Modeling Early Word Learning using Inferences over Referential Intention

Shaan Bijwadia and Tiwalayo Eisape

Final Project for Computational Models of Cognition, Boston College
Joshua Hartshorne
7 May 2018

Abstract: We aim to implement the word-learning model described in Frank, Goodman, & Tenenbaum[1] and replicate their results. We implement their calculations of lexicon posterior score, and two inference models: a modified basic Metropolis-Hastings model, and a parallel tempering scheme based on the paper's inference technique. The models were trained on the same data originally used by Frank et al. The modified "sticky" Metropolis-Hastings model outperformed our replication of the paper's parallel tempering scheme, though both models are outperformed by the paper's original results.

Code

Our program is written in Python 2. After the source folder is copied to a directory, run: `$ pip install -r requirements.txt` to install dependencies. The code contains the following files:

- `sampleLexicon.py`: generates a random lexicon
- `goldStandard.py`: contains the hand-coded gold standard matrix used in Frank et al.
- `utils.py`: various functions, including one to compute the F-score of a lexicon
- `world.py`: constructs the world and corpus from `words.txt` and `objects.txt`
- `breed.py`: breeds two lexicons together for use in parallel tempering
- `mutate.py`: mutates a lexicon using predefined operations
- `params.py`: contains the global parameters for the model.
- `scoreLexicon.py`: computes the posterior score of a lexicon given a corpus
- `lexicon.py`: contains Lexicon object
- `model.py`: runs "sticky" Metropolis-Hastings model
- `wurwur.py`: runs the model used by Frank et al.

To run the modified Metropolis-Hastings model, use the command: `python model.py`. For the parallel tempering model, use `python wurwur.py`. Every few iterations, the program outputs various important statistics, including the information about each individual model, and prints an alert whenever a model is discovered with a posterior score

[1] Frank, M. et al. (2009). Using Speakers' Referential Intentions to Model Early Cross-Situational Word Learning. *Psychological Science*, Vol 20, Issue 5, pp 578-585.

higher than any other the model has seen. At larger intervals, the program displays information about and the contents of the current highest-posterior-score model, if it has changed since the last time the best model was printed.

Implementation

Model Representation

As in Frank et al., words and objects are represented as integers, and the world object contains a key to translate words between English and their encodings. Lexicons are represented as a Python object whose only instance variables are a word list and an object list, where corresponding indices indicate association. The corpus is a Python list of situation objects, each of which contain the words and objects used in its

Posterior Score

A lexicon's posterior score given a corpus is computed as described in Frank et al. The process is made significantly more efficient by precomputation; the `gamma_intents` matrix for each situation needs to be computed only once, and `word_cost` matrix is computed just once per lexicon. This information is cached in memory and retrieved as needed when iterating through each situation in the corpus.

Since the probability of encountering a certain corpus given a lexicon is vanishingly small, we follow the good sense contained in the source paper's code base by performing the computation in log-space. Thus, instead of returning P(L|C)P(L), the function `posteriorScore` actually returns ln(P(L|C)P(L)) = ln(P(L|C) + ln(P(L)). The result is that the scores are returned as numbers on the order of -1.5×10^4, instead of very tiny fractions.

Inference - Sticky Model

Inference is performed as described by Frank et al., using a technique quite similar to standard Metropolis-Hastings. Lexicons are resampled using the `mutate` function which considers the following operations: add a pair, delete a pair, and swap a pair for another one. As in the original implementaion, pairs are added with probability corresponding to the following measure of "mutual information:"

$$MI(w, o) = \frac{C(w, o)}{C(w)C(o)}$$

In the equation above, the function *C* denotes "count," i.e. the number of times a word or object appears in a corpus, or the number of times a pair appears together. This assigns higher probability to pairings that occur frequently relative to the frequency of the word and object generally. Notably, the measure assigns zero probability to pairings that never occur together in the corpus. This information is computed upon construction of the corpus and is stored in `world.mis` as a Python list.

A standard Metropolis-Hastings model is contained in `model.py`, denoted by the name "MH." The model begins by generating a proposal lexicon based on the `mutate` function, and accepting it with probability *p* where *p* is defined by:

$$p = \frac{P(L')}{P(L)}, \text{ if } P(L') < P(L), \text{ 1 otherwise.}$$

In the equation above, *P* denotes the posterior score of a lexicon, and *L'* refers to the proposed lexicon. The value *p* is then taken to the power of 1/*T*, where *T* is the temperature of the model. However, since the function `posteriorScore` returns the natural log of the lexicon's posterior score, the following transformation is used in the calculation:

$$\begin{aligned} p^T &= \left(\frac{P(L')}{P(L)}\right)^T \\ &= e^{\ln\left((\frac{P(L)}{P(L)})^T\right)} \\ &= e^{\frac{\ln P(L') - \ln P(L)}{T}} \end{aligned}$$

Thus the returned values of `posteriorScore` can be plugged into the bottom equation to compute p^T. The proposal is accepted with probability p^T. If the move results in a better lexicon than has been encountered before, it replaces the previous best and is displayed at the end of the search.

