Programming for Computational Linguistics 2018/2019

Final Project — n-gram Language Model
Due 3.3.19

1 Introduction

Your task for the final project will be to **design, implement, and document an n-gram language model in python**. This language model will be able to learn from provided training text, and then generate new text that is statistically similar to the training text. In addition, it will be able to estimate the probability of an arbitrary string of text. The further sections will describe the theory in more detail, specify the obligatory functionality of your language model, and give an overview of how the project will be graded.

2 Theory

In this section, we will give a summary of theoretical workings of an n-gram language model. This will be an abstract description, and will not necessarily correspond directly to your implementation. For more details we refer to the third Chapter of "Speech and Language Processing" by Dan Jurafsky and James H. Martin `https://web.stanford.edu/~jurafsky/slp3/3.pdf`.

2.1 n-grams

An n-gram is simply a sequence of n consecutive words which appear in a text. For example, the text "*the cat in the hat*" contains the following trigrams (3-grams): ⟨*the*, *cat*, *in*⟩, ⟨*cat*, *in*, *the*⟩, ⟨*in*, *the*, *hat*⟩, and ⟨*over*, *the*, *moon*⟩.

Note that the first and last tokens of the text, such as ⟨*hat*⟩ occur in only one trigram, while the middle tokens, such as ⟨*in*⟩, occur in three trigrams. This property is sometimes undesireable, and it will be for this project as well. In order to prevent this, we can pad our text with special padding tokens before constructing our n-grams. To do this, we add $n-1$ instances of a special token ⟨PAD⟩ to the beginning and end of our text. Thus, our text would become "PAD PAD *the cat in the hat* PAD PAD", and our trigrams would be: ⟨PAD, PAD, *the*⟩, ⟨PAD, *the*, *cat*⟩, ⟨*the*, *cat*, *in*⟩, ⟨*cat*, *in*, *the*⟩, ⟨*in*, *the*, *hat*⟩, ⟨*the*, *hat*, PAD⟩, and ⟨*hat*, PAD, PAD⟩ Thus, each non-padding token now occurs in exactly 3 trigrams (or, in general, in n n-grams).

2.2 Language Models

As its name implies, an n-gram language model is a particular type of language model. In this subsection, we will discuss language models in general. A language model is a model of the statistics of text. It models the probabilities of sequences of words, and can estimate the probability of any arbitrary sequence of words being produced. It does this by predicting which words are likely to come next, given a sequence of previous words.

A language model is defined in terms of a vocabulary $\mathcal{V}$. $\mathcal{V}$ is the set of all words which occur in our text, plus a special end-of-string symbol EOS. (In the context of this project, the symbol EOS happens to be the same as our n-gram padding symbol PAD.)

Formally, a language model produces a conditional probability distribution $p(w_i|\langle w_1, w_2, ..., w_{i-1}\rangle)$, where $w_i \in \mathcal{V}$ is the i^{th} word in the text, or EOS. This distribution can be used for two related purposed: predicting the probability of a given text, or generating a random text according to this distribution.

To predict the probability of a text with a language model, the text is represented as a sequence of tokens, and then terminated with an EOS symbol. For example, the text "*The cow jumped over the moon.*" would be represented as the sequence ⟨*the*, *cow*, *jumped*, *over*, *the*, *moon*, ., EOS⟩. This tokenization step may vary in how it treats punctuation, capitalization, and word boundaries. Once we have a sequence of tokens, the chain rule[1] can be used to represent the probability of the full sequence as a product of the probabilities of each word, conditioned on the subsequent words.

$$p(\langle w_1, w_2, ..., w_m\rangle) = \prod_i p(w_i|\langle w_1, w_2, ..., w_{i-1}\rangle)$$

