CS4442b/9542b: Artificial Intelligence II, Winter 2017
Assignment 3: Language Models
Due: April 11

1 Submission instructions

- Submit through OWL by 11:55pm on the due date.
- The extra credit in the assignment does not transfer to quizzes.
- In a few places, I ask you to discuss results. As long as you say something reasonable (i.e. related to the question), you will get the credit.
- Report should be in pdf format.
- Make sure your zip file for submission is no more than 10MB
- VS creates a lot of auxiliary files that may result in super-big size of the zip. To ensure appropriate size, run "clean solution" option before submitting your solution folder. If that doesn't help, use plugin that can create "clean" archives `https://marketplace.visualstudio.com/items?itemName=RonJacobs.CleanProject-CleansVisualStudioSolutionsForUploadi`
- We will not charge late penalty but you will get extra credit of 5% if you hand in your assignment by April 11.
- Submit your Microsoft Visual Studio workspace in a single zipped file with name

  `A3_<first_name>_<last_name>.zip`. Your workspace should have separate projects corresponding to problems: i.e. P1, P2, P3, P4, P5, P6, P7. Plus, if you choose, optional P8. Your workspace can include any additional projects, if needed. The structure of your submission should be as below.

```
A3_<first_name>_<last_name>.zip
|
--- A3_<first_name>_<last_name>
    |
    |-- Ngrams
    |   |
    |   --- <my *.cpp, *.h, *.vcxproj>
    |
    |-- Ngrams.sln
    |
    |-- P1
    |   |
    |   --- <your *.cpp, *.h, *.vcxproj for problem #1>
    |
    |-- P2
    |   |
    |   --- <your *.cpp, *.h, *.vcxproj for problem #2>
```

```
...
    |-- PN
        |
        --- <your *.cpp, *.h, *.vcxproj for problem #N>
```

2 Code Provided

2.1 Intro

In this assignment, you will work with character and word (string) based language models. To efficiently store and count `nGrams`, you should use either hash tables or balanced search trees. C++ standard library has hash table named `unordered_map`. I provide you with code that illustrates efficient `nGram` processing using `unordered_map`, both in the case of character (`char`) and word (`string`) models. If you wish, you can store `nGrams` using another data structure, but make sure it is an efficient implementation (such as a hash table or a balanced tree).

I also provide code to read from files. You must use this code, in order to make sure parsing is done in a consistent way. Lastly, I provide you with code to sample from a probability distribution (for random sentence generation, problem 3) and code to compute edit distance between strings (for spell checking application, problem 6).

All code was compiled and tested under MS studio 2010.

2.2 Code Provided

2.2.1 fileRead.h,fileRead.cpp

Contains code for reading from files. You will need the following functions.

- `void read_tokens(const std::string & filename, std::vector<std::string> & tokens, bool eos)` : reads file from `filename` into a vector of `string`. If `eos = true`, then reads end of sentence marker as a special string `EOS`.
- `void read_tokens(const std::string & filename, std::vector<char> & tokens, bool latin_only)`: reads file from `filename` into a vector `char`. If `latin_only = false`, it reads all characters from file. If `latin_only = true`, it reads only Latin characters and converts upper case to lower case.
- `EOS = "<EOS>"`: special marker for end of sentence.

2.2.2 test.cpp

Illustrates how to build character model and string model based on C++ `unordered_map`, which is a hash table. I will use terms hash table and `unordered_map` interchangeably.

For word (string) language model, you should have `typedef string T`. For character (char) language model, you should have `typedef char T`.

In both cases, an nGram is represented as a `vector<T>`. This vector serves as a key into the hash table. I use `int` as the value associated with the key to give the count of how many times nGram (`vector<T>`) occurred in the text data. When I insert a new nGram, I set its count to 1. If I read that nGram again from the text, I update the count by 1.

One should be careful when using `unordered_map` built-in function `count(key)`. Despite being called `count`, it has only two return values: `0`, in case `key` is not in the hash table, and `1` if `key is in the hash table`. To see how often `key` occurs in the `unordered_map`, that is the value associated with the `key`, use the square bracket operator. But also be aware that the squared bracket operator will insert a new entry in the hash table if entry with the given `key` is not already in the `unordered_map`.

