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

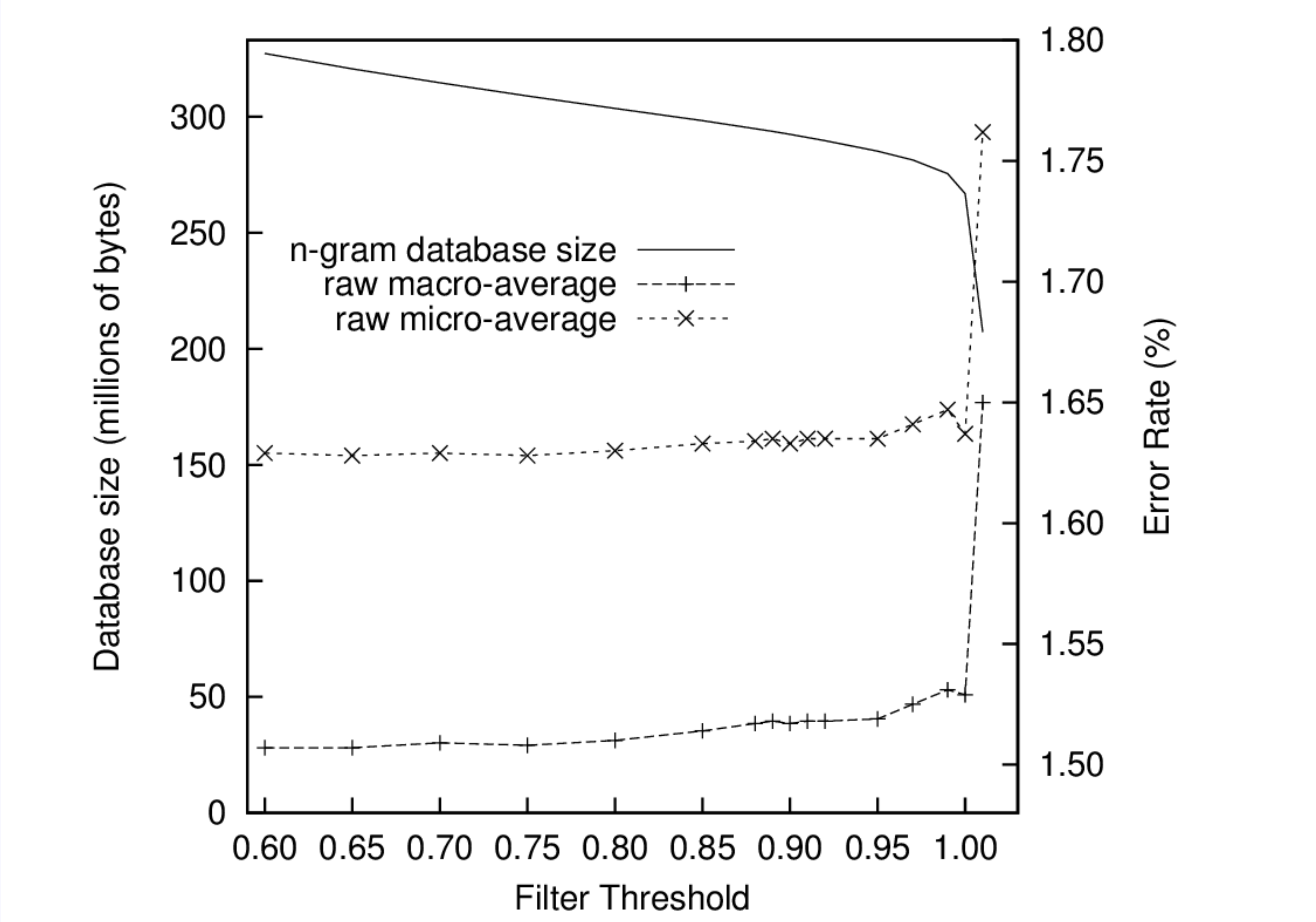
Weighting N-Grams

- 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



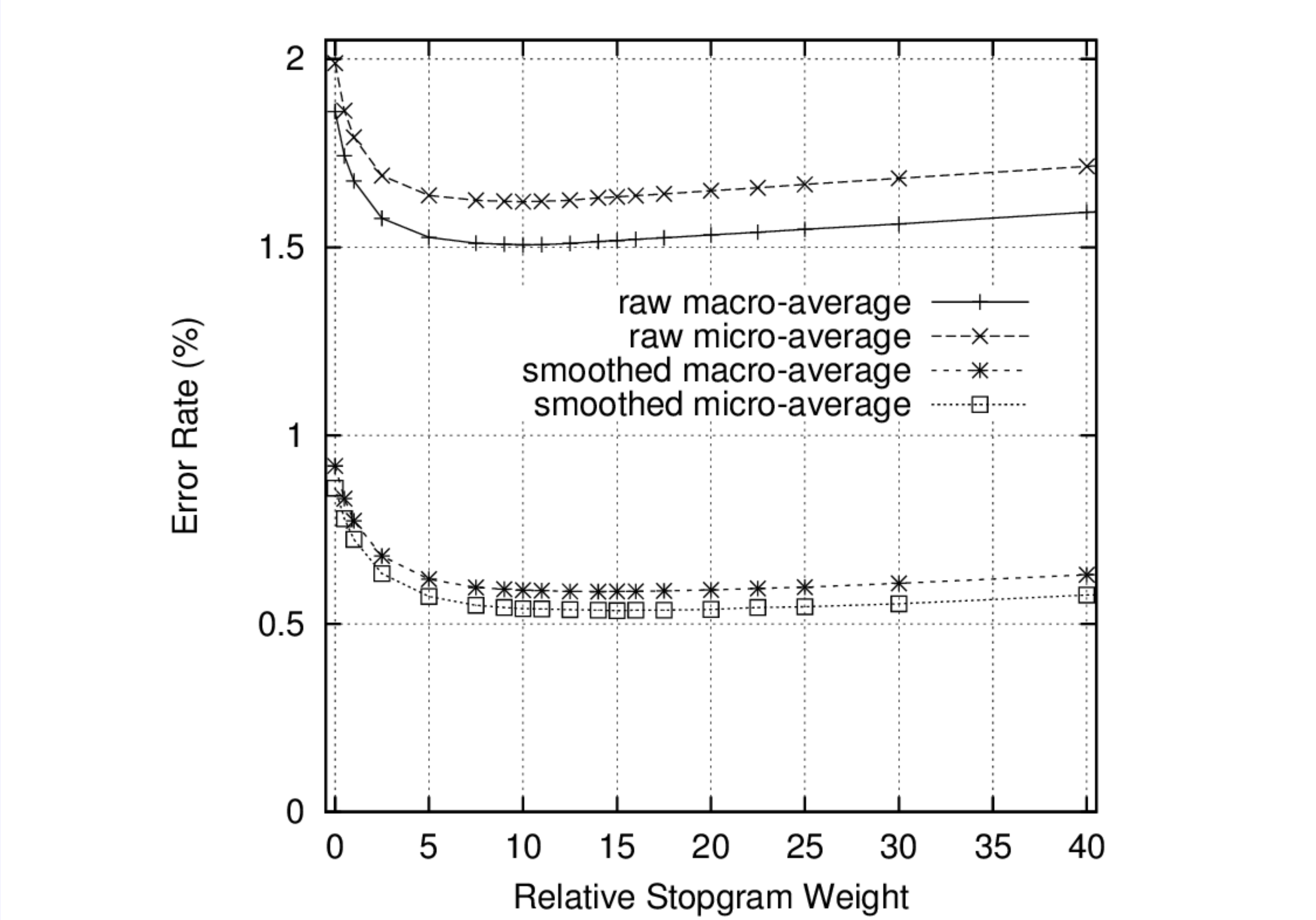
Discriminative Training, aka Stopgrams

- Some letter sequences are invalid in a language
  - appearance in test input thus strongly suggests the input is <u>not</u> 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



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
  - setttings 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