For the concrete example presented in our text,

$$\begin{aligned} p(\langle \textit{the}, \textit{cow}, \textit{jumped}, \textit{over}, \textit{the}, \textit{moon}, ., \text{EOS}\rangle) = \\ & p(\textit{the}|\langle\rangle) \\ & \times \\ & p(\textit{cow}|\langle \textit{the}\rangle) \\ & \times \\ & p(\textit{jumped}|\langle \textit{the}, \textit{cow}\rangle) \\ & \times \\ & p(\textit{over}|\langle \textit{the}, \textit{cow}, \textit{jumped}\rangle) \\ & \times \\ & p(\textit{the}|\langle \textit{the}, \textit{cow}, \textit{jumped}, \textit{over}\rangle) \\ & \times \\ & p(\textit{moon}|\langle \textit{the}, \textit{cow}, \textit{jumped}, \textit{over}, \textit{the}\rangle) \\ & \times \\ & p(.|\langle \textit{the}, \textit{cow}, \textit{jumped}, \textit{over}, \textit{the}, \textit{moon}\rangle) \\ & \times \\ & p(\text{EOS}|\langle \textit{the}, \textit{cow}, \textit{jumped}, \textit{over}, \textit{the}, \textit{moon}, .\rangle) \end{aligned}$$

As the language model can compute estimates for each of these conditional probabilities, it is able to compute an unconditional probability for the full text.

A language model can also be used to generate random text according to this probability distribution. Remember, for any prefix sequence, the language model can predict a probability distribution for which word comes next. Thus, to build a piece of text, we can start with an empty sequence of words. We use the language model to predict a distribution for first word of our text, using the empty sequence as our prefix. We sample a word from that distribution, and add it to our sequence. Our sequence now consists of a single token. We use this word as a prefix, and let our

[1] `https://en.wikipedia.org/wiki/Chain_rule_(probability)`

language model predict the second word of our text. Our sequence now consists of two tokens. We continue this process until we sample an EOS symbol, at which point we stop.

2.3 n-gram Language Models

An n-gram language model is a language model that computes word probabilities from n-gram probabilities. These n-gram probabilities generally come from a large corpus, where we can simply count how often each n-gram occurs. To predict the probability of a given word w_i coming next, we look back at the previous $(n-1)$ words $\langle w_{i-n+1}, ..., w_{i-1}\rangle$. The probability that w_i comes next is proportional to the number of times that w_i followed $\langle w_{i-n+1}, ..., w_{i-1}\rangle$ in the corpus.

3 Specification and Baseline Functionality

In this section, we will briefly specify the obligatory components of your n-gram language model. The file `specification.pdf` (TBA) contains all the details.

3.1 Specification

Your program will be split into three separate modules: `corpus.py`, `lm.py`, and `main.py` (although you may implement additional modules to use if you would like).

- `corpus.py` will be responsible for reading text files, and processing and tokenizing text. It should contain functions `tokenize` (accepts as input a text and returns a list of tokens) and `detokenize` (accepts as input a list of tokens and returns a single string consisting of all of those tokens together). You can implement those functions as you want (for example, tokenization can take only white spaces and punctuation into account). You can extend the basic functionality of those functions later (see Section 5).
- `lm.py` will implement the core logic of the language model, and will be responsible for predicting probability and generating text. It should contain a definition of class `LanguageModel` with methods such as `train` (trains your language model and learns the n-gram statistics), `p_next` (returns the estimated probability distribution for the next word that occurs after the given token sequence), and `generate` (generates a random token sequence according to the underlying probability distribution).
- `main.py` will be responsible for user interaction — users of the language model will envoke `main.py`, and all functionality will be exposed through this file. When run, it should provide the user instructions, and ask what they would like to do. This program should allow the user to:
  - Create a new language model with a user-specified n
  - Load texts from a file, and train the language model on those texts
  - Generate a text from the language model, and print it to the screen
  - Generate a user-specified number of texts from the language model, and write them to a file
  - Exit the program

  The exact interface is up to you, but your program should print enough instructions to make it self explanatory. We should be able to run this program, without reading your report, and figure out how to do what we want (However, you should still fully document the user interface in your report).

3.2 Baseline Functionality

Download from ILIAS the file `train_shakespeare.txt`. The baseline functionality of your language model should allow the user to follow steps below:

- run `main.py`
- create a new trigram language model ($n = 3$)
- train the model on the file `train_shakespeare.txt`
- generate a new text from the language model and print it to the screen
- generate 5 new texts from the language model and save them into a file `new_shakespeare.txt`

Run the steps above on your own and attach the file `new_shakespeare.txt` to the report.