To illustrate, suppose `h` is `unordered_map`, and currently does not have key `"abc"`. If you use `h["abc"]`, it will cause entry with key `"abc"` to be inserted into `h` with some default value in the value field (which is of type `int` in our case). Thus to check if there is an entry with key `"abc"`, use `h.count("abc")`. However, remember that if `h.count("abc")` returns 1, all this means that an entry with key `"abc"` is in your hash table `h`. The actual count of how many times nGram `"abc"` was read from file is in the value field associated with key `"abc"`. You access it with `h["abc"]`. At this point, accessing `h["abc"]` is fine, since you already know that key `"abc"` is in your hash table.

2.2.3 VectorHash.h

- `class VectorHash`: class needed for *unordered_map* in function to construct a hash table for vector keys. You just need to include this into your project.

2.2.4 utilsToStudents.h

- `int drawIndex(vector⟨ double ⟩ &probs))`: samples from probabilities given in input vector of probabilities. Checks that input probabilities approximately add to 1. Use this for the problem of random sentence generation.
- `size_t uiLevenshteinDistance(const std::string &s1, const std::string &s2)`: Computes distance between two strings. Use it for the spell checker problem.

2.3 Samples of Input and Output

Folder `InputOutput` has sample input and output for this assignment.

2.4 Texts

Folder `Texts` contains text files and language files for this assignment.

Problem 1 (10 %)

This problem investigates word distribution in English language and whether it is true that it is enough to know 100 words of English to read about 50% of the text. Read string tokens without `EOS` markers.

(a) Write a program P1 which takes as command line arguments the name of the text file and a value of k. Your program should compute counts of words in an input file and output k most frequent words together with their counts, one per line, word and its count separated by comma. Your program should also compute what percentage of words in the input file are

among the most k frequent words. For example, to output 4 most frequent words from file "text.txt", your program is invoked as

```
P1 text.txt 4
```

The output format should be:

```
and, 4
all, 2
the, 2
their, 2
33.3333 %
```

(b) How many words do you need to know to read about 50% of "DostoevskyKaramazov.txt" and "DrSeuss.txt"? In other words, what should k be set for the most frequent words parameter to get the output from the above program around 50% for these two texts?

Problem 2 (10 %)

This problem investigates language sparseness. Use the word language model. Do not read EOS for this problem.

(a) Write a program `P2` that takes as the command line arguments the names of two text files, the size `n` for the `nGram`, and the last parameter that indicates whether to print out common `nGrams` or not. The program should count and print to the standard output the percentage of nGrams in the second text file that do not occur in the first text file. If the last input parameter is 1, the program should print out the common nGrams between the two files, one per line. If the last parameter is 0, your program should not print out common nGrams.

For example, if we want to count 5-Grams without printing, the program should be executed with:

```
P2 text1.txt text2.txt 5 0
```

The output format should be:

```
65.001
```

If we want to count 6-Grams with the printing option, the program should be executed with:

```
P2 text1.txt text2.txt 6 1
```

The output format should be:

```
he thought much better of it
I can play piano in the
75.001
```

(b) Take two parts of the same novel, "DostoevskyPart1.txt" (as the first input file) and "DostoevskyDostoevskyPart2.txt"(as the second input file). Use your program to compute and write down the percentages of zero count nGrams for $n = 1, 2, 3, ..., 6$. What is the value of n that gives no common nGrams between the two files? What is (are) the largest common nGram for these files?

(c) Repeat part (a) for different writers, "Dickens.txt" (as first) and "KafkaTrial.txt" (as second).

(d) Repeat part (a) for the"opposite" writers, "MarxEngelsManifest.txt" (as first) and "SmithWealthNations.txt" (as second).

(e) Discuss the difference in results between parts (b,c,d).

Problem 3 (15 %)

This problem is about random text generation according to ML language model. Use string model read `EOS` markers.

