Programming Challenge

December 28, 2022

1 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.

1.1 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

1.2 Algorithm

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

1.2.1 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.

1.2.2 Testing

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

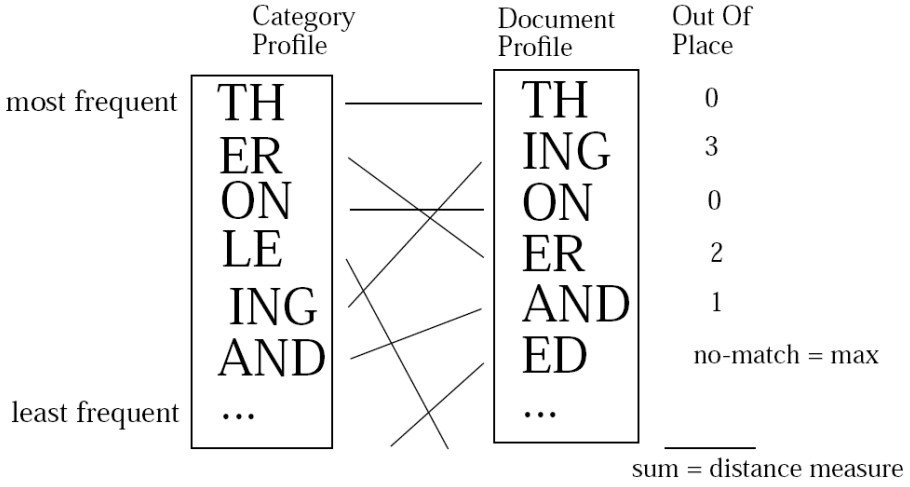
1. Generate the profile for the string to be identified using the same approach in training, we call it *Document Profile* here.
2. Calculate the distance measure between *Document Profile* and each *Category Profile* generated in training stage.
3. Select the minimum distance measure, and identify the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



1.3 Problem A.1 (50 points)

Implement the **Training** stage in section 1.2.1.

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 1.1 and 1.2.1, 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.

1.4 Problem A.2 (50 points)

Implement the **Testing** stage in section 1.2.2.

- 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.