CS 540: Introduction to Artificial Intelligence
Homework Assignment # 7

Assigned: 3/19
Due: 4/9 before class

Hand in your homework:

If a homework has programming questions, please hand in the Java program. If a homework has written questions, please hand in a PDF file. Regardless, please zip all your files into hwX.zip where X is the homework number. Go to UW Canvas, choose your CS540 course, choose Assignment, click on Homework X: this is where you submit your zip file.

Late Policy:

All assignments are due at the beginning of class on the due date. One (1) day late, defined as a 24-hour period from the deadline (weekday or weekend), will result in 10% of the total points for the assignment deducted. So, for example, if a 100-point assignment is due on a Wednesday 9:30 a.m., and it is handed in between Wednesday 9:30 a.m. and Thursday 9:30 a.m., 10 points will be deducted. Two (2) days late, 25% off; three (3) days late, 50% off. No homework can be turned in more than three (3) days late. Written questions and program submission have the same deadline.

Assignment grading questions must be raised with the instructor within one week after the assignment is returned.

Collaboration Policy:

You are to complete this assignment individually. However, you are encouraged to discuss the general algorithms and ideas with classmates, TAs, and instructor in order to help you answer the questions. You are also welcome to give each other examples that are not on the assignment in order to demonstrate how to solve problems. But we require you to:

- not explicitly tell each other the answers
- not to copy answers or code fragments from anyone or anywhere
- not to allow your answers to be copied
- not to get any code on the Web

In those cases where you work with one or more other people on the general discussion of the assignment and surrounding topics, we suggest that you specifically record on the assignment the names of the people you were in discussion with.

Question 1: An n-gram Chatbot [100 points]

In this question you will implement a chatbot by generating random sentences from your HW1 corpus using n-gram language models.

We have created a vocabulary file vocabulary.txt for you to interpret the data, though you do not need it for programming. The vocabulary is created by tokenizing the corpus, converting everything to lower case, and keeping word types that appears three times or more. There are 4699 lines in the vocabulary.txt file.

Download the corpus WARC201709_wid.txt from the homework website (note: this is a processed and different file than the corpus we used in earlier HW). This file has one word token per line, and we have already converted the word to its index (line number) in vocabulary.txt. Thus you will see word indices from 1 to 4699. In addition, we have a special word type OOV (out of vocabulary) which has index 0. All word tokens that are not in the vocabulary map to OOV. For example, the first OOV in the corpus appears as

```
392     are
1512    entirely
0       undermined
12      .
```

The words on the right are provided from the original essays for readability, they are not in the corpus. The word "undermined" is not in the vocabulary, therefore it is mapped to OOV and have an index 0. OOV represents a *set* of out-of-vocabulary words such as "undermined, extra-developed, metro, customizable, optimizable" etc. But for this homework you can treat OOV as a single special word type. Therefore, the vocabulary has $v = 4700$ word types. The corpus has 228548 tokens.

Write a program **Chatbot.java** with the following command line format, where the commandline input has variable length[1] and the numbers are integers:

```
$java Chatbot FLAG number1 [number2 number3 number4]
```

(Part a, 10 points) Denote the vocabulary by the v word types $w_0, w_1, \ldots, w_{v-1}$ in the order of their index (so w_0 has index 0 and represents OOV, and so on). For this homework it is important that you keep this order so that we can automatically grade your code.

You will first create a unigram language model. This is a probability distribution over the vocabulary, for word type $w \in \{0, \ldots, v-1\}$ the estimated probability is

$$p_i \equiv p(w = i) = \frac{c(w = i)}{n},$$

where $c(w = i)$ is the count of word type i in the corpus (i.e. how many times w_i appeared). Note you need to estimate and store p_i for all v word types, including OOV: $p(w = OOV)$ is the fraction of 0's in the corpus. Your $p(w)$ should sum to 1 over the vocabulary, including OOV.