Random Sentence Generation

Let vocabulary V be all unique words in the input text file. Let m denote the size of V. Let vocabulary words be indexed with integers in the range from 0 to m. Thus, our vocabulary $V = \{v^0, v^1, ...v^{m-1}\}$. You will generate a sequence of words $w_1, w_2, ...$, stopping sentence generation when EOS marker is generated.

Case 1: If $n = 1$, then each word in a sentence is generated independently (no context). Compute $P(v)$ for all words $v \in V$, according to the ML unigram model. Store the computed probabilities in a `vector<double> probs`, where $probs[i] = P(v^i)$, for $i \in \{0, 1, ..., m-1\}$. To generate the first word, use provided function `int drawIndex(vector<double> &probs)`. The output of `drawIndex()` is the index of the word chosen. For example, if the output is 10, this means that v^{10} is chosen, and you set the first word in the sentence w_1 to v^{10}. Generate all other words in the same way, stopping when EOS is generated.

Case 2: If $n > 1$, then there is context. For the first word w_1, its context (previous word) is EOS marker. Use ML to estimate $P(v|EOS)$ for all $v \in V$. Store these probabilities in the vector `probs` and generate the first word w_1 using `drawIndex(probs)`. To generate the second word, use ML to estimate $P(v|w_1)$ for all $v \in V$, store these probabilities in vector `probs` and generate the second word w_2 using `drawIndex(probs)`. Continue this procedure until you generate the `EOS` marker.

Note that if $n > 2$, then as more context becomes available, you should use it. For example, if $n = 3$, then to generate the third (and forth, and so on) word, you should use two previously generated words for context. To be precise, to generate the kth word in the sentence w_k, sample from $P(v|w_{k-n+1}, ..., w_{k-1})$, where $w_{k-n+1}, ..., w_{k-1}$ are $n-1$ previously generated words.

Make sure to initialize random seed generator with something like `srand(time(NULL))`.

(a) Write a program `P3` which takes as command line arguments the name of a text file and the size `n` of the nGram model. Your program should construct an ML (maximum likelihood) language model from `text.txt`, and generate and print out a random sentence according to the procedure described in class and summarized above. For example, to generate a sentence using `trigram` model learned from from file `text.txt`, the program should be invoked with

```
P3 text.txt 3
```

(b) Run your program on "KafkaTrial.txt" with $n = 1, 2, 3, 4, 6$. Discuss your results, pay particular attention to the case of $n = 1$ and $n = 6$. You might want to look inside "KafkaTrial.txt" to interpret your results for $n = 6$.

(c) Set $n = 3$ and generate a random sentence from "MarxEngelsManifest.txt". Compare them with the results generated from "KafkaTrial.txt".

(d) Just for fun: Submit the funniest sentence from your program generated with either $n = 2$ or 3 from any text file you wish. I will run a poll for selecting the funniest sentence.

Problem 4 (10 %)

This problem is about Add-Delta language model. Use word model and do not read *EOS* markers.

(a) Write a program

```
P4 textModel.txt sentence.txt n delta
```

that builds Add-Delta model from text in file `textModel.txt` for n-grams with *delta*. The program should read the sentence in the second file `sentence.txt` and output its log probability to the standard output.

For example, to model language from file `textModel.txt`, estimate log probability of sentence in file `sentences.txt`, build a `5-Gram` Add-Delta model with `delta = 0.1`, use:

```
P4 textModel.txt sentences.txt 5 0.1
```

The output should be formatted as

-55.09

Set vocabulary size to the number of unique words in file `textModel.txt` plus one. We add one to account for all the words that occur in file `sentence.txt` but do not occur in file `textModel.txt`. Intuitively, this is maps all words that are not in our model file to the same word "UNKNOWN".

Implementaiton notes:

- Use `double` data type for probabilities to avoid underflows.
- To avoid integer overflows, use `double` instead of `int`.
- Be careful if your count variables are integers. With integer division, count of `2` divided by count of `5` is 0, but the correct answer is `0.4`. Cast integers to `double` before division.
- The output of your program is log probabilities (to avoid underflow), therefore your output should be a negative number, since $log(x) \leq 0$ for $x \leq 1$.
- Make sure that your program works for the special case of unigrams (`n = 1`).
- If `delta = 0`, then Add-Delta model is equivalent to ML model and some sentences will have probability 0. In this case, your program should not crash but output the maximum negative number in double precision, which is pre-defined in the compiler as `-DBL_MAX`.

