Language Recognition with Token Matching

Chris Nakovski

Background

For my final project, I built a language recognizer. Language recognizers are used in a variety of ways, including automatic translation and voice recognition. The task is to take some document in an unknown language, and try to determine what language it was written in. For practical reasons, I limited myself to using three languages for testing on: English, French, and German.

My approach was to take bodies of existing training text in those languages, and compare them with unknown test documents for similarities in the tokens they are composed of. Two different kinds of tokens I used for comparison were words and characters.

For my first stage, I made a file with all of the unique characters for each language from that language's training data. I then calculated the percentage of characters in the test data that matched those found in the training data. The calculated percentage match score was then used to determine if that language matched the test file.

For the second stage, I did basically the same thing as in stage 1, except I used words instead of characters to calculate percentage matches for test data.

Data

For training and test data, I used ebooks downloaded from project the Gutenberg website. They were readily available for free. Specifically, I used 20-30 of the top downloaded ebooks in each of the three languages I analyzed (French, German, and English). I also used Wikipedia articles written in French, German, and English as test data.

The gold standard of recognition is to ideally recognize books written in a language as having a 100% character/word match for that language, while the percent match for other languages would be zero.

I divided my data to be around 95% for development and training data, and the other 5% for test data. This way I have a much larger training set to draw on for hopefully better results.

Stage 1: methodology

For my first stage, I decided to use unique character percentage match comparisons for language recognition. This is a fine grained approach, and works best when the languages trained on have distinct character sets. I also discarded all characters from the counts that only occur once in the training data. This is so that I don't end up recognizing characters that aren't parts of the language by accident.

I would first read in 20-30 ebooks in a language, and save the unique characters found in them along with their counts into a file for later reading. This would be performed on training sets of English books, French books, and German books. These languages use character sets fairly similar to each other, so I expect the percentage matches for all three languages to be close.

I then have a test file, and analyze how well it fits the previously described language character sets. The program calculates what percentage of characters found in the test file are also found in the character count files for the various languages.

Stage 1: results

For my stage 1 language analysis tests, I tested articles from Wikipedia and other online news sources written in French, German, and English. I had 5 language test files for each one of my languages.

I decided to test my language recognizer according to two different evaluation metrics.

In the first system, I only count the highest scoring language result as a positive recognition. If two or more language scores tie for first, I handle it by marking all of the tied scores as positive language identifications. For this method, a reasonable baseline would be for the program to pick a language at random. This means that for any test file, the chances of recognizing it as any given language would be 1/n where n represents the number of languages to choose from. This assumes that there are equal amounts of test data to choose from, which is true in our case. We can extrapolate this to get baseline precision and recall scores. If we pick a language at random from three languages, both our precision and recall scores should be around 1/3. This means that our F score is also 1/3.

<u>method 1 baseline:</u>

Precision: 1/3

Recall: 1/3

F-score: 1/3

In the second system, I evaluate the recognition scores independently of one another. By this I mean that I've chosen an arbitrary percentage match that I count as a positive identification if reached. I've chosen 70% to be my bar in this case. This means that I count all scores of 70% and up as a positive identification per file per language. The second method is a little more difficult to gauge a baseline for. Since we are using a 70% match as our bar for recognition, let's say that at random 30% (100 – 70)% of the time a language would have a score of 70% or higher. This is totally arbitrary, but it's a better baseline than nothing. Therefore, our precision, recall, and baseline are 3/10.

<u>method 2 baseline:</u>

Precision: 3/10

Recall: 3/10

F-score: 3/10

Note: The task of language recognition in itself is ambiguous and can take on different meanings. For example, the tasks of answering the question “Is this test in German” and “What language is this text written in” both fall under the category of Language recognition but are themselves different questions. I have structured my code and evaluation in such a way as to be able to answer both of these. The first method aims to answer what the most likely language a text is in, while second gives a yes/no answer as to whether a text is in a specified language.

