Bayesian Inference with Tears

a tutorial workbook for natural language researchers

Kevin Knight

September 2009

1. Introduction

When I first saw this in a natural language paper, it certainly brought tears to my eyes:

$$P(k|\mathbf{x}_{-i}, \beta) \;=\; \int P(k|\theta)P(\theta|\mathbf{x}_{-i}, \beta)\, d\theta$$

Not tears of joy.

Later, I realized that I was no longer understanding many of the conference presentations I was attending. I was completely lost, just staring at some fly buzzing at the window. Afterwards, in the hallways, colleagues would jump up and down, exclaiming how simple it all was. "Put a Gaussian prior on the hyper-parameter!" they'd say. "Integrate over the possible values!" Or my all-time favorite: "Sample with *burn-in*!"

Here's some Kleenex. Let's go.

2. Turning point

At some point in my life, I downloaded a decision tree package, and I trained it on some data that I had. I didn't know a thing about decision trees. I thought it was damn cool, though, and I showed the results to a friend. He said, "What splitting criterion did you use?"

Well, someone smarter than me might have said "information gain". I was too dumb to know that people actually expect you to throw around buzzwords even if you don't understand them. So I said, "I don't know, who cares." I'm sure my friend thought, what an idiot.

That was not the turning point in my life, though.

The turning point was EM. Here was a learning algorithm that could figure out the answer *without you telling it the answer*. Even more mysterious, you couldn't download EM from the web. EM was a framework, a philosophy. There were starting to be a bunch of conference papers using EM, and it looked important. It seemed to me that there was going to be a split. Some researchers were going to learn EM and apply it to lots of problems. Others weren't going to bother.

At the time, I was trying to align English sound sequences with Japanese sound sequences, and I knew that EM could do it for me. It was hard, though. I spent two years reading Brown et al 93. When I finally got it to work, I was pretty fired up, and I told David Yarowsky. He said, "EM is the answer to all the world's problems." Wow! I figured everybody should know about it, so I wrote "A Statistical MT Tutorial Workbook". At that time, I thought EM was only for alignment, but later I realized it was about more stuff. I worked with a college student named Jonathan Graehl, and we managed to package up

EM in a finite-state software toolkit called Carmel, which people *could* download from the web. That let me and others do lots of EM experiments without lots of new code and new bugs.

3. Another turning point?

When I recently started seeing work on Bayesian inference, I asked myself, is this another turning point? I looked at all that math and all those integral signs, and I breathed deeply. At the time, I didn't have a mantra to go with my breathing. (Argh, I still don't.) I attended Bayesian tutorials and invited talks, but there was that same fly buzzing at the window. Zzzt.

Anyway, I started thinking, what if I just don't bother?

I had Mr. Robinson for calculus in high school, but I mean, man, that was a long time ago. I remember Beth, who sat in front of me in class. She always got A's. She worked very hard. We all had uniforms. Beth Hatfield… Hmm, I said to myself.

4. Does it work?

Well, do Bayesian techniques really work, on real problems? If it works, then I should bother. Bayesian techniques seem to beat plain EM[1] on many tasks, but it doesn't yet seem to be a central component of big state-of-the-art systems. Maybe the engineers just haven't gotten hold of it yet.

Scaling seems to be a problem, compared to plain EM. But then again, Bayesian techniques are said to be less competitive when large amounts of data are available, anyway – prior knowledge is more powerful when data is scarce. Bayesian techniques also have a number of extra knobs and hyper-parameters to twist, and search involves a lot of black art. "How much burn-in did you use?" Good question.

Overall, I see a lot of respectable people who get quickly turned on, then quickly turned off, by Bayesian techniques. So who the heck knows? If you want to not bother, my respect for you is undiminished. Life's short. You could get married and have a child in the same amount of time.

5. Promises

Starting with Section 8, I will relate some bits that I came to understand. I wanted to be able to understand more conference presentations, and I also figured there might be a way to apply these techniques to my work. I was attracted by two promises:

[1] Throughout this tutorial workbook, I will refer to "EM" or "plain EM". By this, I don't just mean the EM algorithm, the computer method that iteratively estimates parameter values for a data set. I mean to include all those traditional generative models (HMM, etc, etc) that have proven popular and worthwhile in natural language processing, plus the EM-based methods for training those models. When people say "EM works!" or "EM doesn't work!", that's usually what they mean. This tutorial workbook presents some **alternative generative models**, plus some **alternative training procedures**, which I lump together under "Bayesian techniques". In reality, you could mix and match, applying the EM algorithm to the new Bayesian models, or applying new training procedures to our existing popular, worthwhile models.