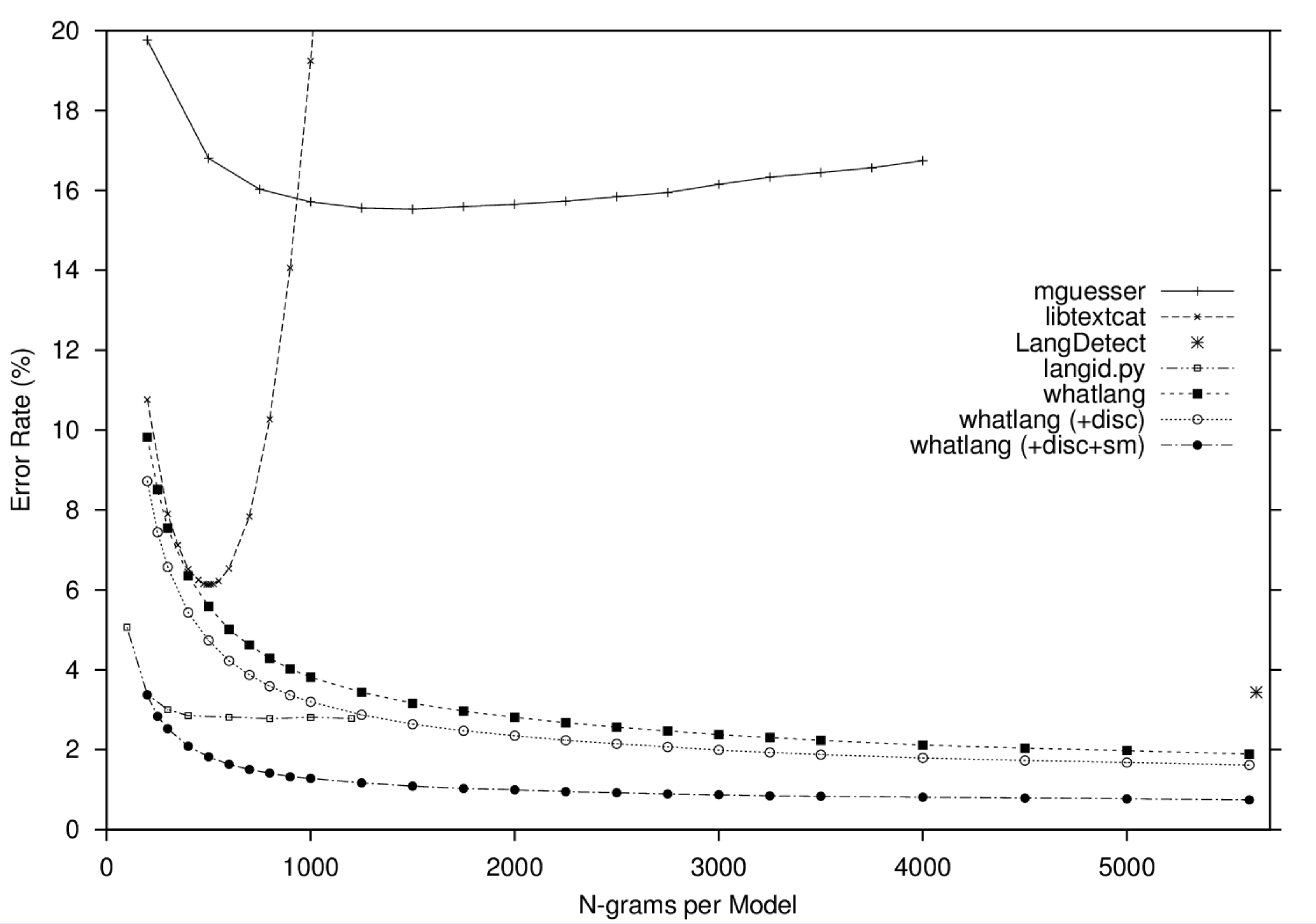
Data Size

- 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

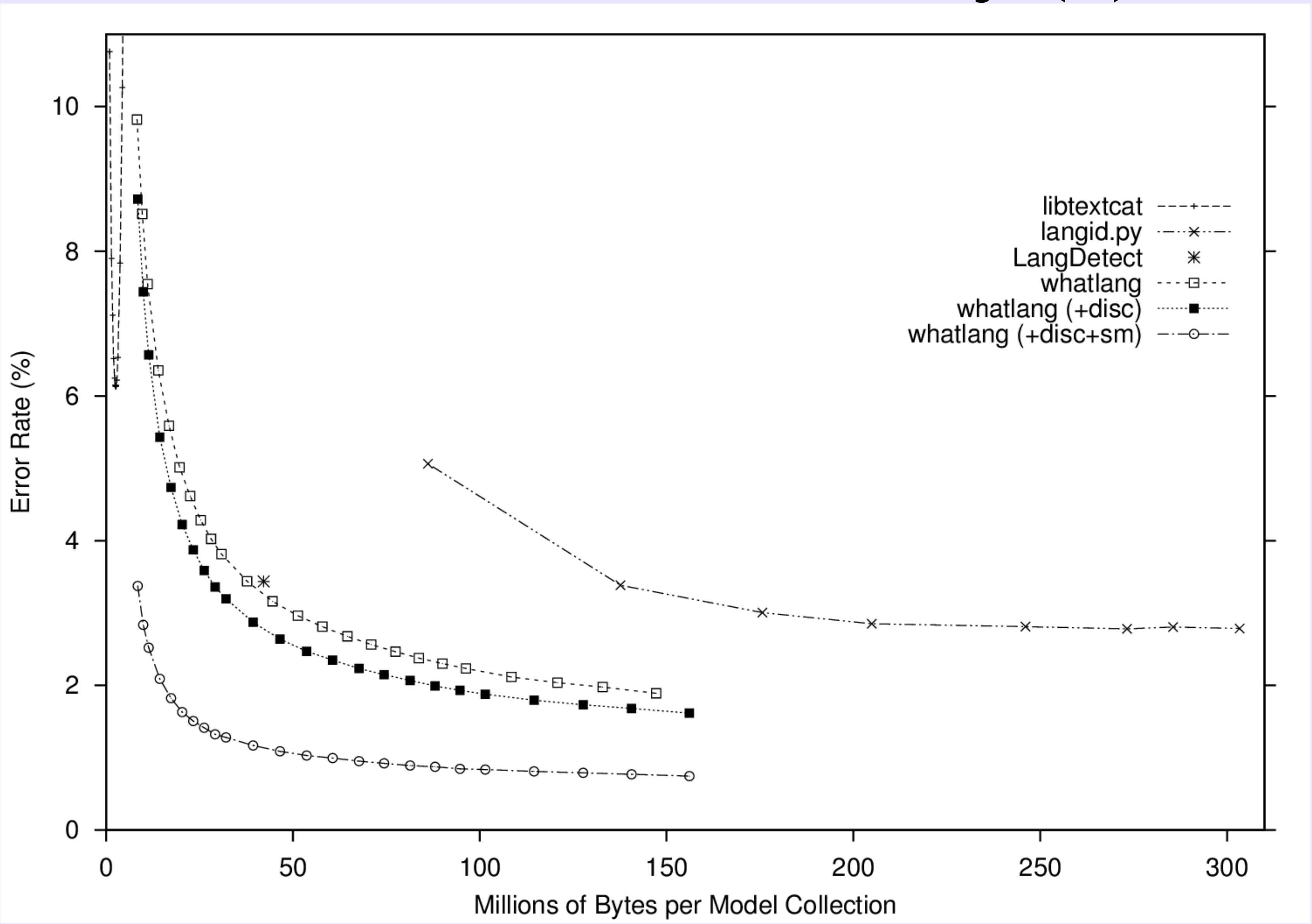
Training Performance

Program	**Time**	**RAM (MB)**	**Model Size (MiB)**
whatlang	698s (115s elapsed) @ 500 1671s (296s elapsed) @ 3500 2173s (396 s elapsed) @ 5600	~100 @500 ~380 @3500 ~630 @5600	16.6 @ 500 101.6 @ 3500 158.5 @ 5600
libtextcat	481s	25	5.2
mguesser	166s	<1	21
LangDetect	1061s (756s elapsed)	90	43
langid.py	115548s (6 threads,19856s elapsed)	~10000	260.5

Model Size vs. Accuracy



Model Size vs. Accuracy (2)



Evaluation Performance

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

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, http://youversion.com, http://www.gospelgo.com

Characteristics of the Models

Program	**N-Gram Size**	**Encoding**	**Model Format**
whatlang	3-6 bytes (configurable)	don't care	single binary file
libtextcat	1-5 bytes	don't care	one text file per model
mguesser	1-5 bytes	don't care	one text file per model
LangDetect	1-3 characters	requires UTF-8	one JSON file per model
langid.py	1-5 bytes (configurable)	presumes UTF-8	single binary file

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