I’ve recorded the results below as a confusion matrix:

First Method (Highest score for positive ID)

	Actual language	Predicted English	Predicted German	Predicted French
English	5	4	5	3
German	5	0	5	0
French	5	1	3	4

Recall: 13/15

Precision: 13/25

F-score: 13/20

Second method (70% for positive ID)

	Actual language	Predicted English	Predicted German	Predicted French
English	5	5	5	5
German	5	5	5	5
French	5	5	5	5

Recall: 1

Precision: 1/3

F-score: ½

Stage 2: methodology

For my first stage, I decided to use unique word percentage match comparisons for language recognition. This is a coarser grained approach than counting characters. This method should also be less reliant on languages having distinct character sets. I also discarded all words from the counts that only occur once in the training data. This is so that I don't end up recognizing a bunch of character sequences that aren't valid words by accident.

I would first read in 20-30 ebooks in a language, and save the unique words found in them along with their counts into a file for later reading. This would be performed on training sets of English books, French books, and German books. These languages use very different word sets, so I expect the percentage matches for all three languages not to be close.

I then have a test file, and analyze how well it fits the previously described language word sets. Specifically, the program calculates what percentage of words found in the test file are also found in the word count files for the various languages.

Stage 2: results

For my second stage language analysis tests, I also tested articles from Wikipedia and other online news sources written in French, German, and English. I had 5 language test files for each one of my languages.

I decided to test my language recognizer according to the same two evaluation metrics as before in phase 1.

I've recorded the results below as a confusion matrix:

First Method (Highest score for positive ID)

	Actual language	Predicted English	Predicted German	Predicted French
English	5	5	0	0
German	5	0	5	0
French	5	0	0	5

Recall: 1

Precision: 1

F-score: 1

Second method (70% for positive ID)

	Actual language	Predicted English	Predicted German	Predicted French
English	5	5	0	0
German	5	0	3	0

French	5	0	0	5

Recall: 4/5

Precision: 1

F-score: 8/9

Instructions for Phase 1

1. Set the flag WORD_MODE in LanguageRecognizer.java to be false.
2. Compile WordCounter, ByteCounter, and LanguageRecognizer. This can be done with the javac command. Example: “javac WordCounter”.
3. Run ByteCounter, giving the name of a directory containing training data and a filename of an output word counts file for the training data when prompted.

   Note: this step may be skipped if you have existing ByteCount files ready for use.
4. Repeat step 2 for all other languages
5. Run LanguageRecognizer, and submit the number of languages to be tested on along with the languages’ names and those languages’ count files you created from the training data back in step 2.
6. Enter the names of the test files to analyze when prompted.

Instructions for Phase 2

1. Set the flag WORD_MODE in LanguageRecognizer.java to be true.
2. Compile WordCounter, ByteCounter, and LanguageRecognizer. This can be done with the javac command. Example: “javac WordCounter”.

3. Run WordCounter, giving the name of a directory containing training data and a filename of an output word counts file for the training data when prompted.

Note: this step may be skipped if you have existing WordCount files ready for use.

4. Repeat step 2 for all other languages
5. Run LanguageRecognizer, and submit the number of languages to be tested on along with the languages' names and those languages' count files you created from the training data back in step 2.
6. Enter the names of the test files to analyze when prompted.

Discussion

The techniques used in this project were very basic, but still managed to deliver impressive results. Stage 2 evaluations for language recognition reached scores of 1 in both precision and recall just by counting whether words were identified to be in a language. I believe that stage 1 didn't achieve success because the languages used had almost identical character sets. If the comparison had been done on test data with a unique writing system, it would have been more effective because the character sets of the training languages would vary enough to be easily differentiable from each other. When combined and refined, stage 1 and stage 2 could form the key components of a high performance language recognizer.

Further refinement of this language recognizer could include features like variable weights on words depending on how often they are used in a language's training data. Another possible improvement would be to use N-grams larger than just words to recognize languages. These further refinements would require a much larger corpus to train on however.

This goes to show that there are tasks in computational linguistics that are solvable without intensive computation, and only rudimentary data analysis. Perhaps this sort of recognition and cross referencing of tokens in text could solve other identification problems. For example, maybe we could use word counts in an article to guess its subject. I predict that the same methods used here can also effectively identify other attributes in data.

Sources

Book Sources - Project Gutenberg. Web, 29 May 2015

Test Language articles - Wikipedia. Web, 29 May 2015