CS4442b/9542b: Artificial Intelligence II, Winter 2014
Assignment 2: Language Models
Due: March 16

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. This is just to make you think a bit about your results. As long as you say something reasonable (i.e. related to the question), you will get the credit.
- Assignment report should be submitted in PDF format
- Submit your Microsoft Visual Studio workspace in a single zipped file with name

  `A2_<first_name>_<last_name>.zip`.

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

```
A2_<first_name>_<last_name>.zip
|
--- A2_<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 Files Provided

2.3 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 upper case to lower case.
- `EOS = "<EOS>"`: special marker for end of sentence.

2.4 test.cpp

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

For word (string) language model, you should have either `typedef string T` . For character (char) language model, you should have either `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 that 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 use 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`. That is it function `count` does not return the value associated with the `key`, which, in our case, is the count of how many times we read `key` from the text file. To see

how often `key` occurs in the `unordered_map`, you should use the squared 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 actually already there.

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")` method. 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 will be in the value field associated with key `"abc"`, which you can access with `h["abc"]`. At this point, accessing `h["abc"]` is safe, since you already know that key `"abc"` is in your hash table.

2.5 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. No need to understand it. If you want to understand it, ask our wonderful TA who wrote it :). He is much more advanced C++ user than me.

2.6 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. For the random number generator to work correctly, you should include in your main program statement that seeds the random number generator, for example using `srand(time(NULL))`. This also requires `<time.h>` to be included. More sophisticated ways to seed random number generators can be found on the web.

- `size_t uiLevenshteinDistance(const std::string &s1, const std::string &s2)`: Computes distance between two strings. Use it for the spell checker problem.

Problem 1 (10 %)

This problem investigates language sparseness, using the word language model. Do not read end of sentences for this problem.

(a) Write a program `P1` that takes as the command line arguments the names of two text files, the size `n` for the `nGram`, and the last parameter which 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 0, the program should not print out the common nGrams between the two files. If the last parameter is bigger than 0, your program should print out common nGrams, one nGram per line.

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

```
P1 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:

```
P1 text1.txt text2.txt 6 1
```

The output format should be:

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

(b) Take two parts of a novels by the same writer, "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 (b) for different writers, "Dickens.txt" (as first) and "KafkaTrial.txt" (as second).

(d) Repeat part (b) for the"opposite" writers, "MarxEngelsManifest.txt" (as first) and "Smith-WealthNations.txt" (as second).

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

Problem 2 (10 %)

This problem continues investigating data sparseness. Use string language model and read `EOS` markers.

(a) Write a program P2 which takes as the command line arguments the names of two text files and the size n for the nGram. For example, if using 5-Grams, the program should be executed with:

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

The program should count and print to the standard output the percentage of the sentences in the second file that have zero probability under ML (maximum likelihood) nGram model constructed from the first text file. Note that you do not have to compute the probability of a sentence in this problem, just to find if the probability of a sentence is 0. This is the case if the count of any nGram in the sentence is 0. Also note that you need to compute counts of all nGrams for $m = 1, ..., n$. For example, if $n = 3$, you need to compute counts of 1-gram, 2-gram, 3-gram. You can store them all in the same hash-table, or you can have a vector of hash tables, each index for separate value of `n`. Do not store anything under index `0` in this case, since we have no `0`-Grams.

Function `read_tokens` automatically puts end of sentence maker `EOS` at the end of the file if you set input parameter `eos = true`. Suppose file `trainP2.txt` contains:

```
xx ab ab.
bc bc bc
```

When you use my function `read_tokens` you will get a vector of tokens:

```
xx ab ab <EOS> bc bc bc <EOS>
```

You should treat the contents of `trainP2.txt` as two sentences:

```
xx ab ab <EOS>
bc bc bc <EOS>
```

Suppose file `testP2.txt` contains:

```
ab.
bc.
xx.
```

When you use my function `read_tokens` with `eos = true`, it will read the file as:

```
ab <EOS> bc <EOS> xx <EOS> .
```

You should treat the contents of `testP2.txt` as three sentences:

```
ab <EOS>
bc <EOS>
xx <EOS>
```

Your program invoked as

```
P2 trainP2.txt testP2.txt 2
```

should output

`33.33`

The first two sentences have non-zero probability, and the last sentence `xx <EOS>` has zero probability because bigram `xx <EOS>` never occurs in the first file `trainP2.txt`.

When invoked as

```
P2 trainP2.txt testP2.txt 1
```

your program should output

`0`

This is because all there sentences under unigram model (based on counts of individual words) have non-zero probability.

(b) Write down the outputs of your program for input files "DostoevskyKaramazov.txt" as the first file, "Dickens.txt" as the second file, for $n = 1, 2, 3, 4, 5, 6$.

(c) For for $n = 6$, what is (are) the sentence of non-zero probability of length at least 6?

Problem 3 (20 %)

You will develop a random text generator according to a learned language model. Use string language model with reading of `EOS` marker.

