Computer Science 241

Program 1 (33 points)
Checkpoint due Friday, January 14th, 2022 at 10:00 PM
Program due Friday, January 28th, 2022 at 10:00 PM

Read all of the instructions. Late work will not be accepted.

Overview

For the first programming assignment you will work individually to create a program that is able to "learn" human language and generate new sentences in the language. Its concept of language is fairly limited: it knows only an estimate that the probability that a word w_t will follow a sequence of $(n-1)$ words $w_{t-n+1}, w_2, \ldots, w_{t-1}$. In the natural language processing community, these are known as "n-gram" models, and they have proven to be very useful. You will observe that the order of the model, n, has a big effect on the kinds of "sentences" it generates. When it learns from a collection of Grimm's fairy tales, it produces things like:

- When $n = 2$: `<s>` i not upon the cat 'and as she was properly heated the peasants heard it grieves me to his nose to the tree but all that was about the stairs `</s>`
- When $n = 3$: `<s>` and the horse and rode into the jug at his cap on with the sack of meal that we should at least you might take a draught the maid 'what a maypole!' said she and was forced at last the boy 'or take yourself off out of her `</s>`
- When $n = 4$: `<s>` where have you been' said his father 'i have been travelling so long that i should like very well to find out where she is however' said the star-gazer as he looked through his maps but the castle was a garden and around it was a great deal better `</s>`
- When $n = 5$: `<s>` worse still she neglected to make the old woman's bed properly and forgot to shake it so that the boar jumped up and grunted and ran away roaring out 'look up in the tree there sits the one who is to blame' so they looked up and espied the wolf sitting amongst the branches and they called him a cowardly rascal and would not suffer him to come down till he was heartily ashamed of himself and had promised to be good friends again with old sultan the straw the coal and the bean in a village dwelt a poor old woman who had dim eyes could not see it and thought it very beautiful and said to himself 'i will not lose her this time' but however she again slipped away from him unawares and ran off towards home and as the prince followed her she jumped up into the pigeon-house and shut the door behind it and then the raging beast which was much too heavy and awkward to leap out of the window was made of fine black ebony and as she sat looking out upon the snow she pricked her finger and three drops of blood fell upon it `</s>`

(Here `<s>` denotes a special "start-of-sentence" token and `</s>` a special end-of-sentence token.) The program "learns" by reading and gathering statistics from an input *training* text file. Further explanation of n-grams is provided towards the end of this document.

You will implement this functionality in the `LanguageModel` class; a skeleton `.java` file with several dummy methods has been provided to you. The pre- and post-conditions for each method are specified; it will be your job to complete all of the empty methods according to those conditions. You are free to add as many additional `private` helper methods as you would like.

Program 1 Specifications

The Program1 Driver and Its Command-Line Specification

A program named Program1 (`Program1.java`) has been provided to you. **Do not modify Program1's code.** This program drives your `LanguageModel` package, building a language model from the input text, querying the user for phrases and then randomly generating sentences to complete those phrases. It accomplishes this by calling public methods from your `LanguageModel` class. Because it calls your methods, you must not change the method header for any of the public methods. When run, you must supply this program with four arguments: the name of the input text file, the maximum n-gram order to consider, the name of the output vocabulary file to write, and name of the counts file to write, e.g.:

```
C:> java.exe Program1 input.txt 6 vocab.txt counts.txt
```

or (in Linux):

```
$ java Program1 input.txt 6 vocab.txt counts.txt
```

Writing counts can be slow, so Program1 can be called without the last two arguments; e.g.:

```
$ java Program1 input.txt 6
```

When called in this manner, neither the vocab or counts files are written.

I will use Program1 for grading; it is available for you to use during development. The results produced by my completed version on a specific test file will be available to you, but not for the first week of the assignment. It is important that you develop your own test cases to confirm the correctness of your code. Once my output is released, be sure to compare the output of your code against the the file to make sure that your formatting is identical (and that your output is correct). You must create at least one additional test file (named `test1.txt`) and submit it with your program. For additional large scale tests, you will probably want to clean up or "normalize" in the input text. In particular, each line must begin with `<s>` and end with `</s>`, and all words must be separated by exactly one space character. I will provide a pair of scripts to do help with the text pre-processing.