When FLAG=100, number1 specifies the word type index i for w_i. You should print out two numbers on two lines: $c(w = i)$ and $p(w = i)$. When printing the probabilities for this homework, keep 7 digits after the decimal point and perform rounding. For example,

```
$java Chatbot 100 0
7467
```

[1] We provide code skeleton that already handles variable length input.

```
0.0326715

$java Chatbot 100 1
36
0.0001575

$java Chatbot 100 2000
3
0.0000131

$java Chatbot 100 3001
140
0.0006126

$java Chatbot 100 4699
8
0.0000350
```

(Part b, 10 points) Now you implement random sampling from a probability distribution. That is, you will generate a random word type according to its unigram probability. Here is how you do it:

1. Given the multinomial distribution parameter $(p_0, p_1, \ldots, p_{v-1})$, you split the interval $[0, 1]$ into v segments. Segment 0 is $[l_0 = 0, r_0 = p_0]$. Note it is closed on the left. Segment i (for $i = 1, \ldots, v-1$) is
$$\left(l_i = \sum_{j=0}^{i-1} p_j, r_i = \sum_{j=0}^{i} p_j \right].$$
Note these segments are open on the left but closed on the right. Also recall that we want you to order these segments by their word type index.
2. You generate a random number r uniformly in $[0, 1]$.
3. You check which segment r falls into, and output the index of that segment.

A word on sparse storage: this is not an issue for unigrams, but soon you will have many p_i's being zero in bigrams and trigrams. When you implement the segments above, you should not create segments for any $p_i = 0$. Mathematically, such segments are empty and will never get selected. Storage-wise, you do not want to waste space on them. Instead, the suggested data structure is to record the the triple (i, l_i, r_i) in increasing order of i but only for nonzero p_i's. This is known as sparse storage format. Again, be sure to arrange the segments in increasing order of word type index.

However, in order to test your code in a reproducible way, we will specify the random number r from commandline. Specifically, When FLAG=200, we provide number1 and number2 (we guarantee that $number2 \geq number1$), and you should let $r = number1/number2$ (remember to use Java 'double' here so you don't get an integer zero!) instead of a random r. Your code should output three numbers on three lines: the word type index i that this r selects, l_i the left end of w_i's interval, and r_i the right end of w_i's interval.

```
$java Chatbot 200 326 10000
0
0.0000000
0.0326715

$java Chatbot 200 327 10000
1
0.0326715
0.0328290

$java Chatbot 200 329 10000
2
0.0328290
0.0344873

$java Chatbot 200 5000 10000
2364
0.4998906
0.5147278

$java Chatbot 200 99997 100000
4699
0.9999650
1.0000000
```

(Part c, 20 points) Now you will create a bigram language model of the form $p(w \mid h)$, where both w and h (the history) are word types in the vocabulary. Fixing h, $p(w \mid h)$ is a multinomial distribution over word types $w = 0, \ldots, v-1$, and is estimated as follows:

$$p(w \mid h) = \frac{c(h, w)}{\sum_{u=0}^{v-1} c(h, u)},$$

where $c(h, w)$ is the count of the bigram (adjacent word pair) h, w in the corpus. These counts are obtained by letting the history start at the first word position in the corpus, then gradually moving the history one position later, until finally the (history, word) pair "use up" the corpus. For bigrams, that means history stops at the 2nd to last word position in the corpus. For example, if the corpus is "cake cake cake you want cake cake" then $c(cake, you) = 1$, $c(cake, cake) = 3$, $c(cake, want) = 0$. Note it is perfectly fine to estimate $p(w = i \mid h = i)$ for the same word type i. It is also perfectly fine if either w or h or both are OOV.

It is possible that $c(h, w) = 0$ for some history, word combinations. As long as the history appears in the corpus (which is the case for our bigrams), we naturally have the estimate $p(w \mid h) = 0$.

The above discussion is for a fixed h, where $p(w \mid h)$ is a multinomial distribution. You will need to do so for all possible $h = 0, \ldots, v-1$, so that you will end up with v multinomial distributions. This is where the sparse storage becomes important.

When FLAG=300, number1 specifies the history word type index h, and number2 specifies the word type index w. You should print out three numbers on three lines: $c(h, w)$, $\sum_{u=0}^{v-1} c(h, u)$, and $p(w \mid h)$. For example,

```
$java Chatbot 300 414 2297
1054
1082
0.9741220

$java Chatbot 300 0 0
406
7467
0.0543726

$java Chatbot 300 0 1
0
7467
0.0000000

$java Chatbot 300 2110 4240
115
917
0.1254089

$java Chatbot 300 4247 0
41
1435
0.0285714
```

(Part d, 10 points) Now you will use the same function in Part b to sample from a bigram given h. That is, instead of using the unigram probability $p(w)$, we fix some h and you will generate a random word type from $p(w \mid h)$. The method is the same, you just need to do more bookkeeping and record the segments separately for each history h. Specifically, for history h the segments are:

$$[l_{h0} = 0, r_{h0} = p(w = 0 \mid h)]$$

$$\left[l_{hi} = \sum_{j=0}^{i-1} p(w = j \mid h), r_{hi} = \sum_{j=0}^{i} p(w = j \mid h) \right], i = 1, \ldots, v - 1.$$

Again, you should use sparse storage.

