method `random_text(self, length)` which returns a string of characters chosen at random using the `random_char(self, context)` method. Your starting context should always be `c` `~` characters, and the context should be updated as characters are generated. If `c == 0`, your context should always be the empty string. You should continue generating characters until you've produced the specified number of random characters, then return the full string.

Here is some example output. Even with setting the random seed, **your output does not need to perfectly match the example output** as there are multiple functions that can perform this task.

```
    >>> m = NgramModel(1, 0)
    >>> m.update('abab')
    >>> m.update('abcd')
    >>> random.seed(1)
    >>> m.random_text(10)
    abcdbabcda
```

1.3 Generating Shakespeare with character-level n-gram language models [(https://michaelmilleryoder.github.io/cs2731_fall2024/hw3#generating-shakespeare-with-character-level-n-gram-language-models)](https://michaelmilleryoder.github.io/cs2731_fall2024/hw3#generating-shakespeare-with-character-level-n-gram-language-models)

Now you can train a language model using the training corpus of Shakespeare. Afterward, try generating some Shakespeare with different order character n-gram models. For example, you can try using different n by running the following commands:

```
>>> m = create_ngram_model(NgramModel, 'shakespeare_input.txt', 2)
>>> m.random_text(250)

>>> m = create_ngram_model(NgramModel, 'shakespeare_input.txt', 3)
>>> m.random_text(250)

>>> m = create_ngram_model(NgramModel, 'shakespeare_input.txt', 4)
>>> m.random_text(250)

>>> m = create_ngram_model(NgramModel, 'shakespeare_input.txt', 7)
>>> m.random_text(250)
```

You may make any additional assumptions and design decisions, but state them in your report (see below). For example, some design choices that could be made are how you want to handle uppercase and lowercase letters or how you want to handle digits. The choice made is up to you, we only require that you detail these decisions in your report and consider any implications of them in your results. There is no wrong choice here, and these decisions are typically made by NLP researchers when pre-processing data.

1.4 Calculate perplexity of test documents [(https://michaelmilleryoder.github.io/cs2731_fall2024/hw3#calculate-perplexity-of-test-documents)](https://michaelmilleryoder.github.io/cs2731_fall2024/hw3#calculate-perplexity-of-test-documents)

Using the `perplexity` method, calculate the perplexity of each test document. For each file in the test data (`nytimes_article.txt` and `shakespeare_sonnets.txt`), calculate the perplexity for each **non-blank** line and the average across all lines in the document. Do this for trigram, 4-gram and 7-gram character-level language models trained on Shakespeare plays (`shakespeare_input.txt`).

Deliverables for part 1 (https://michaelmilleryoder.github.io/cs2731_fall2024/hw3#d for-part-1)

In your report, include:

1. A description of how you wrote your program, including all assumptions and design decisions
2. What do you notice about the short passages you've generated from n-gram models with different *n*? Are they as good as **1000 monkeys working at 1000 typewriters (https://www.youtube.com/watch?v=no_elVGGgW8)**

   (https://www.youtube.com/watch?v=no_elVGGgW8)
   ? Are there patterns in what models generate first? Report some of your generated text and discuss.
3. Perplexity values for trigram, 4-gram, and 7-gram character-level language models on each test file (*New York Times* article and Shakespeare sonnets).
4. What does your perplexity indicate across different test documents? What does a comparison of different *n* in the n-grams in terms of perplexity tell you? Which performs best? Why do you think your models performed the way they did?

Part 2: Train a GPT-2 character-level language model

In this section, you will train an LLM-based model (GPT-2) on the same task: character-level language modeling. You will use the Hugging Face set of packages. You will then generate from your trained LLM and compare the output against the character n-gram models.

To do so, run each cell and fill out the missing code sections by copying the following Google Colab notebook (**hw3_char_llm_skeleton.ipynb (https://colab.research.google.com/drive/10tJGMSLiXK2W30jknulvprC06V5CQ_el?usp=sharing)**) to your own local space.

Note that training the model will take 30 minutes minimum, so make sure you schedule enough time for this part.

If you run out of Google Colab resources, you will have to wait until later to run the notebook. Email Lorraine to inform her of this issue and ask for extra time if needed.

Deliverables for part 2 [(https://michaelmilleryoder.github.io/cs2731_fall2024/hw3#d for-part-2)](https://michaelmilleryoder.github.io/cs2731_fall2024/hw3#deliverables-for-part-2)

In your report, include

1. What settings you used for sampling. Did you experiment with different settings, such as the *k* in top-*k* sampling?
2. A comparison of the generated output between character n-gram approaches and the GPT-2 version. Does one have more understandable words than the other? Are there any other differences you notice? Please point to specific examples.

Submission

Please submit the following items on Canvas:

- Your report with results and answers to questions in Part 1 and Part 2, named `report_{your pitt email id}_hw3.pdf`. No need to include @pitt.edu, just use the email ID before that part. For example: `report_xianglli_hw3.pdf`.
- The code of your program for part 1
- A link to your Google Colab file for part 2
- A `README.txt` file explaining
    - the computing environment you used; what programming language and version you used; what packages did you use in case we replicate your experiments (a `requirements.txt` file for setting up the environment may be useful if there are many packages).
    - any additional resources, references, or web pages you've consulted
    - any person with whom you've discussed the assignment and describe the nature of your discussions
    - any generative AI tool used, and how it was used
    - any unresolved issues or problems

Grading

This homework assignment is worth 56 points. The grading rubric will be posted on Canvas.

Acknowledgments (https://michaelmilleryoder.github.io/cs2731_fall2024/hw3#a

Part 1 of this assignment is based on a homework assignment by Prof. Diane Litman and Prof. Mark Yatskar. Part 2 of this assignment is from Prof. Michael Yoder. **Shakespeare data**

Homework 3 rubric

Criteria	Ratings		Pts
Part 1: Submits everything needed: code, README file, report	**5 pts** **Full Marks**	**0 pts** **No Marks**	5 pts
Part 1: Describes how program was written and any assumptions and design decisions	**5 pts** **Full Marks**	**0 pts** **No Marks**	5 pts
Part 1: Reports generated text for at least 2 character n-gram models	**8 pts** **Full Marks**	**0 pts** **No Marks**	8 pts
Part 1: Discusses any patterns noticed about generated text across different models	**7 pts** **Full Marks**	**0 pts** **No Marks**	7 pts
Part 1: Provides perplexity values for trigram, 4-gram and 7-gram models	**8 pts** **Full Marks**	**0 pts** **No Marks**	8 pts
Part 1: Discusses differences in perplexity values across different n	**7 pts** **Full Marks**	**0 pts** **No Marks**	7 pts
Part 2: Provides link to a Colab with cells filled out	**6 pts** **Full Marks**	**0 pts** **No Marks**	6 pts
Part 2: Provides settings used for sampling	**3 pts** **Full Marks**	**0 pts** **No Marks**	3 pts
Part 2: Provides examples of GPT-2 output	**4 pts** **Full Marks**	**0 pts** **No Marks**	4 pts

Criteria	Ratings		Pts
Part 2: Compares text generated by character n-gram and GPT-2 models	**3 pts** **Full Marks**	**0 pts** **No Marks**	3 pts
			Total Points: 56