

Computer Science 241

(Pair) Program 1 (33 points)

Checkpoint due Monday, April 10th, 2017 at 10:00 PM

Program due Tuesday, April 18th, 2017 at 10:00 PM

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

Overview

For the first programming assignment you will work with your assigned partner 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 $w_{t-n+1}, w_2, \dots, w_{t-1}$. 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. Now that you have a sense of what you will be doing, let’s take a look at *how* you will be working on the assignment.

Pair Programming¹

Pair programming is a software development technique where two programmers work together in front of one keyboard. One partner types code while the other is suggesting and/or reviewing every line of code as it is being typed. The person typing is called the driver. The person reviewing and/or suggesting code is called the navigator. The two programmers should switch roles frequently (e.g. every 20 minutes). For this to be a successful technique the team needs to start with a good program design so they are on the same page when it is finally time to start typing on the computer. **No designing or programming is to be done without both partners present!** Pair programming has been shown to increase productivity in industry and may well increase yours, but there are additional reasons it is being used in this class. First, it is a means to increase collaboration, which is something department graduates now working in industry report that they wish they had more experience with. Second, working in pairs is a good teaching tool. Inevitably, in each pairing the partners will have different styles and abilities (for instance, one person may be better at seeing the big picture while the other is better at finding detailed bugs or one person might like to code on paper first while the other likes to type it in and try it out). Because of that you will have to learn to adjust to another person’s style and ideally you will meet each other half way when there are differences in approach. It’s important that each person completely understands the program and so both parties need to be assertive. Be sure to explain your ideas carefully and ask questions when you are confused. Also it is crucial that you be patient! There is plenty of time allotted to complete this assignment as long as you proceed at a steady pace. Ask for help from me or the department tutors if you need it.

You will be assigned a partner by me via email. Because there is no lab, you will need coordinate with your partner to find times when you can both be present. You will need to contact me ASAP if there is any reason you will not be able to collaborate. **Let me stress again that no designing or programming is to be done without both partners present! If I determine this happened I will fail you and your partner for this assignment.**

¹These guidelines are based on a previous version developed by Perry Fizzano.

Program 1 Specifications

File and Directory Naming Requirements

Pick one of the two partners to host `prog1` in their repository (see later section for more details about the repository). The other partner will have their own repository, but should *not* add any `prog1` directory to it. The following instructions apply to the partner hosting `prog1`. Spacing, spelling and capitalization matter.

- The writeup and all of your source code should be found directly in `yourWorkingCopy/prog1`, where `yourWorkingCopy` is the base directory of the working copy created when you cloned your repository from Github.
- Your source code will be written in a Java file named `LanguageModel.java` (a skeleton of this file has been provided to you).
- Your plan must be a plain-text file named `plan.txt`.
- Your writeup must be a plain-text file named `writeup.txt`.

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. 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 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 provided Github repository. Exactly once at the beginning of the quarter I will create a repository for you named: `201702_csci241_username` where `username` is replaced by your *Western* username.
- Exactly once at the beginning of the quarter each student should *clone* this repository to his or her WWU account. See the Github Tutorial on Canvas for instructions.
- As noted above, your work will be done in a `prog1` sub-directory of your working copy. Therefore, exactly once at the beginning of working on Program 1, either you or your partner will need to create this directory. That person will also need to add, commit and push your writeup and every source code file to your repository. **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 master branch (unless you create new branches, master will be your only branch). You can browse Github's web interface to confirm that the files you intend to submit have been properly pushed.
- You must actively use Git during the development of your program. If you do not have *at least 5* commits for Program 1, points will be deducted.
- Information on Git and Github is provided in the Git Tutorial on Canvas.

You *must* use this repository for the development and submission of your work (failure to have 5 or more non-trivial revisions will result in potentially **large penalties**). It is strongly advised that you attempt clone, and use your repository *as soon as possible* and contact if there are any problems.

Plan

Prior to the checkpoint, you will meet with your partner to construct a plan for the program, documented in a plaintext file named `plan.txt`. Using proper spelling² 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 and your partner will meet to complete this assignment (e.g. Mondays, Tuesdays and Wednesdays from 5-7pm until it's done).
4. A list of at least three resources you plan to draw from if you get stuck on something.

²Hint: `aspell -c plan.txt`

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 `prog3` directory
- Create, add, commit and push the following files:
 - `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)

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:

- `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, I 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
<code>LanguageModel</code> (constructor)	4
<code>getMaxOrder</code>	1
<code>saveVocab</code>	3
<code>saveCounts</code>	3
<code>randomCompletion</code>	4
<code>randomNextWord</code>	5
<code>getCounts</code>	5
<code>convertCountsToProbabilities</code>	3
Total	33

You may also have 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

Write-Up

With your partner, you need to create, add and commit a *plaintext*³ document named `writeup.txt`. In it, you should include the following (numbered) sections.

1. Your names
2. Declare/discuss any aspects of your client or server code that are 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. An acknowledgment and discussion 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. How did you use `test1.txt`? 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?

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`)

³E.g. created with vim, kate or gedit.

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

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

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

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

Here $C(w_{t-n+1}, \dots, w_{t-2}, w_{t-1}, w_t)$ denotes the number of times the sequence of words $w_{t-n+1}, \dots, w_{t-1}, w_t$ appears in the input (training) file; likewise, $C(w_{t-n+1}, \dots, w_{t-2}, w_{t-1})$ denotes the number of times the word sequence $w_{t-n+1}, \dots, w_{t-1}$ 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 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 (1). 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:⁴

⁴It is my convention that $A .. B$ refers to the range A to B , inclusive, as long as $A \leq B$, or else it evaluates to an empty range.

```

drawRandomWord(history,order)
  d := drawRandomNumber (between 0 and 1)
  cumulativeSum := 0
  for i in 0 .. vocabSize-1 do
    cumulativeSum := cumulativeSum + P( ith vocabulary word given history )
    if cumulativeSum > d
      return ith vocabulary word
    fi
  od
  return very last vocabulary word
end

```

Note that you should use $\min(\text{length}(\text{history}), \text{order} - 1)$ 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. As a optional challenge, you can devise a more efficient algorithm to do this random draw - but it must give the same answers as the above algorithm.

Academic Honesty

To remind you: aside from your designated partner, 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 and your partner need help from another pair, 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, but you should attend these hours with your partner. I am also available via email (make sure you and your partner is included in the email and do not wait until the last minute to email). If you participate in academic dishonesty, you will fail the course.