MatrixOrigin Programming Test

MatrixOrigin Technologies

January 19, 2021

1 Overview

There are three problems in this test. For each problem, please provide source code files and a makefile. All programs should compile in Linux at the assigned test servers. You must use C++ programming language to solve the problems. You can use standard C++ libraries. No other third party code is allowed in this test. During the test, you can use your books, use Internet for research, or any other means to come up with proper solutions. However, you should not discuss or share any code, ideas or specific solutions of any problems in this test with other contestants. Any such activities will entirely disqualify the solutions of the involved.

2 Evaluation

Each problem in this test has a maximum score you can get. Here is how we grade your solution. First, if your program compiles correctly by simply typing "make" at a prompt in any directory after extracting your solution files, and produces expected output without a bug, you get the maximum score on correctness. If your Makefile does not work, you get 0 point on that problem. Please make sure that your Makefile works without a fail!

Secondly, after your program passes the correctness, it would be evaluated on performance. That is, we will measure the average time it takes to execute on random, fairly large input files. If there are N number of contestants that passed correctness test, their solutions will be sorted in terms of their speed according to their average time performance. The first ranked solution will get the maximum performance score allocated, and the last ranked solution will get 0 point, while the rest will get corresponding scores according to their ranks linearly decreasing from maximum performance score to 0. If your program fails on the correctness test, you automatically get 0 point on the performance evaluation.

Also please note that there is a sample test input and output files. If your program passes this basic test case provided, you will get 10% of the problem maximum score, even though your program fails on other more complex test cases.

Because the test involves a lot of reading and design within a limited amount of time, there is no requirement for documentation of your source code files or putting together a readme file. The evaluation is strictly based on the "correctness" and "speed" of your program.

3 Submission

Please use Linux tar command to archive your source codes and Makefile. No object files or executables should be included in the archive. An archive name should be firstname.lastname.tar, for example, yingfeng.zhang.tar. If you name your archive name anything other than firstname.lastname.tar, you will get 10 points deduction!

Please submit your solution to "yingfeng.zhang @matrixorigin.cn" or "yingfeng.zhang @gmail.com". Please note that you can submit your solution only once. Any subsequent submissions will be ignored.

4 Problem A: Language Identification

In this test, you need to implement a language identification system. That is, basing on training data, your system could automatically identify the language of any given text. The approach you have to use is based on the well-known N-gram analysis method proposed by Cavnar and Trenkle. The details of the algorithm is described below. You could also read more details in the related paper "N-Gram-Based Text Categorization", while it would suffice reading material below.

4.1 N-Grams

An n-gram is an *n-character* slice of a longer string. The *character* here also means *byte* in our problem context. In this system, we use n-grams of multiple lengths simultaneously, with n ranging from 1 to 5.

As all languages share the **white-space** and **digit** characters, such as space, tab, newline, [0-9], etc, we need to replace them into the underscore character "_" (0x5F in hexadecimal value) to unify their occurrences. If such characters appear adjacently, they need to be grouped into one underscore character. The **white-space** and **digit** characters could be checked using C library function isspace and isdigit.

Thus, the string "5 grams" would be composed of the following n-grams:

1-grams: _, g, r, a, m, s
2-grams: _g, gr, ra, am, ms
3-grams: _gr, gra, ram, ams
4-grams: _gra, gram, rams
5-grams: _gram, grams

4.2 Algorithm

The algorithm consists of two stages, including training and testing.

4.2.1 Training

In training stage, you are given the training set. They are the sample documents labeled with the language they are written in. You need to generate the frequency profiles, one for each language in the training set. The profile generation is simple, which is described below.

1. Scan down the training document, generate all possible n-grams.
2. For each n-gram, store the number of its occurrences in the training document.
3. Sort those n-grams by the number of occurrences in descending order. For those n-grams with equal occurrence number, they are sorted lexicographically, that is, in ascending order of ASCII code.
4. Keep just the top 400 n-grams, which are now in descending order of frequency.
5. The resulting file is then an n-gram frequency profile for the document, we call it *Category Profile* here.

