Lab 5: Your own small language model

Large language models (LLMs) for AI are all the rage. In this lab you will write a software for working with *small* language models. The software should read a text file to infer a language model from, and then use that model to generate new “text” (not necessarily correct), thus implementing an *Artifical Idiot.*

You are supposed to make this a full C++ project, styled so that it is easy to re-use by others.

- Use git and Github/Bitbucket/GitLab for easy code sharing/distribution and project management.
- Write a Makefile for easy compilation.
- Write a small report to be found in your Github repository

The computational problem

We want to construct a language model that operates on k-grams, or “words”, of fixed length k. The k-grams are created over an alphabet defined by the input, and it may contain any character. So the k-grams are not restricted to linguistic words. The hope is that “real” words are generated by chance often enough to make the text look decent. Larger k means stronger constraints on the output and increased probability of being a proper words.

The model is trained on an input text to extract two key components:

1. The frequency of each k-gram, and
2. the probability distribution over the *next* character following each such k-gram.

For every k-gram w, we examine the set of characters c that appear immediately after w in the text. This allows us to infer a conditional probability distribution over characters, denoted by $P(c|w)$, which represents the probability of observing character c given the preceding k-gram w.

Let $n(w, c)$ be the number of times character c follows k-gram w in the input text, and let $n(w)$ be the total number of occurrences of w. We then estimate the conditional probability as:

$$P(c|w) = \frac{n(w,c)}{n(w)}$$

Example 1

If $k = 2$ and the input text is “ababac”, then the bigram (2-gram) frequencies are:

bigram	f
`ab`	0.4
`ba`	0.4
`ac`	0.2

This is because there are five bigrams in the input text, and “ab” and “ba” appears twice, while “ac” appears once.

Here are the character transition frequencies:

bigram	next char	f
`ab`	`a`	1.0
`ba`	`b`	0.5
`ba`	`c`	0.5

After “ab” in the input, there has only been an “a” following, whereas the bigram “ba” has been followed by either “b” or “c”.

Example 2

If k is 3 and the input text is “Hubba bubba”, then the trigram frequencies are:

trigram	f
`Hub`	0.111
`ubb`	0.222
`bba`	0.222
`ba`	0.111
`a b`	0.111
`bu`	0.111
`bub`	0.111

Note that “ubb” and “bba” are found twice in the input string, and there are nine trigrams in the input text. Three trigrams contain a space, because that was part of the input.

Transition probabilies are a bit boring, but there is so little variation in the small input text:

trigram	next char	f
`Hub`	`b`	1.0
`ubb`	`a`	1.0

trigram	next char	f
`bba`	*space*	1.0
`ba`	`b`	1.0
`a b`	`u`	1.0
`bu`	`b`	1.0
`bub`	`b`	1.0

Generating text

Once a model has been implemented, it can be used to generate new text. The basic algorithm is:

1. Choose a starting k-gram $w = w_1 w_2 \dots w_k$ according to the k-gram frequencies.
2. Look up a character distribution for w, and sample the next character c.
3. Output c.
4. Set $w = w_2 w_3 \dots w_{k-1} c$, i.e., remove the first character and add the sampled character c.
5. Go to 2, unless you have output the requested text length.

C++ has a module `random` which is helpful for generating random numbers as well as discrete objects (like characters) from a probability distribution.

The software

The goal is a Unix tool `slm` (for Small Language Model) that takes three arguments:

- An integer k that decides the word size.
- A filename for a textfile containing any kind of text that is used for the initial training phase, giving output probabilities for characters.
- An integer to decide how many characters to output.

You should be able to use the model like this:

```
$ slm 2 moby_dick.txt 5
it we
```

Here “it we” is conceivable output, five characters from the model.

The code

You should use at least two classes for this project and each class should be found in its own .cpp/.h files. The main function is to be found in a separate file, possibly with supporting code.

A Makefile should be used to compile the code. That is, when we download your project, we should be able to run `make` in the project folder or src folder to have it compile to an executable.

Hints

I think it will be helpful to use `std::map`, which is like a dictionary in Python.

The git repository

You are expected to use git for source code version control and management. We will look at your git commit history to assess whether you have made frequent commits.

- README.md or README.txt, i.e., a textfile that briefly describes your software, how to compile the code, and how to use it. This is file that Github/Bitbucket/GitLab uses to present the project. It is also, in effect, the lab report.
- The code should be found in a folder named `src`.

Submission

When you are ready, you submit a textfile containing a link to your repository.

Grading

- Has git been used actively and sensibly? (1p)
- Is a working and sensible Makefile used? (1p)
- Is the code structured as requested? That is, do the classes have their own files? (1p)