4 Report

Write a 2-page report (font size 10pt) describing what you did. Your report should describe and justify any specific design decisions you took with your language model. For example, how do you store the ngrams? how do you represent PAD tokens? etc.

You should have a section "Documentation" describing how users should run and use your program. This section is not code documentation so don't put your code with comments here. Instead, you should describe how to use `main.py`: what options does it have, what do they do, etc.

You should describe each extension that you implemented (see Section 5) in its own section of your report. Explain what the extension is, how it works, and why you chose to implement it.

5 Possible Extensions And Analysis

When your language model will have the baseline functionality we encourage you to extend it. Below are some ideas for possible extensions. If you have another idea for an extension you would like to try, ask us if it would be appropriate. Every extension you add should be well documented in your report.

1. Better text processing – cleverer tokenization techniques may yield better language model and subjectively better final text. You can implement those functions on your own (think about special cases, capitalization, etc.) or use some external library (for example NLTK). Better text processing will also factor into the subjective output evaluation.
2. "Speech and Language Processing" contains a lot of ideas for extensions:
   - smoothing, for example additive smoothing[2]
   - back-off methods, such as Katz back-off [3]
   - treatment of out-of-vocabulary words
3. Add a `perplexity` function to your model. It should calculate the perplexity of the given text. Run this function on the file `dev_shakespeare.txt` (see ILIAS) and report the achieved perplexity (use log probabilities to avoid numerical underflow).

[2]`https://en.wikipedia.org/wiki/Additive_smoothing`
[3]`https://en.wikipedia.org/wiki/Katz\%27s_back-off_model`

4. You can implement beam search [4] for your function `generate`
5. As an extension you can also analyze the training/development data and the behavior of your model. Be sure that all the additional code you create is available to use from `main.py` and well documented. As this extension would involve significant additional analysis, you may exceed the 2-page limit for your report to adequately describe your findings. Possible ideas for analysis:
   - train unigram, bigram, and trigram models on `train_shakespeare.txt`. Generate text from every of the models and present them in your report. What differences can you see in those texts? If you have implemented the `perplexity` method – report the perplexity of every of them on `dev_shakespeare.txt`.
   - Change the size of the training data (by taking more and more tokens) and analyze how the perplexity changes (but ensure that the dictionary is the same, see bottom of page 12 in "Speech and Language Processing"). How the amount of out-of-vocabulary words changes? You can plot something here.
   - find and present the most common n-grams of words in the training data
   - ...

6 Grading

Your grade for the project will be determined by the following rubrik, out of a maximum of 100 points.

40 points — Baseline Functionality Your project has to contain all the functionality described in Section 3.2. Additionally, file `specification.pdf` specifies the required components of your n-gram language model (modules, functions, returned types, etc.). Your program have to meet this specification.

10 points — Code Quality Is your code well-structured and readable? Do you use comments to explain what you are doing and why? Are your variable names descriptive? Do you break subtasks nicely into functions? (We will learn about this more on 8.02).

10 points — Tests Do you use unit tests to test individual components of your program? How much of your code is covered by these tests? Do these tests cover tricky edge-cases that are likely to fail?

20 points — Report The quality of your report. Does it specify what you did? Does it document how to interact with your project?

20 points — Extensions and Analysis For every extension you implement and document, you can receive up to 10 points. To get the full 10 points, your extension must be fully implemented and properly documented in your report. Feel free to spend your time on the extension(s) which are the most interesting to you. Remember to document them well. If you have an idea for an extension, and are unsure if it would be appropriate — ask us!

2 bonus points — Subjective Text Quality Run your `generate` function until you generate something amusing. Submit that text with your project as a .txt file. (We will not necessarily grade on how amusing the text is, but how "realistic" it is. This includes stylistic realism (capitalization, punctuation, etc.) and syntactic/semantic realism ((relative) grammatical correctness, semantic meaning, etc.)

[4] `https://en.wikipedia.org/wiki/Beam_search`