4.2.2 Testing

In testing stage, given an unknown string of text, you need to identify its language. It is performed using steps below.

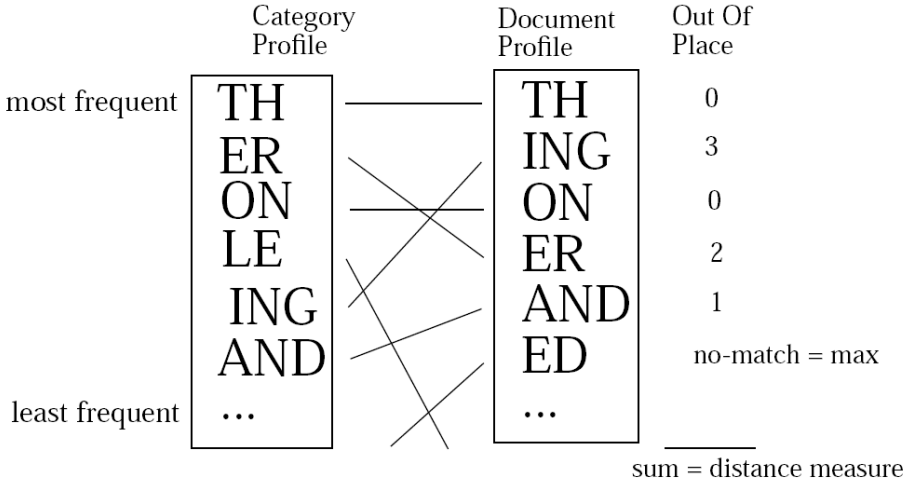
1. Generate the profile for the string to be identified using the same approach in training, we call it *Document Profile* here.
2. Calculate the distance measure between *Document Profile* and each *Category Profile* generated in training stage.
3. Select the minimum distance measure, and identify the unknown string as the language of *Category Profile*.

To measure the profile distance, it merely takes two n-gram profiles and calculates a simple rank-order statistic we call the *out-of-place* measure. This measure determines how far out of place an n-gram in one profile is from its place in the other.

Figure 1 gives a simple example of this calculation using a few n-grams. For each n-gram in the *Document Profile*, they would find its counterpart in the *Category profile*, and then calculate how far out of place it is. The sum of all of the out-of-place values for all n-grams is the distance measure for the document from the category.

For example, in Figure 1, the N-gram "ING" is at rank 2 in *Document Profile*, but at rank 5 in *Category Profile*. Thus it is 3 ranks out of place. If an N-gram (such as "ED" in the figure) is not in *Category profile*, it takes some maximum out-of-place value, such as 400 in this problem context.

Figure 1: Calculating The Out-Of-Place Measure Between Two Profiles



4.3 Problem A.1 (50 points)

Implement the **Training** stage in section 4.2.1.

The *Problem_A* folder contains files below:

- A configuration file *profile.config*.
- The *train* folder contains training files.
- An empty folder *profile* for output.

- Input File
  profile.config is also shown in Table 1. Each line contains three items delimited by space characters. The 1st item is the path of training file. The 2nd item is the path of profile, which file would be the output of your program. According to this two items, your program should read in the training file and generate the profile. The 3rd item is the language of the training file, which item could be ignored in Problem A.1, while it would be useful in Problem A.2.

Table 1: *profile.config* as Configuration File

train/danish.txt profile/danish.profile danish
train/dutch.txt profile/dutch.profile dutch
train/english.txt profile/english.profile english
train/finnish.txt profile/finnish.profile finnish
train/french.txt profile/french.profile french
train/german.txt profile/german.profile german
train/italian.txt profile/italian.profile italian
train/norwegian.txt profile/norwegian.profile norwegian
train/swedish.txt profile/swedish.profile swedish
train/turkish.txt profile/turkish.profile turkish

  The training files in *train* folder are just plain sample files. You need not even know those languages at all, and it would suffice to simply assume those documents as byte streams.