Compilation

Your code should be able to compile cleanly in linux with the following commands on any of the department's lab machines:

```
javac LanguageModel.java Program1.java
```

The Repository

- All code for this program will be developed under your official individual course Github repository, created for you prior to Week 2, available at

  `https://github.com/hutchteaching/202210_csci241_USER`

  where `USER` is replaced with your *WWU* username.
- In a working copy of your repository, you should create a `prog1` directory. All other files for Program 1 should be directly in this directory. **Reminder: if you do not add, commit and push your files, I cannot see them, and thus cannot give you any points for them.** Specifically, your files for grading should be pushed to the main branch (unless you create new branches, main will be your only branch). You can browse Github's web interface to confirm that the files you intend to submit have be properly pushed.
- You must actively use Git during the development of your program. **If you do not have *at least* 5 non-trivial commits for Program 1, points will be deducted.**
- Information on Git and Github is provided in the Git Tutorial on Canvas, and these skills are exercised in Lab 1.

Plan

Prior to the checkpoint, you will need to draft plan for the program, documented in a plaintext[1] file named `plan.txt`. Using proper spelling[2] and grammar, the plan should include the following numbered sections:

1. A one paragraph summary of the program in your own words. What is being asked of you? What will you implement in this assignment?
2. Your thoughts (a few sentences) on what you anticipate being the most challenging aspect(s) of the assignment.
3. A proposed schedule for when you will work on the assignment (e.g., Mondays, Tuesdays and Wednesdays from 5-7pm, and Thursdays 9am-2pm).
4. A list of at least three resources you plan to draw from if you get stuck on something.

Checkpoint

Shortly after the assignment is posted (deadline listed at the beginning of this document), you will have a checkpoint to make sure you are on track. To satisfy the checkpoint, you need to do the following:

- Clone your repository (if not already cloned)
- Create your `prog1` directory
- Create, add, commit and push the following files within the `prog1` directory:
  - `LanguageModel.java` (can be the skeleton I provided, for now)
  - `plan.txt` (the plan in the previous section)
  - `writeup.txt` (can be empty for now)

[1] E.g., created by Notepad, kate, gedit or vim.

[2] Hint: aspell -c plan.txt

Write-Up

Shortly before you submit your program, you will need to draft your assignment write up in a plaintext document named `writeup.txt`. In it, you should include the following (numbered) sections.

1. Your name
2. Declare/discuss any aspects of your code that is not working. What are your intuitions about why things are not working? What potential causes have you already explored and ruled out? Given more time, what would you try next? Detailed answers here are critical to getting partial credit for malfunctioning programs, and failure to disclose obvious problems will lead to additional penalties.
3. Acknowledge and discuss of any parts of the program that appear to be inefficient (in either time or space complexity).
4. In a paragraph, describe how you tested that your code was working. What potential problems did `test1.txt` test for? What other files, if any, did you create for testing? Be specific about your testing strategies.
5. What was the most challenging aspect of this assignment, and why?

Grading

Submitting your work

When the clock strikes 10 PM on the due date, a script will automatically check out the latest pushed version of your assignment. **(Do not forget to commit and push the work you want submitted before the due date!)** Your repository should have in it, at the least, within the `prog1` directory:

- `LanguageModel.java`
- Your writeup: `writeup.txt`
- Your plan: `plan.txt`
- At least one test input file: `test1.txt` (call others `test2.txt`, etc.)
- Any other source code needed to compile your program / class

Your repository need not and **should not contain your .class files.** Upon checking out your files, we will replace your version of `Program1.java` (if one was committed) with my original one, compile all `.java` files, run `Program1` against a series of test documents, analyze your code, and read your writeup.