(a) Small models. A small model that explains the data is better than a big model. Bayesian systems can be set up to prefer small models, whereas typical EM doesn't. Moreover, when EM has too many parameters to play with, it usually heads off in the wrong direction.

(b) Small steps versus big steps. If you give EM enough parameters, it will memorize the training data, and learn nothing useful. EM will figure out how to take "one giant step" to explain the data, rather than many small steps. This is a complete failure case for EM, which comes up often. For example, if we have 5000 English/French sentence pairs, and we tell EM to find phrase pairs inside them, EM will come back and say, "Hey, I found these 5000 *awesome* giant phrase pairs, and by the way, P(French | English) = 1.0". Bayesian systems can be biased against this bad behavior.

I don't know that Bayesian techniques will actually deliver on these promises, on the big problems that we care about. But I'll explain why folks are fired up.

It will help if you are a little bit comfortable with EM -- if you aren't, you might want to work through "A Statistical MT Tutorial Workbook" or some such.

6. This tutorial workbook contains no novel scientific work

I've assembled this tutorial workbook from natural language papers that I've tried to understand. If you want to read original work, check out Sharon Goldwater's reading list on the web. Sections in this workbook are mostly from Goldwater/Griffiths 07, Post/Gildea 09, Cohn/Goldwater/Blunsom 09, etc.

I also recommend Philip Resnik's tutorial, "Gibbs Sampling for the Uninitiated". It's got much of the math that is omitted in this workbook, and it tackles a complementary set of natural language applications. The more examples you can learn from, the better.

7. Acknowledgments

I would like to thank Adam Pauls, David Chiang, and Jonathan Graehl for discussions and help. Adam patiently explained various things about Gibbs sampling, David did the segmentation example, and Jonathan did a number of experiments. Thanks also, NSF IIS-0904684.

8. Unsupervised tasks

Let's first list some unsupervised natural language problems. We'll look at some of these in depth as we go. Of course, each of these problems could also be solved by supervised techniques operating on annotated data, but what's fun about that?

Chinese segmentation.

Input: a very long sequence of Chinese characters, written naturally, with no spaces.
Output: the same sequence, broken into word-like units.

Part-of-speech tagging.

Input: a long sequence of word tokens, plus a dictionary that lists all the legal tags for each word type.
Output: a tag for each token, taking context into account.

Tree substitution grammar.
    Input: a Treebank.
    Output: a set of useful multi-level grammar rules that incorporate appropriate context. This is sort of like the Chinese segmentation problem, except on trees instead of strings.

Word alignment.
    Input: a bilingual text broken into sentence pairs.
    Output: links between pairs of words that mean the same thing.

Decipherment.
    Input: a cipher created from an original English text via a secret letter-substitution table.
    Output: the original English text.

9. Tree substitution grammar

Let's start with the problem of learning a useful grammar of English. We are given a Treebank and seek to extract a useful grammar from it. Quick personality test:

(a) Just read off the context-free productions -- there's your grammar.
(b) Treebanks are dumb! Let's learn grammar from raw text!
(c) Tell me more.

If you said (b), then you are frequently unstable in stressful situations, your youthful pained introversion has given way to a more outgoing profile that people mistake for gawkiness, and you should drink more water. We're not going to do (b). But I do love you.

If you said (c), then you already know that context-free productions don't make for good grammars. When used for analysis, probabilistic CFGs assign bad structure. When fired up in generation mode, they produce nonsense English. Look at this grammar:

S → NP VP	DT → the	VBD → put
NP → DT NNS	DT → a	VBD → sat
VP → VBD	NNS → boxes	

And look at this derivation we can generate:

S(NP(DT(a) NNS(boxes)) VP(VBD(put))) = "a boxes put"

A rule like "NP → DT NNS" is too general, because you don't know how the DT is going to expand. There are many methods for getting more context into grammars. One method is **Tree Substitution Grammar** (TSG). In addition to single-level rewrites, a TSG may have multilevel rules like "NP → DT(the) NNS". That rule is a lot less dangerous. We might decide to exclude dangerous rules like "NP → DT NNS" or "NP → DT(a) NNS" from the TSG, or just assign them very low probabilities.