- Output File
  The output are profiles in folder *profile*. Their paths are the 2nd items in *profile.config*. The profile should contain at most 400 lines. Each line is an n-gram and its occurrence number, which are delimited by space character. These lines are sorted by the n-gram occurrence in descending order. For those n-grams with equal occurrence number, they are sorted lexicographically, that is, in ascending order of ASCII code. A sample profile is like below:

_ 6
e 4
, 3
,_ 3
. 3
._ 3
e, 3
e,_ 3
...

Please follow the standards listed below for your implementation:

- The n-gram extraction and their sequences in profiles should be implemented rigorously on section 4.1 and 4.2.1, otherwise the evaluation on your program correctness might be impacted.
- The executable file name of the implementation should be *Problem_A_1*.
- The usage should be *Problem_A_1 profile.config*.
- Your program would read in *profile.config*, and train out each profiles into *profile* folder.

Firstly your program would be evaluated on correctness using sample data. The *Problem_A* folder contains *sample.config*. In running your program using *Problem_A_1 sample.config*, you could verify the correctness of your program output. That is, in *sample* folder, if your program output file *your_sample.profile* is the same with standard output file *standard_sample.profile*, then you have passed this sample test case and you will get 10 points. Otherwise you automatically get 0 point on Problem A.1 and A.2.

Then your program would be evaluated on correctness using training files under *train* folder. If the n-grams and their rankings in your profiles are absolutely the same with the standard answer, you will get 20 points. Otherwise, you will get 0 points on correctness and your program would not be evaluated further in Problem A.

After your program passed the correctness test, it would be evaluated on speed using large training data. The maximum score is 20 points. Based on the relative speed of your program among the programs that passed the correctness test, you will receive a score from 0 to the maximum score. The details on performance evaluation scheme is described in section 2.

4.4 Problem A.2 (50 points)

Implement the **Testing** stage in section 4.2.2.

- Input File
  Your program reads in two files. One is the *profile.config* as in Table 1, the other is a test file. Each line in the test file is a raw text in an unknown language. A sample test file is like below:

  ```
  Perché ha giocato così poco in questa stagione?
  En god projektor vil gi et bedre resultat.
  Hun har allerede en relevant forskerkontakt.
  I think he is very dangerous and unstable.
  ...
  ```

- Output File
  The output are the languages identified on each line in test file. Each identified result should be one of the languages in *profile.config*, that is, the 3rd item in Table 1. The results are delimited by new-line character. A sample output file is like below:

italian
norwegian
danish
english
...

Please follow the standards listed below for your implementation:

- The executable file name of the implementation should be *Problem_A_2*.
- The usage should be *Problem_A_2 profile.config input_file output_file*.
- Your program would load in profiles, which files should be already generated in Problem A.1. The profile paths could be accessible from *profile.config*. Then your program would identify the languages for each line in *input_file*, and output those results into *output_file*.

The *sample* folder contains the sample input/output files for your development use, which test result would not be included in score calculation. You could run your program using *Problem_A_2 profile.config sample/sample_input.txt sample/your_sample_output.txt*. Then in *sample* folder, you could compare your program output file *your_sample_output.txt* with standard answer file *standard_sample_output.txt*.

Your program would be evaluated on accuracy with maximum score 25 points. The accuracy is evaluated by how many languages are identified correctly in your output. Please note that the accuracy result is not necessary 100 percentage using this algorithm, while a relatively high accuracy, such as 90 percentage, could be expected.

If your accuracy is the same or above the accuracy of standard solution, you will get the maximum score 25 points. Otherwise, suppose your accuracy is Y and the accuracy of standard solution is S, your score would be $\frac{Y}{S} \times 25$.

