N-gram Language Model Instructions

LING/CSE 472

Spring 2021

1 Overview

Creating an n-gram language model is an impressive, challenging, but manageable term project for anyone who is new to programming and implementing algorithms. For this project, you will write a program that trains and evaluates three different n-gram language models on provided datasets. In particular, you will:

- Implement and train a unigram, bigram, and trigram language model
- Handle zero counts by using Laplace smoothing and `<UNK>` tokens
- Validate each model's performance on the development set as you develop your program
- Implement *optional* extensions to your program
- Evaluate each model's performance on the held-out test set after completing the *entire* program
- Produce a *brief* write-up (1-2 pages, single-spaced) that:
  - Describes the n-gram language models you implemented
  - Report your models' perplexity on the training, development, and sets in a small table

2 Specifications

This section details the algorithmic specifications of your n-gram language models.

2.1 System output

For each model, you will implement a `LanguageModel()` class. This class should minimally include two core methods, `train()` and `score()`, but may also include any number of helper functions:

- *Important*: All probabilities should be calculated in $\mathbf{log_2}$ space to avoid underflow.
- The `train()` method should read in the training corpus, pre-process the sentences (i.e., split the text into sentences and tokens), then calculate the probabilities for each n-gram observed during training.

– **Output**: For each observed n-gram, print the n-gram along with its *logged* maximum likelihood estimate, rounded to the third decimal place. Each n-gram∼probability pair should be printed on its own line, sorted from highest to lowest probability (then in alphabetical order). For instance, the n-gram∼probability pairs for the trigram model should appear in the following format:

```
# <w1> <w2> <w3> <log2(P(w3|w1 w2))>
cookies and cream -1.286
With great power -4.573
The quick brown -7.861
...
```

- The `score()` method should read in a validation corpus (i.e., the development or test set), pre-process the sentences, then calculate the perplexity of the language model on the corpus.

  – **Output**: For each sentence s_i in the validation corpus, print the sentence, followed by the *logged* probability of the sentence. The sentences should be printed in the order that they appear in the corpus, with one sentence∼probability pair per line. After the last sentence, print the model's perplexity, rounded to the third decimal place. For example:

```
With great power comes great responsibility . -8.314
The quick brown brown fox jumped over the lazy dog . -15.019
I love cookies and cream ! -24.567
...
Perplexity = -18.765
```

2.2 Perplexity

Recall that the perplexity PP of a language model on a dataset W is the inverse probability of the dataset, normalized by the total number of words N in the dataset:

$$\begin{aligned} PP(W) &= P(w_1 w_2 ... w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{P(w_1 w_2 ... w_N)}} \end{aligned}$$

Since we represent each n-gram probability in $\log_2$ space, we can calculate perplexity accordingly:

$$PP(W) = 2^H$$

where

$$\begin{aligned} H &= -\frac{1}{N} \log_2 P(w_1 w_2 ... w_N) \\ &= -\frac{1}{N} \sum_{i=1}^{M} \log_2 P(s_i) \end{aligned}$$

and M is the total number of sentences in the dataset.

2.3 Start and stop tokens

For the bigram and trigram models, you will need to introduce one to two start tokens (`<s>`) at the start of each sentence and one stop token (`</s>`) at the end of each sentence. Given a sentence s_i of length n tokens (i.e., $s_i = w_1\ w_2\ w_3\ ...\ w_n$), you can approximate $P(s_i)$ like so:

$$P_{\text{bigram}}(s_i) = P(w_1|\texttt{<s>}) \times P(w_2|\ w_1) \times P(w_3|w_2) \times ... \times\ P(w_N|w_{n-1}) \times P(\texttt{</s>}|\ w_n)$$

$$P_{\text{trigram}}(s_i) = P(w_1|\texttt{<s>}_1\ \texttt{<s>}_2) \times P(w_2|\texttt{<s>}_2\ w_1) \times P(w_3|w_1\ w_2) \times ... \times$$
$$P(w_n|w_{n-2}\ w_{n-1}) \times P(\texttt{</s>}|w_{n-1}\ w_n)$$

2.4 Laplace smoothing

To handle *unseen* n-grams, implement Laplace ("add-1") smoothing by pretending you observed each n-gram one additional time in the training corpus:

- For the unigram model, pretend you saw each *unigram* one additional time, where N is the numbers of words in the training corpus and $|V|$ is the size of the vocabulary:

$$P_{\text{unigram}}(w_i) = \frac{\text{count}(w_i) + 1}{N + |V|}$$

- For the bigram model, pretend you saw each *bigram* one additional time:

$$P_{\text{bigram}}(w_i|w_{i-1}) = \frac{\text{count}(w_{i-1}\ w_i) + 1}{\text{count}(w_{i-1}) + |V|}$$

- For the trigram model, pretend you saw each *trigram* one additional time:

$$P_{\text{trigram}}(w_i|w_{i-2}\ w_{i-1}) = \frac{\text{count}(w_{i-2}\ w_{i-1}\ w_i) + 1}{\text{count}(w_{i-2}\ w_{i-1}) + |V|}$$

2.5 UNK-ing

Please handle *out-of-vocabulary* (OOV) words in your program accordingly:

- In `train()`, convert any token that appears only once in the training set to the special `<UNK>` token, then proceed with training probabilities as normal.
- In `score()`, convert any OOV words to `<UNK>` before calculating sentence probabilities.