Points

This assignment will be scored by taking the points earned and subtracting any deductions. You can earn up to 33 points:

Component	Points
Write Up & Test Case File	5
`LanguageModel (constructor)`	4
`getMaxOrder`	1
`saveVocab`	3
`saveCounts`	3
`randomCompletion`	4
`randomNextWord`	5
`getCounts`	5
`convertCountsToProbabilities`	3
Total	**33**

You may also have (potentially significant) deductions from your score for

- Poor code style (e.g. inconsistent indentation, incomprehensible naming conventions, excessively long methods, code duplication, etc.)
- Inadequate versioning
- Errors compiling or running

Details

N-gram Model Terminology

An n-gram is a sequence of n words. The "order" of an n-gram model is just n. The "history" is the first $(n-1)$ words in the n-gram, which is what the model uses to predict the last word in the n-gram. For example, consider the sentence

```
<s> i went to the store </s>
```

If we were looking at trigrams (3-grams, aka n-grams of order 3), we would have the following trigrams:

- Trigram `<s> i went` (in which the history is `<s> i`)
- Trigram `i went to` (in which the history is `i went`)
- Trigram `went to the` (in which the history is `went to`)
- Trigram `to the store` (in which the history is `to the`)
- Trigram `the store </s>` (in which the history is `the store`)

Computing and Storing N-Gram Probabilities

An n-gram model predicts the next word w_t given the history (previous $(n-1)$ words); i.e., with

$$P(w_t|w_{t-n+1},\ldots,w_{t-2},w_{t-1}) \tag{1}$$

Our program will use *maximum likelihood* estimates of these probabilities, according to the following equation:

$$P(w_t|w_{t-n+1},\ldots,w_{t-2},w_{t-1}) = \frac{C(w_{t-n+1},\ldots,w_{t-2},w_{t-1},w_t)}{C(w_{t-n+1},\ldots,w_{t-2},w_{t-1})} \tag{2}$$

Here $C(w_{t-n+1}, \ldots, w_{t-2}, w_{t-1}, w_t)$ denotes the number of times the sequence of words w_{t-n+1}, $\ldots, w_{t-1}, w_t$ appears in the input (training) file; likewise, $C(w_{t-n+1}, \ldots, w_{t-2}, w_{t-1})$ denotes the number of times the word sequence $w_{t-n+1}, \ldots, w_{t-1}$ appears in the the input data. Estimating the probabilities takes places in three steps:

1. First, we must collect all of the n-gram counts for bigram (2-gram), trigrams (3-gram), all the way up to $maxOrder$-grams. The lower order terms are needed at the beginning of a sentence, when we don't have full histories, and are useful if we want to generate text using a lower order model. These should be stored in a `HashMap<String,Integer>`. The string of the n-gram can just be the sequence of words in the n-gram, each separated by a single space.

2. Second, we must collect all of the history counts for all of the same orders. The histories are just the n-gram without the last word. These are slightly different than the counts in step 1 because these will include counts of single word histories (needed for bigrams), whereas the n-gram counts only include counts of bigrams and up. These can (and should) be collected at the same time as the n-gram counts in step 1, though. These should also be stored in a `HashMap<String,Integer>`. The string of the history is the just sequence of words in the history, each separated by a single space.

3. Finally, the counts can be turned into probabilities according to Equation 2. For all n-grams in the n-gram counts, use the n-gram counts for the numerator, and the history counts for the denominator. These probabilities should be stored in a `HashMap<String,Double>`, mapping an n-gram to the probability of the last word in the n-gram given the previous words in the n-gram.

Drawing a Random Word

You can think of a distribution over a set of words as partitioning the interval $(0, 1)$ on the real line. More probable words have larger regions, less probable words have smaller ones. To draw a word, you will draw a number between 0 and 1 according to a uniform distribution, and whichever word it falls upon is the word to be drawn. This should be implemented with the following algorithm:[3]

```
drawRandomWord(history,order)
   d := drawRandomNumber (between 0 and 1)
   cumulativeSum := 0
   for i in 0..vocabSize do
      cumulativeSum := cumulativeSum + P( ith vocabulary word given history )
      if cumulativeSum > d
         return ith vocabulary word
      endif
   endfor
   return "<fail>"
end
```

[3]I am using the class convention that $A..B$ refers to the range of A to B, including A and up to but excluding B. If $A \geq B$ then the range is empty.