Your program would also be evaluated on speed. The maximum score is 25 points. Based on the relative speed of your program among the programs that passed the correctness test in Problem A.1, you will receive a score from 0 to the maximum score. The details on performance evaluation scheme is described in section 2.

Table 2: Data Set Example of Text Classification

	doc ID	words in document	doc class
training set	1	Chinese Beijing Chinese	China
	2	Chinese Chinese Shanghai	China
	3	Chinese Macao	China
	4	Tokyo Japan Chinese	Japan
test set	5	Chinese Chinese Chinese Tokyo Japan	?
	6	Tokyo Japan Chinese Shanghai	?

5 Multi-Class Classification Problem

Many real world classification problems are multi-class classification problems, such as text classification, digit recognition, automated protein classification, etc. Many algorithms have also been porposed to solve these problems. Usually, we call an algorithm serving for a classification problem as a classifier. Some classifiers can directly serve to multi-class problems, i.e. the Nearest Neighbor classifier, while some other are origninally designed to work on binary class problems, i.e. the Suport Vector Machine (SVM).

There are two basic ways to solve a mutli-class classification problem: to work with a mutli-class classifier, and to work with a series binary classifiers. In this test, we are going to introduce a typical multi-class classification problem that is text classification, and have a try on these two ways: one is driven by Naive Bayes classifier and the other is driven by Decision Directed Acyclic Graph (DDAG) based on Naive Bayes.

5.1 Text Classification

1. Text Classification Introduction
   Text classification is a typical multi-class classification problem. Here is an example. Table 2 shows a collection of one-sentence documents with their labels. Usually, a document can be labeled with many classes. For instance, a document about the 2008 Olympics can be a member of two classes: the China class and the sports class. However, we only consider the case that a document is a member of exactly one class. A given training set is used to learn for the underlying rules of text classification. After learning, the classifier will be performaned on a test set to tell each document's category.

   The bag-of-word model is used in our test to represent a text document. We can collection word's occurring frequency in each document (see table 3). Then, each document can be represented by a numeric vector whose elements hold the words' occurring frequency in this document (see table 4). And a numeric ID can also be assigned to each class label.

2. Input & Output File Format

   - Input File
     There are two kinds of input file: input for training and input for test. Both of these two kinds files are text file, and they share a predefinde format. For the example in table 4 (class China has an

Table 3: Bag-of-Word Occurring Frequency of Documents

doc ID	Beijing	Chinese	Japan	Macao	Shanghai	Tokyo
1	1	2	0	0	0	0
2	0	2	0	0	1	0
3	0	1	0	1	0	0
4	0	1	1	0	0	1
5	0	3	1	0	0	1
6	0	1	1	0	1	1

Table 4: Vector-Representation to Documents

doc ID	doc vectors
1	$\{1, 2, 0, 0, 0, 0\}$
2	$\{0, 2, 0, 0, 1, 0\}$
3	$\{0, 1, 0, 1, 0, 0\}$
4	$\{0, 1, 1, 0, 0, 1\}$
5	$\{0, 3, 1, 0, 0, 1\}$
6	$\{0, 1, 1, 0, 1, 1\}$

ID as 0, while class Japan has an ID as 1), the content of the input training file is as follows:

0,1
4
1,0,0:1,1:2
2,0,1:2,4:1
3,0,1:1,3:1
4,1,1:1,2:1,5:1

The content of the input test file is as follows:

0,1
2
5,-1,1:3,2:1,5:1
6,-1,1:1,2:1,4:1,5:1

The first line of the input file holds all class labels available in the classification problem with a comma separating each label. The second line is the count of the documents stored in this file. From the third line to the end of the file, each line describes a document with its document ID, class ID, and its words occurring freqency. Each line can be separated into several columns by the comma symbol. Among these columns, the first column is the document ID. The second column is the document's class ID, while a negative integer ID, e.g. -1, represents an unknown class. And the other columns hold the vector shown in table 4 in a way that only the non-zero elements are listed out. Each non-zero element is recorded as a pair of numbers separated by a semicolon, while the first number is the zero-based element index, and the second number is the word's occurring frequency.