When FLAG=400, we provide number1 and number2 (we guarantee that $number2 \geq number1$), number3 is the word type for history h, and you should let $r = number1/number2$ to pick the corresponding word type w from $p(w \mid h)$. Your code should output three numbers on three lines: the word type index i that this r selects, l_{hi} the left end of w_i's interval conditioned on h, and r_{hi} the right end of w_i's interval conditioned on h.

```
$java Chatbot 400 0 100 414
0
0.0000000
0.0009242

$java Chatbot 400 1 100 414
2297
0.0055453
0.9796673

$java Chatbot 400 98 100 414
2298
0.9796673
0.9861368

$java Chatbot 400 81 100 4697
4533
0.8000000
1.0000000

$java Chatbot 400 15 100 4442
4007
0.1463415
1.0000000
```

(Part e, 20 points) Finally you create a trigram language model of the form $p(w \mid h_1, h_2)$, where now the history is the pair of word types h_1, h_2 in that order. Fixing h_1, h_2, $p(w \mid h_1, h_2)$ is a multinomial distribution over word types $w = 0, \ldots, v-1$, and is estimated as follows:

$$p(w \mid h_1, h_2) = \frac{c(h_1, h_2, w)}{\sum_{u=0}^{v-1} c(h_1, h_2, u)},$$

where $c(h_1, h_2, w)$ is the count of the trigram (adjacent word triple) h_1, h_2, w in the corpus. For the cake corpus $c(cake, cake, you) = 1$, $c(cake, cake, cake) = 1$, $c(cake, cake, want) = 0$, $c(cake, you, want) = 1$ and for $u \neq want$ we have $c(cake, you, u) = 0$, $c(want, cake, cake) = 1$ and for $u \neq cake$ we have $c(want, cake, u) = 0$.

Now we have a new problem: the history h_1, h_2 may not appear in the corpus at all! For example, $h_1 = you, h_2 = cake$ never appeared. If so, you simply declare that $p(w \mid h_1, h_2)$ is undefined for any w. We will handle the situation later. In fact, only a small fraction of possible trigram histories appear in the HW1 corpus. You should only store trigram probabilities for these histories. This is also part of the sparse storage strategy.

When FLAG=500, number1 specifies the history word type index h_1, number2 is h_2, and number3 is w. You should print out three numbers on three lines: $c(h_1, h_2, w)$, $\sum_{u=0}^{v-1} c(h_1, h_2, u)$, and $p(w \mid h_1, h_2)$. In the case that $p(w \mid h_1, h_2)$ is undefined, the third line should be the text *undefined*. For example,

```
$java Chatbot 500 23 12 123
0
0
undefined

$java Chatbot 500 5 660 3425
10
402
0.0248756

$java Chatbot 500 2799 556 2364
1
3
0.3333333

$java Chatbot 500 414 2297 2364
99
1054
0.0939279

$java Chatbot 500 0 0 0
35
406
0.0862069
```

(Part f, 10 points) Now you will sample from the trigram model $p(w \mid h_1, h_2)$. The method is the same, though you will decline to generate a word if the trigram is undefined. When FLAG=600, we provide number1 and number2 (we guarantee that $number2 \geq number1$), number3 is h_1 and number4 is h_2, and you should let $r = number1/number2$ to pick the corresponding word type w from $p(w \mid h_1, h_2)$. When this conditional probability is defined, your code should output three numbers on three lines: the word type index i that this r selects, $l_{h_1,h_2,i}$ the left end of w_i's interval conditioned on h_1, h_2, and $r_{h_1,h_2,i}$ the right end of w_i's interval conditioned on the history. Otherwise, your code should output a single line with text *undefined*.

```
$java Chatbot 600 2 5 660 3425
2178
0.3636364
0.4545455

$java Chatbot 600 2 5 3001 104
3083
0.3529412
0.4117647

java Chatbot 600 50 100 496 4517
```

```
540
0.4545455
0.5000000

$java Chatbot 600 33 100 2591 2473
286
0.3086420
0.4444444

$java Chatbot 600 0 100 2297 414
undefined

$java Chatbot 600 0 100 496 4517
5
0.0000000
0.0795455
```

(Part g, 20 points) Now the fun begins! You will generate random sentences using your n-gram language models. But for building a chatbot, we will specify a sentence prefix $s_1, s_2, \ldots, s_t$ which are t initial words (represented by word type indices) in the sentence. Your code will complete this sentence as follows:

1. set seed for randomizer
2. Repeat:
   (a) $h_1 = s_{t-1}, h_2 = s_t$
   (b) generate a random word s_{t+1} from $\tilde{p}(w \mid h_1, h_2)$
   (c) $t = t + 1$ // shifts the trigram history by one position in the next iteration.
3. Until the generated word is a period, or a question mark, or an exclamation mark.