This traditional Metropolis-Hastings search proved to be rather poor at finding lexicons with posterior scores much higher than its initial value, and tends to stop discovering "new bests" after a few hundred iterations. To help it along, we developed the following scheme, referred to hereafter as the sticky inference model: two models are run simultaneously, and the process keeps track of the highest-scoring lexicon that either of them has seen. One of them, dubbed MH, performs standard Metropolis-Hastings search. The other one, called the sticky model, is "reset" every three hundred iterations: its current lexicon is discarded, and the model begins again using the current best model as its starting position. Ideally, new promising spaces are found by the MH model, which the sticky model begins exploring while the MH model is free to wander elsewhere, combining advantages of both breadth and depth in search.

Inspiration of the sticky model came from the suspicion that from any given "good" lexicon, a better lexicon can be constructed through minor adjustments. However, the standard MH model tends to wander away from promising lexicons quickly, without returning. A sticky model is disrupted before this can occur too drastically. Repeated runs of the model reveal that the sticky model is responsible for finding most of the new "best" lexicons, especially at high iterations.

We run the two models in parallel with temperature 1, which generally results in acceptance rates >50% very early in the run, which reduce to <25% after quite a bit of training. Parameters are lightly tuned by hand to α=12, γ=0.1, and κ=0.05. An improved study would experiment liberally with the parameters to find the optimal settings, or infer them.

Inference - Parallel Tempering

In `wurwur.py`, we attempt to replicate the parallel tempering technique (referred to in the source paper as simulated tempering) used by Frank et al. Five models are initialized, with different temperatures, and run simultaneously. Every five iterations, two models are selected randomly[2] and bred together. Possible breeding moves include swapping lexicons, swapping word-object pairs, and swapping the words or objects between pairs across lexicons. Proposal lexicons are accepted with probability p^T, computed as before, according to each model's T parameter, which were initialized to [0.0001, 1, 10, 100, 1000], as in the source paper. Ideally, more greedy (low temperature) models will explore the the space immediately around promising innovations found by the higher temperature models.

Results

Lexicons are judged based on F-score, a statistic that compares a lexicon to the gold-standard, the hand-coded lexicon judged to be most "correct." By far the best-performing models in terms of both F-score and posterior probability are generated by the sticky model of inference. The model approaches its final value relatively quickly, after about 10,000-20,000 iterations. Eventually, the model stops finding better lexicons, only occasionally finding a new lexicon with a slightly better score. After a representative 10,000 iterations, the model resolved to a lexicon with an F-score of 0.46, which is about 16% less than 0.55, the F-score found by the model in the source paper.

The wurwur model fares much more poorly, ceasing to find new best lexicons after only a few hundred iterations, with underwhelming F-scores. An anecdotal run for thousand iterations (though it didn't update the best matrix after iteration 350) resulted in a lexicon with F-score of 0.168421. Parameter optimization by hand did not appear to significantly impact the result.

[2] In standard parallel tempering, the best models are selected to be bred together. However, we follow the implementation in Frank et al., which chooses models to breed uniformly.

Results - final best lexicons for representative runs of the models

Model	Sticky Model	Wurwur Model
Iterations	10,000	1000
F-score	0.461538	0.168421
Precision	0.454545	0.126984
Recall	0.46875	0.25
Lexicon	meow: baby sheep: sheep pig: pig hat: hat rings: ring birdie: duck hand: hand bigbird: bird bird: duck moocow: cow on: ring bunnyrabbit: bunny blue: ring mhmm: hand kittycat: kitty put: ring baby: book bunnies: mirror yellow: ring laugh: cow and: book oink: pig bear: bear piggie: pig and: hand book: book reach: ring david: book over: rattle lamb: lamb ah: bird bottle: bear see: rattle	fun: hand, family: woman pool: rattle, now: hat dododo: duck, bottle: bear on: ring, cows: pig meow: kitty, soft: bird kittycat: kitty, bathroom: rattle roll: pig, wiggle: hand will: cow, davids: lamb bunnies: bunny, smilie: eyes littler: lamb, bunnyrabbit: mirror dada: boy, c: bird w: bird, bumpba: lamb, fall: duck, splashes: duck rings: ring, bunny: bunny laugh: cow, another: woman daddy: girl, hand: hand lambie: lamb, bottles: face who: woman, lamb: lamb horses: book, courtney: girl hmm: hat, old: man grab: hat, time: bird rollie: duck, cute: sheep forest: bunny, them: ring our: bird, set: face hat: hand, he: duck grandma: pig, pretty: mirror mommy: boy, sheep: sheep mouth: rattle, talks: bird jumping: bunny, j: rattle wanna: hat, mots: mirror kittycats: kitty, when: cow must: bird

Conclusion

Standard Metropolis-Hastings appears to be insufficient to tackle the task of word learning. Previous research has found success in using a simulated tempering model. Though our inquiry was unable to do so, we have discovered a simple technique that produces decent results at the task. The posterior score calculation developed by Frank et al. performed quite well and was impressively easy to implement in an efficient manner. Further research is required to find scoring techniques which can better discern between referential nouns and closely associated non-referential words ("oink" mapping to "pig"), and to find lexicons with even larger F-scores. Promising areas to explore include particle filtering and a smarter implementation of breeding in parallel tempering.