- Output File
  Output file is a text file, too. The first line also records all the class

IDs in the document collection, and the second line holds the count of the documents. From the third line to the end of the file, each line records a document with its document ID and its class ID. A comma also acts as the separator to these two IDs. Here is a valid output file to the example shown in table 4.

0,1
4 1,0
2,0
3,0
4,1

3. Data Set

   A data set to this problem consists of a training set and a test set. Two data set are provided in our test: one is a sample data set for developing, and the other one is the test data set for evaluating your solution. Only the sample data set is accessible to you.

5.2 Sub-Problem A: Naive Bayes (40 points)

Naive Bayes is a successful classifier that has been applied in many applications. It is a classifier that directly serve to a multi-class classification problem. An online introduction to this classifier is available at here[1].

Please follow the online introduction to develop a Naive Bayes classifier which can work the text classification problem introduced in section 5.1. Besides, please make your solution meet the requirements listed here:

- add-one smothing method should be considered for zero removing.
- The Naive Bayes classifier you are working on is required to be reused in the next problem (Section 5.3). Please take this point into consideration when you design the solution.
- The executable file's name should be Problem_NB. And it should accept a command line like: Problem_NB training_file test_file output_file. Where, training_file is for the file containing the training data, test_file is for the file containing the test data, and output_file is for the output file.
- Please utilize the sample data set (including training and test data) to verify your solution.

5.3 Sub-Problem B: Decision Directly Acyclic Graph (40 points)

Decision Directly Acyclic Graph (DDAG) was proposed to combine a series binay classifier for solving a multi-class classification problem. A complete introduction to DDAG is available in the attached paper named Large Margin DAGs for Multiclass Classification. Although the work introduced by this paper built a real multi-class classifier based on the SVM classifier, we are going

[1] http://nlp.stanford.edu/IR-book/html/htmledition/naive-bayes-text-classification-1.html

to employ a Naive Bayes classifier as a binary classifier and integrate it into the DDAG structure in this test.

Please read the introduction in the given paper, and give a implementation to a mutli-class classifier under the DDAG framework to work out the text classification problem introduced in section 5.1. In addition, please make your solutin meet the requirements listed here:

- The binary classifier integrated into the DDAG architecture should be the Naive Bayes classifier you developed in the previous problem (Section 5.2).
- The executable file's name should be Problem_DAGNB. And it should accept a command line like: Problem_DAGNB training_file test_file output_file. Where, training_file is for the file containing the training data, test_file is for the file containing the test data, and output_file is for the output file.
- Please utilize the sample data set (including training and test data) to verify your solution.

6 Language Independent Extractive Summarization

After reviewing the attached paper Language Independent Extractive Summarization, your job is to implement the proposed text summarization schemes in the paper. The paper uses PageRank and HITS. Your program should take an option (-pagerank, -hits) to produce a correct output depending on the algorithm used. Your program takes a text file as an input and produces its summary to output file. In this problem, we represent a summary of a document as a list of sentences in order of importance (or weight) computed based on the algorithm in the paper. The weight should be normalized in range [0, 1] where 1 signifies the highest weight.

6.1 Input and Output

The input is the filename that is a text document. You can assume it is a pure text document without tags. The output is the summary of the input text document according to the summarization algorithm chosen, PageRank or HITS. The output format should show each sentence with its weight indicated in parenthesis at the start of a sentence. For example, an output could be like this:

```
(0.94323) this is the most important key sentence . . .
(0.82434) this is the second important key sentence . . .
(0.63232) another sentence . . .
(0.4232) yet another one . . .
(0.00002) least unimportant sentence . . .
```

6.2 Program Usage

```
Problem C <-pagerank|-hits> <input text file> <output summary file>
```