(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. For example, to generate a sentence using `3-gram` word based model learned from from file `text.txt`, the program should be invoked with

```
P3 text.txt 3
```

First your program should construct an ML (maximum likelihood) language model from `text.txt`, and then generate a random sentence according to the following procedure.

Random Sentence Generation:

Let the vocabulary `V` be the set of all unique words in the input text file, that is all the unigrams. Denote the size of `V` , that is the number of distinct unigrams, as `|V|`.

The first word w_1 in the sentence has no previous context and should be generated from $P(v)$, $v \in V$. Compute $P(v)$ for all words v in your dictionary V, according to the ML unigram model. Store the generated probabilities in a `vector<double> probs`. To be specific, assume

that the vocabulary words are indexed from 0 to $|V|-1$, that is we list them as $v^0, v^1, ..., v^{|V|-1}$. Then $probs[0] = P(v^0)$, $probs[1] = P(v^1)$, ..., $probs[v^{|V|-1}] = P(v^{|V|-1})$.

To generate the first word, use the function for sampling that I provided you with, `int drawIndex(vector<double> &probs, int |V|)`. The output of `drawIndex()` is the index of the word chosen (sampled). For example, if the output is 10, this means that the word with index 10 in your vocabulary is chosen, namely v^{10}. Thus you set the first word in the sentence w_1 to v^{10}. Words with higher probabilities are more likely to be chosen, as discussed in class.

If $n = 1$, then you generate the second word exactly as the first one. If $n >= 1$, then to generate a second word, you now have context. Use ML to estimate $P(v|w_1)$ for all $v \in V$ as discussed in class, where w_1 is the first word you have already generated. Store these probabilities in the vector of `probs` and generate the second word using my sampling function `drawIndex()`.

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 more precise, to generate the kth word in the sentence w_k, for an nGram model, 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, and $v \in V$.

Optional: If you wish, you can generate the first word in a slightly better way. Notice that some words are more likely to be at the beginning of a sentence, namely those which occur at the beginning of the sentences in the training text file. These words follow the end of sentence marker $\langle EOS \rangle$. Thus you can sample the first word v from $P(v|\langle EOS \rangle)$.

(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: hand in 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 the funniest sentence, and the winner will get an edible present.

Problem 4 (20 %)

In this problem you will implement Add-Delta and Good-Turing language models. The model should be built from the first input text file. Using this model, log of probability of each sentence in the second file should be estimated and printed out to the standard output, one line per sentence.

Recall that Add-Delta has parameter `delta`, and Good-Turing has parameter `threshold`. All parameters and text files will be specified as a command line arguments. The last command line argument specifies which language model to use. If it is 1, then Add-Delta should be used. If it is 0, Good-Turing should be used.

For example, to model language from file `textModel.txt`, estimate log probability of sentences in file `sentences.txt`, build a `5-Gram` Good-Turing language model with `threshold = 3`, use:

```
P4 textModel.txt sentences.txt 5 3 0
```

To model language from file `textModel.txt`, estimate log probability of sentences in file `sentences.txt`, build a `2-Gram` Add-Delta language model with `delta = 0.01`, use:

```
P4 textModel.txt sentences.txt 2 0.01 1
```

Use word language model for this problem. We need to choose the vocabulary size. Unlike problem 3, vocabulary cannot be the number of unique words in the file `textModel.txt`, since we have to account for unseen unigrams (words). In a somewhat ad-hoc manner, let us take the size of the vocabulary to be the number of unique words in file `textModel.txt` multiplied by 2.

For this problem, we will not include `EOS` markers as words in our language model. Therefore, you should build language model from `textModel` without `EOS` marker. However, since you need to parse the second file `textFile.txt`, into sentences, I recommend that you read it with `EOS` markers. Then use these markers to break `textFile.txt` into sentences, but do not include `EOS` into your sentences.

For example, suppose file `sentences.txt` contains:

```
Dogs like to bite.
Cats like to nap.
```

When you use my function `read_tokens` with `eos = true`, it will read the file as:

```
dogs like to bite <EOS> cats like to nap <EOS> .
```

You should treat the contents of `sentences.txt` as two sentences without `EOS` markers:

```
dogs like to bite
cats like to nap
```

Your program should output log probability for each sentence on a separate line. For the example above, the output should be formatted as:

```
-35.08
-55.09
```

(a) Write a program `P4` specified as above.

Implementaiton notes:

- I suggest to use `double` data type for probabilities to avoid underflows.
- Also be careful to avoid integer overflows. When multiplying numbers that could be large, use `double` instead of `int`.
- Be careful if your count variables are of integer type. With integer division, count of `2` divided by count of `5` is 0, but the correct answer is `0.4`. Cast integers to `double` type before dividing.
- The output of your program is log probabilities (to avoid underflow), therefore your output should be a negative number, since $log(x) <= 0$ for $x <= 1$.
- Make sure that your program works for the special case of unigrams (`n = 1`).
- Recall that the threshold `threshold` for Good-Turing is used as follows. If an nGram has rate `r < threshold`, use Good-Turing estimate of probability. If `r >= threshold` use ML estimate of probabilities. Do not forget to renormalize so that probabilities add up to 1. Use the first re-normalization method we covered in class, namely the total weight of all unseen nGrams should stay as estimated by Good-Turing.