(b) Run your program and report the output of your program for the following cases:

- `P4 KafkaTrial.txt testFile.txt 1 1`
- `P4 KafkaTrial.txt testFile.txt 2 1`
- `P4 KafkaTrial.txt testFile.txt 2 0.001`
- `P4 KafkaTrial.txt testFile.txt 3 0.001`

Problem 5 (15%)

This problem is about Good-Turing language model. Use word model and do not read *EOS* markers.

(a) Write a program

`P5 textModel.txt sentence.txt n threshold`

that builds Good-Turing model from text in file `textModel.txt` for n-grams with parameter *threshold*. The program should read the sentence in the second file `sentence.txt` and output its log probability to the standard output. Recall that the `threshold` for Good-Turing is used as follows. If an nGram has rate `r` $<$ `threshold`, use Good-Turing estimate of probability. If `r` $\geq$ `threshold` use ML estimate of probabilities.

For example, to model language from file `textModel.txt`, estimate log probability of sentence in file `sentence.txt`, build a `4-Gram` Good-Turing model with `threshold = 6`, use:

`P5 textModel.txt sentence.txt 4 6`

The output should be formatted as

-55.09

Just as in Problem 4, set vocabulary size to the number of unique words in file `textModel.txt` plus 1.

Implementaiton notes:

- Do not forget to renormalize so that probabilities add up to 1.
- If the user sets `threshold` so high that N_r`=0` for some `r` $<$ `threshold`, then some estimated GT probabilities will be 0. Before computing probabilities, loop over N_r to check that they are not zero for all $r \leq threshold$. You have to do this separately for 1-grams, 2-grams, ..., n-grams. If needed, reset threshold to a lower value so that N_r`>0` for all `r` $<$ `threshold`.
- Use GT probabilities for all n-Grams. Namely, if input $n = 3$, you will need tri-grams, bi-grams, and unigrams to compute probability of a sentence. Use GT estimates for all of them.

(b) Run your program and report the output of your program for the following cases:

- `P4 KafkaTrial.txt testFile.txt 1 1`
- `P4 KafkaTrial.txt testFile.txt 2 5`
- `P4 KafkaTrial.txt testFile.txt 3 5`

Problem 6 (20 %)

In this problem we will use Add-Delta language model to classify which human languages (i.e. English, French, etc.) a given sentence is written in. Use the character based language model that reads all the characters, i.e. `latin_only = false`. Set vocabulary size to 256.

Folder `Languages` contains training and test files for six languages. Each language file has the name corresponding to the language. Training files have endings `1.txt` (`english1.txt, danish1.txt`, etc), and test files have ending `2.txt` (`english2.txt, danish2.txt`, etc). Assume that all the text files are stored in the same directory where the program is run, so you do not have to specify their location.

Train the language models, separately, for each language on training files, i.e. those ending in `1`. Language classification is performed as follows. Given a sentence, compute its probability under French, English, etc. language models and classify the sentence with the language giving the maximum probability. For this problem, a sentence is a sequence of characters of fixed length `senLen`, given as an input parameter. You need to parse an input file into consecutive chunks of characters of length `senLen`, and classify each chunk. If the last chunk is of length less than `senLen`, it should be omitted from classification.

Your program should output the total error rate (in percents), and the confusion matrix. The total percentage error rate for all languages is the overall number of misclassified sentences in all language files divided by the overall number of sentences in all language files, multiplied by 100 to express as percentage.

The confusion matrix is a 6 by 6 array, where `C(i,j)` is the number of sentences of language `i` that were classified as language `j`. That is diagonal has the correct classifications, and off-diagonal wrong classifications.

(a) Write program `P6` for language identification. Your program is invoked with

```
P6 n delta senLength
```

Where `n` is the size of the nGram, `delta` is the parameter for Add-Delta, and `senLength` is the sentence length.