2.6 Vocabulary notes

- For the bigram and trigram models, include the special tokens `</s>` and `<UNK>` in the vocabulary. (*Food for thought*: Why do you think we exclude `<s>` from V?)
- UNK-ing low frequency words and adding special tokens will change the size of V, which will in turn impact your Laplace-smoothed probabilities! When smoothing the n-gram probabilities, $|V|$ should reflect the size the of the vocabulary *after* taking these steps.

3 Starter code

The starter code for this project can be found on Canvas in a zipped file called *project.zip*. This file gives you the following contents:

```
project/
├── data/
│   ├── austen-train.txt
│   ├── austen-dev.txt
│   └── austen-test.txt
├── main.py
├── unigram.py
├── bigram.py
└── trigram.py
```

Please implement your unigram, bigram, and trigram language models in *unigram.py*, *bigram.py*, and *trigram.py* respectively. You are welcome to modify any of the Python files as you see fit—or even write the program entirely from scratch! To run your language models using the starter code, call *main.py*:

- To train a model, pass the training set to *main.py* in the command line. This will invoke the model's `train()` function:

```
python main.py data/austen-train.text
```

- By default, *main.py* will train a unigram model. To train a bigram or trigram model, use the `-n <integer>` construction to indicate the order of the n-gram:

```
python main.py data/austen-train.text -n 1    # unigram (default)
python main.py data/austen-train.text -n 2    # bigram
python main.py data/austen-train.text -n 3    # trigram
```

- To evaluate a language model on a validation set (i.e., the development or test set), use the `-t <corpus_fn>` construction. This will invoke the model's `score()` function. For example, to evaluate a trigram model on the development set, run the following command:

```
python main.py data/austen-train.text -n 3 -t data/austen-dev.text
```

4 Datasets

We have provided you with the following training, development, and test sets:

- `data/austen-train.txt`
- `data/austen-dev.txt`
- `data/austen-test.txt`

The datasets combine three books by Jane Austen (*Emma*, *Persuasion*, and *Sense and Sensibility*), which we obtained through NLTK's selection of Project Gutenberg texts. We shuffled the sentences across the books and assigned 80% of the sentences to the training set, 10% to the development set, and 10% to the test set.

Each dataset is formatted with one sentence per line, where each sentence comes pre-tokenized, with whitespace delimiting token boundaries. You may rely on these token boundaries during pre-processing. In your models, please do not lowercase the tokens (e.g., do treat *The* and *the* as separate word types).

5 Tips

- Start with the simplest model first: Begin with unigrams, end with trigrams.
- It can help to create mini toy datasets for quick, iterative development!
- While you shouldn't need to install any fancy packages for this project, you will need the `log2()` function from Python's `math` module to calculate probabilities in $\log_2$ space.
- Before programming, map out how you want your code to run, breaking each problem down into bite-sized pieces. You can even do this in your code by defining helper functions and using comments to outline what steps you need to take in each function. For instance, you might map out a function like the following:

```
def load_data(corpus_filename):
    pass

    # step 1: read in the data from the file as a chunk of text

    # step 2: convert the text into a list of sentences, stripping
    # newline characters (\n)

    # step 3: represent each sentence as a list of tokens

    # step 4: insert start and stop tokens

    # step 5: return the list of lists
```

- The most straightforward way to store counts and probabilities is using nested dictionaries. For instance, a trigram dictionary might look like:

```
D = {
    "cream": {
        "yummy ice": 2,
        "cookies and": 3,
        "coffee with": 4,
        ...
        },
    ...
    }
```

6 Optional extensions

Time and interest permitting, we encourage you to extend your program! Below are some ideas for extensions; please note that all of these options will require you to modify *main.py*. You are also welcome to discuss different extension ideas with us.

- **Recommended**: Implement a way to save and load trained probabilities. This will allow you to skip re-training a model each time you run the program!
  - *Tip*: One way to do this is to redirect the output from `train()` to a new file (e.g., by calling `python main.py data/austen-train.text > my-unigram-model.txt`). You can then read in the n-grams and their logged probabilities to populate a nested dictionary. Alternatively, you can save and load a dictionary directly using Python's `json` library.
- Add a method to `LanguageModel()` that *generates* sentences using the Shannon Visualization Method.
- In a file called *trigram-backoff.py*, adapt your trigram language model to use Stupid Backoff smoothing.
- In a file called *trigram-interp.py*, adapt your trigram language model to use linear interpolation. The `LanguageModel()` class should accept the values for $\lambda 1$, $\lambda 2$, and $\lambda 3$ as arguments.
- Create a new `LanguageModel()` class in a file called *ngram.py* that accepts an argument indicating the order of the n-gram. For example, when the argument is 3, the program would train a trigram language model; when the argument is 4, it would train a 4-gram language model; et cetera. If you implemented any additional smoothing methods, you may also expand this class to accept arguments that determine which smoothing method is used.
- **Advanced**: Vectorize your math using the NumPy package. Storing and computing counts and probabilities using vectors/matrices can significantly reduce the runtime of your program, compared to using nested dictionaries.

Alternatively, if you would like to delve into error analysis, try evaluating your *Jane Austen* language models on new corpora to see how they perform on data written by different authors.