- For Good-Turing, if the user sets `threshold` so high that N_r=0 for some `r` $\leq$ `threshold`, then your estimated GT probabilities will be 0. Before computing probabilities, loop over N_r to check that they are not zero for all $r <= threshold$. You have to do this separately for 1-grams, 2-grams, ..., n-grams. Terminate program with an error message if this condition is not satisfied.
- 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`.
- Good-Turing and Add-Delta each worth 10%. If you do not implement either one of them, your program should output "Not implemented" to the standard output if the user tries to invoke the non-implemented model.

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

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

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

- `P4 KafkaTrial.txt testFile.txt 1 1 0`
- `P4 KafkaTrial.txt testFile.txt 2 5 0`
- `P4 KafkaTrial.txt testFile.txt 3 5 0`

Problem 5 (25 %)

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 `P5` for language identification. Your program is invoked with

```
P5 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:

- `P5 1 0 50`
- `P5 2 0 50`
- `P5 3 0 50`

(c) Run your program and report the error rate on the following cases. Notice that in all these cases `delta = 0`, so this is equivalent to ML estimation.

- `P5 1 0.05 50`
- `P5 2 0.05 50`
- `P5 3 0.05 50`

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

- `P5 3 0.05 50`
- `P5 3 0.005 50`
- `P5 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:

- `P5 2 0.05 10`
- `P5 2 0.05 50`

- `P5 2 0.05 100`

(g) Optional, for 5% extra credit. Repeat (b-d) with a character model that reads only 26 Latin characters. Use option `latin_only = true` to read character tokens and set vocabulary size to `26`. Discuss performance as compared with the `256` character model.

Problem 6 (25 %)

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

(a) Write a spelling program `P6` that is invoked with:

```
P6 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.

You have to implement Add-Delta smoothing, but Good-Turing is optional. You will get +5 extra credit if you implement both. If you do not implement Good-Turing, your program should terminate with an error message and terminates if user sets model to 0.

Use word language model without reading `EOS` markers to read `textTrain.txt` and `dictionary.txt`. File `textCheck` will contain several sentences to spell checking of. Check each sentence separately. It is convenient to read `textCheck.txt` with `EOS` marker to make spell checking easier. 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 format of the output should look like:

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

You will implement a simpler version than what was discussed in class. Assume that there is only one misspelled word per sentence. Given the next sentence `S` to check, first compute its probability. Next iterate over all words in the sentence. Suppose current word is `w`. Find all words in the dictionary that are within edit distance of one from `w`, using the function for edit distance that I provide in `utilsToStudents.h`. Let `C(w)` be the set of all words within edit distance of `1` from `w`. Replace `w` in the sentence with a word from `C(w)`. Compute probability of this new sentence. If you get a larger probability, store the sentence. Repeat for all words in `C(w)`, updating the current best sentence (i.e. the highest probability sentence). Repeat this process for all words in sentence `S`. Print to the output the best sentence. Note that the highest probability sentence can be the input sentence `S` unchanged.

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

- `P6 trainHuge.txt textCheck.txt dictionary.txt 2 3 1 1`
- `P6 trainHuge.txt textCheck.txt dictionary.txt 2 3 0.1 1`

- `P6 trainHuge.txt textCheck.txt dictionary.txt 2 3 0.01 1`

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

- `P6 trainHuge.txt textCheck.txt dictionary.txt 1 3 0.01 1`
- `P6 trainHuge.txt textCheck.txt dictionary.txt 2 3 0.01 1`
- `P6 trainHuge.txt textCheck.txt dictionary.txt 3 3 0.01 1`

(d) If you implemented Good-Turing, report and discuss the results of your program for the following cases:

- `P6 trainHuge.txt textCheck.txt dictionary.txt 1 3 1 0`
- `P6 trainHuge.txt textCheck.txt dictionary.txt 2 3 1 0`
- `P6 trainHuge.txt textCheck.txt dictionary.txt 3 3 1 0`

Extra Credit Problem 7 (20 %)

Implement program `P7` that given two text files, builds a word language model from the first file and tests how good this model is on the second file for Add-One, Add-Delta, Good-Turing models. Your program should compare the rates predicted by the first model on the actual empirical rates found in the second file. You should make sure two files are of the same size in the number of tokens. You can simply clip the longer file to have the same number of tokens as the shorter file.

Extra Credit Problem 8 (20 %)

Implement program `P8` that improves your spelling correction program in problem 6 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.