Note in step 1(b) a complication arises from the sentence prefix, and we introduced a placeholder $\tilde{p}$:

- The sentence prefix is empty. In this case, simply let $\tilde{p}(w \mid h_1, h_2)$ be the unigram model $p(w)$ which does not require any history.
- The sentence prefix has only one word s_1. In this case, let $\tilde{p}(w \mid h_1, h_2) = p(w \mid h = s_1)$ the bigram model.
- The sentence prefix $h_1 = s_{t-1}, h_2 = s_t$ as history is undefined for a trigram model. If so, let $\tilde{p}(w \mid h_1, h_2) = p(w \mid h_2)$ the bigram model.
- Otherwise, let $\tilde{p}(w \mid h_1, h_2) = p(w \mid h_1, h_2)$ the trigram model.

When FLAG=700, number1=*seed*, which is the seed of the random number generator; number2=t (which only needs to be 0, 1, or 2), and the next t numbers on the commandline specify the sentence prefix $s_1, s_2, \ldots, s_t$. We will guarantee that s_i is not period, or a question mark, or an exclamation mark.

If $seed = -1$, you actually do not set the seed (this allows you to generate different random sentences). Otherwise you should set the seed to *seed*. To set the seed in Java, use the following code:

```
Random rng = new Random();
if (seed != -1) rng.setSeed(seed);
```

In step 1(b) each time you should generate a new random number $r \in [0, 1]$ in order to generate the random word. This should be done with

```
rng.nextDouble();
```

You should try your code multiple times with the same sentence prefix: when $seed = -1$ your code should complete the sentence in different ways; otherwise it should be the same completion.

Your code will output the completed sentence (starting at s_{t+1}), one word index per line.

```
$java Chatbot 700 0 0
3696
12

$java Chatbot 700 1 0
3694
5
20
0
4683
0
3679
12

$java Chatbot 700 0 1 414
2297
334
2511
2178
60
12

$java Chatbot 700 3 2 2110 311
2869
0
12


$java Chatbot 700 1 1 523
2565
539
4237
557
```

```
4670
2042
1876
3364
4369
4633
533
1057
665
255
5
4223
3298
2364
104
588
617
4281
533
2159
2328
1252
2438
4642
104
957
3905
322
2904
1110
12
```

That last sentence is a little jarring to this professor because it is:

> barely loopholes because they believe would hinder gaining public trust will be created by ai , the professor is a black box to be implemented into different languages with a considerably smaller amount of data .

(Part h, no points) For this part you do not need to develop your code any further, but you will test out the Chatbot that you have developed by actually "talking" to it.

Please download `ChatbotDriver.java` and place this file together with your `Chatbot.java` in the same directory. This driver class basically takes the user input, apply some rules to generate a prefix based on the input (or just use the input itself as the prefix), and call "`java Chatbot 700 -1 prefix`" to generate a response and visualize it as actual words ("OOV" will be displayed for OOV indices). You can compile both files and try chatting with your Chatbot. Also make sure you have the txt files in the directory as well.

Note: because of randomness, your results will differ.

```
$javac Chatbot.java ChatbotDriver.java
$java ChatbotDriver
You: What's your opinion on self-driving cars?
Chatbot: self-driving cars right now , tesla still received plenty of OOV and
needed some more focus towards intellectual fields .
You: healthcare system
Chatbot: healthcare system can be useful .
You: tell me a joke
Chatbot:  to assume that those personal information exposure or government
website being hacked can bring down an interstate .
You: say anything
Chatbot:  politicians into supporting their control over your life would be
an ideal way of monitoring the physical hardware is not yet implemented
technology in terms of employment market .
You: what is your idea of an ideal world?
Chatbot: my idea of an ideal world we live our everyday lives .
```

Have fun and try to make it more intelligent by modifying `ChatbotDriver.java`! (e.g. adding more rules to the `generateCommand` method.)