Selecting and Weighting N-Grams to Identify 1100 1185 Languages

Ralf D. Brown
Language Technologies Institute
Carnegie Mellon University

2 September 2013
16th International Conference on Text, Speech, and Dialogue
Why Language Identification?

- Internet is becoming more multilingual
- Text processing often uses language-specific models or techniques
  - to process arbitrary data from the web, we need to select the appropriate model/technique
The Approach

- vector space models
  - one (or more) per language/encoding pair to be identified
- k nearest neighbors
  - cosine similarity (normalized inner product) as the distance measure
Selecting N-Grams

- Use the K highest-frequency n-grams of length 3 through N which don't
  - start with multiple whitespace characters
  - start with multiple digits
  - start with a punctuation mark repeated three times
  - contain a newline

- In the original application, unigrams caused too many false positives and bigrams only slowed down the program
Weighting N-Grams

- Two main factors: probability and length
- Need to include probability factor to be able to distinguish between multiple languages including an n-gram
  - but less than full because common n-grams will also be common in the test input
- Want to give bonus for length because longer n-grams are more informative but less common
  - but proves to have very little impact
Filtering N-Grams

- Not all n-grams contribute equally
- If an n-gram occurs nearly as frequently as one of its substrings, the substring does not help to identify the language
  - remove the substring from the model and include another n-gram which was not in the top K
Effects of Filtering

![Graph showing effects of filtering]
Discriminative Training, aka Stopgrams

• Some letter sequences are invalid in a language
  – appearance in test input thus strongly suggests the input is not in that language

• Failure to occur in the training data is a strong indicator of invalidity
  – the more training data, the stronger the indication

• Add n-grams from other language models which don't appear in the training data, giving them negative weight
Selecting Stopgrams

- Determine languages similar enough for confusion
  - compute cosine similarity between models
- Combine all n-grams in similar-enough models
  - weight by max frequency, max similarity, and amount of training data
- Scan training data for n-grams in combined set
  - add any that don't appear with the negative of the previously computed weight
- Scale stopgram weights by a further factor of 9
Stopgram Weighting

![Graph showing error rate (%) against relative stopgram weight]

- raw macro-average
- raw micro-average
- smoothed macro-average
- smoothed micro-average

2 September 2013
Ralf D. Brown @ TSD2013
Scoring Input

• Naive method
  – convert input into a feature vector of term frequencies
  – multiply f.v. by each model's term vector

• Far faster method – direct incremental computation
  – for each n-gram in input, increment the score for each model containing that term by its weight in the model
  – normalize by length of input

• Apply optional inter-string score smoothing
Score Smoothing

• In running text, consecutive strings are likely to be in the same language

• Add a portion of the previous string's scores for each model to the current string's scores
  – greatly reduces errors
  – but too much smoothing will cause actual language change to be missed
How Many N-Grams to Use?

• Unlike some methods, vector-space cosine similarity always benefits from more n-grams in the models
  – accuracy asymptotically approaches an optimum
• Thus, the choice is a simple trade-off between resource requirements and accuracy
Experiments

• Compared **whatlang** against four other open-source programs
  - libtextcat (rank-order statistics)
  - mguesser (hashed vector space)
  - LangDetect (Naive Bayes)
  - langid.py (NB with information-gain selection)

• Modified libtextcat, mguesser, and LangDetect to provide per-line identifications

• Speed-optimized LangDetect and langid.py
A Caveat

- langid.py accuracy can most likely be improved, but
  - training is very slow
  - there are multiple parameters to tune
  - settings expected to improve accuracy require more than 16 GB for training
Data

• Training
  - 1278 files in 1190 languages
  - some converted into multiple encodings for 1297 models total

• Testing
  - 1225 files, 3 omitted because fewer than 50 strings
    • also no test files for Northern Uzbek (accidental omission) or Klingon
  - 1185 languages in test set
Data Sources

- GigaWord corpora
  - English, French, Spanish, Arabic, Chinese
- European Parliament
  - Danish, Dutch, Finnish, German, Greek, Italian, Swedish
- Wikipedia
  - used over 100 languages, ~200 have useful amounts
  - requires cleaning
- Bible
  - and some Bible school text
Data Size

- Mean amount of available training data: 1.4 million bytes per model
  - amount used limited to 1.0, 1.1, 1.5, or 2.0 million bytes, depending on program
- Test strings range from 25 bytes to 65 characters (potentially up to 195 bytes)
- Test sets contain 50 to 1000 strings per language/script pair
  - mean is 710.8, median is 713
## Training Performance