The output of your program should be formatted as:

```
16.79

134 3 0 0 0 1
24 341 1 0 0 0
38 2 85 0 0 0
23 1 0 213 0 0
26 9 1 3 221 0
77 2 0 0 0 50
```

Where the first line is the percentage error rate, and the next size lines is the confusion matrix.

(b) Run your program and report the error rate on the following cases:

- `P6 1 0 50`
- `P6 2 0 50`
- `P6 3 0 50`

(c) Run your program and report the error rate on the following cases.

- `P6 1 0.05 50`
- `P6 2 0.05 50`
- `P6 3 0.05 50`

(d) Run your program and report the error rate on the following cases:

- `P6 3 0.05 50`
- `P6 3 0.005 50`
- `P6 3 0.0005 50`

(e) Compare and discuss the performance between (b,c,d).

(f) Explore and discuss how classification performance changes with the sentence length by running your program on the following cases:

- `P6 2 0.05 10`
- `P6 2 0.05 50`
- `P6 2 0.05 100`

Problem 7 (20 %)

In this problem you will develop a simple spell-checker.

(a) Write a spelling program `P7` that is invoked with:

```
P7 textTrain.txt textCheck.txt dictionary.txt n t delta model
```

The input parameters are as follows: name of text file for model training, name of text file to check spelling, name of text file with dictionary, `n` for the nGram model, threshold for Good-Turing, delta for Add-One smoothing, model to use. As before, if `model = 0` use Good-Turing, and if `model = 1` use Add-Delta.

Use word language model without reading `EOS` markers to read `textTrain.txt` and `dictionary.txt`. File `textCheck` will contain several sentences (one per line) to check spelling of. Check each sentence separately. It is convenient to read `textCheck.txt` with `EOS` marker to separate into sentences. Output the result of checking separately on new line. For example, if `textCheck.txt` has sentences:

```
I lke to eat cereal in the morning.
Pla nicely!
```

The output should be formatted as:

```
i like to eat cereal in the morning
play nicely
```

We will assume that there is only one misspelled word per sentence. For each word `w` in the sentence, find all candidate words in the dictionary with edit distance of $\leq$ `1` from `w`, using

the function `uiLevenshteinDistance()`. Let `C(w)` be the set of such candidate words for `w`. We also include `w` in `C(w)`.

As discussed in class, we will consider all possible sentences where one word `w` is replaced with a word from `C(w)`. This means you should iterate over all words in the sentence, and for each word `w` iterate over all words in `C(w)`, replacing `w` with a word from `C(w)` to generate the next possible sentence.

You will implement a simpler version than what was discussed in class. Instead of implementing the noisy channel model, you just select a sentence with the highest probability according to the language model. Print to the standard output the best sentence. Note that the highest probability sentence can be the unchanged input sentence.

(b) Report and discuss the results of your program for the following cases:

- `P7 trainHuge.txt textCheck.txt dictionary.txt 2 3 1 1`
- `P7 trainHuge.txt textCheck.txt dictionary.txt 2 3 0.1 1`
- `P7 trainHuge.txt textCheck.txt dictionary.txt 2 3 0.01 1`

(c) Report and discuss the results of your program for the following cases:

- `P7 trainHuge.txt textCheck.txt dictionary.txt 1 3 0.01 1`
- `P7 trainHuge.txt textCheck.txt dictionary.txt 2 3 0.01 1`
- `P7 trainHuge.txt textCheck.txt dictionary.txt 3 3 0.01 1`

(d) Report and discuss the results of your program for the following cases:

- `P7 trainHuge.txt textCheck.txt dictionary.txt 1 3 1 0`
- `P7 trainHuge.txt textCheck.txt dictionary.txt 2 3 1 0`
- `P7 trainHuge.txt textCheck.txt dictionary.txt 3 3 1 0`

Extra Credit Problem 8 (20 %)

Implement program `P8` that improves your spelling correction program in problem 7 in any way. You can build a better language model, implement the noisy channel model discussed in class, implement a better edit distance between strings. Hand in your improved program, and report/discuss the cases where your new program does something better compared to the old program.