Note that you should use $\min(length(history), order - 1)$ history words, so that the overall probability of word given history is an order *order* n-gram whenever possible. By using this algorithm, using a deterministic vocabulary order (case-insensitive ascending alphabetic order), and using the same random number generator seed, you should draw exactly the same "random" sentences as my program.

Potential Questions, Answered

- **Q: Can we store n-grams of different orders in different data structures (e.g. separate bigrams from trigrams)? Or do they all need to be in the same HashMaps?**
  - A: The way the methods are set up assumes that n-grams of all orders (from bigrams to the max order) are stored in the same HashMaps, so please use that convention.
- **Q: Can we add any parameters to the public methods in LanguageModel.java?**
  - A: No, you must not. You can, however, add private helper methods to LanguageModel.java as you see fit.
- **Q: What do we do when the file names for counts and vocab are null?**
  - A: If the filename for count or vocab is null, do not write that file (this is a time-saving measure, since writing the counts can be slow for large n-gram orders).
- **Q: Can we modify our method headers to include a throw clause (e.g. for FileNotFoundException)?**
  - A: No. This will break compilation. To handle these exceptions please use try/catch blocks (see the Java Basics Guide).
- **Q: Can we use a pre-defined sort method or do we have to implement our own?**
  - A: You can use any sort method implemented in a class from any of the following packages: java.io, java.lang or java.util. You do not need to implement your own. java.util.Collections, in particular, may be helpful here.
- **Q: Should `<s>` and `</s>` be included in our vocab (e.g. printed out to the vocab file)?**
  - A: Yes, please include them.
- **Q: Are we allowed to make global private variables?**
  - A: This is strongly discouraged, as it can be abused and lead to poorly written code. Do what you can to avoid using them, and if you can't find a way to get around global private variables, please contact me and we'll strategize.
- **Q: How fast should our program run?**
  - A: If you pass `Program1` only two arguments (i.e. do not have it write the counts or vocab), then with `shakespeare.txt` and a maxOrder of 4, your program should process everything and prompt you to enter a history in $\ll 30$ seconds. With the same arguments except maxOrder is 7, it should still prompt for a history in under 60 seconds. If you do pass it counts and vocab arguments, it will take a bit longer do to the disk writes.

- **Q: If a vocab word isn't really a word (e.g. just a dash), should it still count?**
  - A: Yes. Anything that is delimited by whitespace in our input file is considered a "word" – even if it isn't a word in the traditional sense.
- **Q: If the user enters a capital letter in the prompt, should or program lowercase it for them?**
  - A: No need to. It'll just leave to your program outputting the `<fail>` token, and that's fine.
- **Q: Should our n-gram span sentences?**
  - A: No. This means that `<s>` should only appear in the first position for an n-gram or history, and that `</s>` should only appear in the last position for an ngram, and `</s>` should never appear in the history. Logistically, it's a lot easier to process your text file one line at a time and accumulate counts as you go.
- **Q: My output isn't always the same as yours. Any suggestions?**
  - A: One thing is to make sure you're giving the same sentences in the same order. Because of the random number generator, the completion of the sentence will be different if you do them in different orders. Another thing to check is to make sure your counts are identical to mine. If your counts are even slightly off, your probabilities will be slightly off, and sooner or later you'll draw a different word than me. As soon as you draw a different word, the rest of the completion will be different.

Academic Honesty

To remind you: you must not share code with your classmates: you must not look at others' code or show your classmates your code. You cannot take, in part or in whole, any code from any outside source, including the internet, nor can you post your code to it. If you need help from other students, all involved should step away from the computer and *discuss* strategies and approaches, not code specifics. I am available for help during office hours, as are department tutors, as are our TAs. I am also available via Piazza. If you participate in academic dishonesty, you will fail the course.