<table>
<thead>
<tr>
<th>Program</th>
<th>Time</th>
<th>RAM (MB)</th>
<th>Model Size (MiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>whatlang</td>
<td>698s (115s elapsed) @ 500</td>
<td>~100 @500</td>
<td>16.6 @ 500</td>
</tr>
<tr>
<td></td>
<td>1671s (296s elapsed) @ 3500</td>
<td>~380 @3500</td>
<td>101.6 @ 3500</td>
</tr>
<tr>
<td></td>
<td>2173s (396 s elapsed) @ 5600</td>
<td>~630 @5600</td>
<td>158.5 @ 5600</td>
</tr>
<tr>
<td>libtextcat</td>
<td>481s</td>
<td>25</td>
<td>5.2</td>
</tr>
<tr>
<td>mguesser</td>
<td>166s</td>
<td>&lt;1</td>
<td>21</td>
</tr>
<tr>
<td>LangDetect</td>
<td>1061s (756s elapsed)</td>
<td>90</td>
<td>43</td>
</tr>
<tr>
<td>langid.py</td>
<td>115548s (6 threads,19856s elapsed)</td>
<td>~10000</td>
<td>260.5</td>
</tr>
</tbody>
</table>
Model Size vs. Accuracy

![Graph showing model size vs. accuracy](image)
Model Size vs. Accuracy (2)

![Graph showing the relationship between model size and accuracy. The x-axis represents millions of bytes per model collection, and the y-axis represents error rate (%). Different lines represent various model collections, each with a different symbol and line style.](image-url)
## Evaluation Performance

<table>
<thead>
<tr>
<th>Program</th>
<th>N-Grams</th>
<th>Time</th>
<th>RAM</th>
<th>Error Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>whatlang</td>
<td>500</td>
<td>32s</td>
<td>19 MB</td>
<td>4.735% / 4.632%</td>
</tr>
<tr>
<td>libtextcat</td>
<td>500</td>
<td>2269s</td>
<td>20 MB</td>
<td>6.440% / 6.130%</td>
</tr>
<tr>
<td>whatlang</td>
<td>3500</td>
<td>59s</td>
<td>97 MB</td>
<td>1.876% / 1.772%</td>
</tr>
<tr>
<td>mguesser</td>
<td>1500</td>
<td>17,129s</td>
<td>81 MB</td>
<td>15.365% / 15.429%</td>
</tr>
<tr>
<td>langid.py</td>
<td>800</td>
<td>522s</td>
<td>2.7 GB *</td>
<td>2.781% / 2.445%</td>
</tr>
<tr>
<td>whatlang</td>
<td>5600</td>
<td>66s</td>
<td>143 MB</td>
<td>1.615% / 1.522%</td>
</tr>
<tr>
<td>LangDetect</td>
<td>5634</td>
<td>1141s (1590s CPU)</td>
<td>9.1 GB</td>
<td>3.435% / 3.108%</td>
</tr>
</tbody>
</table>

whatlang scored without inter-string smoothing

all elapsed times include ~6 seconds scoring overhead

langid.py momentarily requires nearly twice as much RAM at startup
Conclusions

- whatlang is faster (on short strings) and more accurate than four other open-source language identification programs
- filtering out less-useful n-grams improves accuracy
- adding negative weights for “impossible” n-grams improves accuracy
- assuming successive strings are likely to be in the same language greatly improves accuracy
Questions?
Obtaining the Programs

- whatlang
  - http://la-strings.sourceforge.net/
- libtextcat
  - https://github.com/scientific-coder/libtextcat
- LangDetect
  - https://code.google.com/p/language-detection/
- langid.py
  - https://github.com/saffsd/langid.py [“ralfbrown” branch]
- mguesser
  - http://www.mnogosearch.org/guesser
Obtaining the Data

- Europarl
  - http://www.statmt.org/europarl/

- Wikipedia
  - http://sourceforge.net/projects/la-strings/files/Language-Data/

- Bibles
  - Creative Commons-licensed Bibles from above URL
  - others from http://bible.is, http://bibles.org,
## Characteristics of the Models

<table>
<thead>
<tr>
<th>Program</th>
<th>N-Gram Size</th>
<th>Encoding</th>
<th>Model Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>whatlang</td>
<td>3-6 bytes (configurable)</td>
<td>don't care</td>
<td>single binary file</td>
</tr>
<tr>
<td>libtextcat</td>
<td>1-5 bytes</td>
<td>don't care</td>
<td>one text file per model</td>
</tr>
<tr>
<td>mguesser</td>
<td>1-5 bytes</td>
<td>don't care</td>
<td>one text file per model</td>
</tr>
<tr>
<td>LangDetect</td>
<td>1-3 characters</td>
<td>requires UTF-8</td>
<td>one JSON file per model</td>
</tr>
<tr>
<td>langid.py</td>
<td>1-5 bytes (configurable)</td>
<td>presumes UTF-8</td>
<td>single binary file</td>
</tr>
</tbody>
</table>
Similarity Computation

- For each byte position in input
  - Start at root node of trie (compacted 256-ary tree)
  - While node has children and more input available
    - descend according to next byte of input
    - advance input pointer
    - look up weighting factor for current match length
    - For each match record associated with new node
      - add weight in record times length factor to score for model# in record