The generative story of TSG is simple. You start with an S. Then you pick a rule among all the S rules (whose probabilities sum to one). Let's say you picked

rule42.	S → NP VP(VBZ(sang) PP) P(rule42 | S) = 0.002

That's a good start on a tree. There are two unfinished parts, the NP and the PP. Let's pick a rule among all the NP rules, e.g.:

rule76987.	NP → DT(the) NNS P(rule76987 | NP) = 0.14

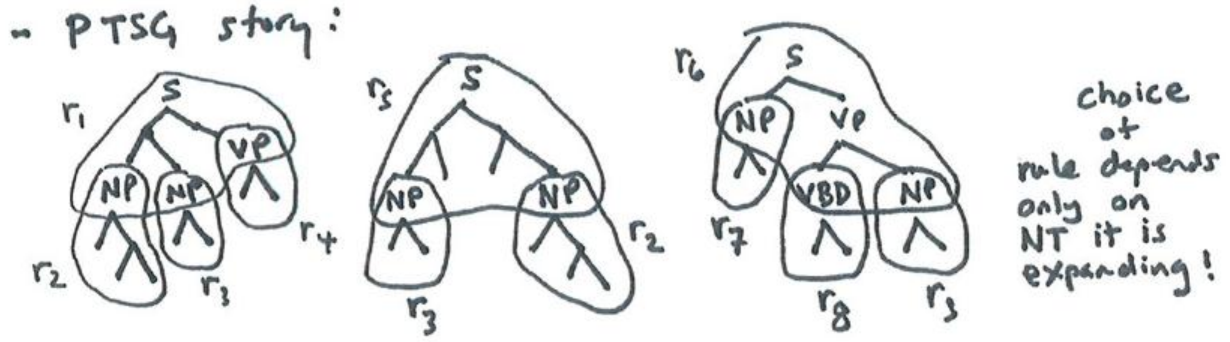
We continue like this until all non-terminals have been replaced by words. The probability of the derivation is the product of all the rule probabilities.

Notice that there are multiple ways to derive the same tree. We could have picked

rule890.	S → NP(DT(the) NNS) VP(VBZ(sang) PP) P(rule890 | S) = 0.000000001

and wound up in the same place, via a lower scoring derivation.

If we want to build a whole Treebank with this story, we can build 40,000 trees. Here's the TSG story of Treebank generation, as a picture:



With a trained TSG, we can stochastically generate new sentences. Or, we can parse a string s. To parse, we consider every derivation whose leaf sequence is s. We then output the derivation of highest probability:

$$\underset{\text{deriv with yield } s}{\text{argmax}} \; P(\text{deriv}) = \prod_{r \in \text{rule in deriv}} P(\text{rule} \mid \text{root}(r))$$

Since there are multiple ways to derive s via the same tree, some people will suggest this alternative:

$$\underset{\text{tree with yield } s}{\text{argmax}} \; \sum_{\text{deriv of tree}} P(\text{deriv}) = \prod_{r \in \text{rule in deriv}} P(\text{rule} \mid \text{root}(r))$$

Now, how do we learn a TSG? If someone had chopped up the Treebank into fragments (TSG rules) for us, then we could estimate the rule probabilities like this:

$$P(\text{rule} \mid \text{root(rule)}) = \text{count(rule)} / \sum_{r \in \text{rules with same root as rule}} \text{count}(r)$$

The problem is that you can't simply read the rules off the Treebank. Where does one rule end, and another rule start? More fundamentally, even if we learn a TSG, how do we know it's a good one? Given two TSGs, which one is better? Well, a standard thing to do is pick the one that assigns the highest likelihood to the training data (Treebank).

$$\underset{\text{TSG}}{\text{argmax}}\ P(\text{Treebank} \mid \text{TSG}) = \underset{\text{TSG}}{\text{argmax}} \prod_{\text{tree in Treebank}} \sum_{\text{deriv of tree}} P(\text{deriv}) = \prod_{r \in \text{rule in deriv}} P(\text{rule} \mid \text{root}(r))$$

EM is now ready to find the TSG that's the argmax of this expression. The inside-outside EM algorithm gives us polynomial running time, so we don't need to worry about explicitly enumerating the exponential number of derivations of a given tree.

EM works great in a lot of situations, but we have two very serious problems here. First of all, we will run out of memory as soon as we try to set up the parameter table. This table lists every rule in the TSG we are training, which means an entry for every tree fragment in the Treebank. But there are too many unique fragments, from the lowly 12,000 single-level rewrites, to the medium-size TSG rules, to the giant whole-sentence TSG rules. I've sent email to everyone I know who works with TSG, and none of them can even tell me how many TSG rules are implicit in the Treebank.

The second problem is even worse. Supposing we had the memory and could crank up EM, what model would it find? Consider these three possible solutions:

A. a TSG consisting only of the single-level rewrite rules from the Treebank.
B. a TSG consisting only of giant rules, each covering one whole Treebank sentence.
C. a "nice" TSG, where rules are smallish, but capture enough context to be reliable when used independently.

EM will try to find the model that yields the largest P(Treebank). In Cases A and B above, there is one derivation per sentence, so we can compute P(Treebank) directly.

Case A: $$P(\text{Treebank}) = \prod_{r \in \text{instances of CFG rules in Treebank}} P(r \mid \text{root}(r))$$

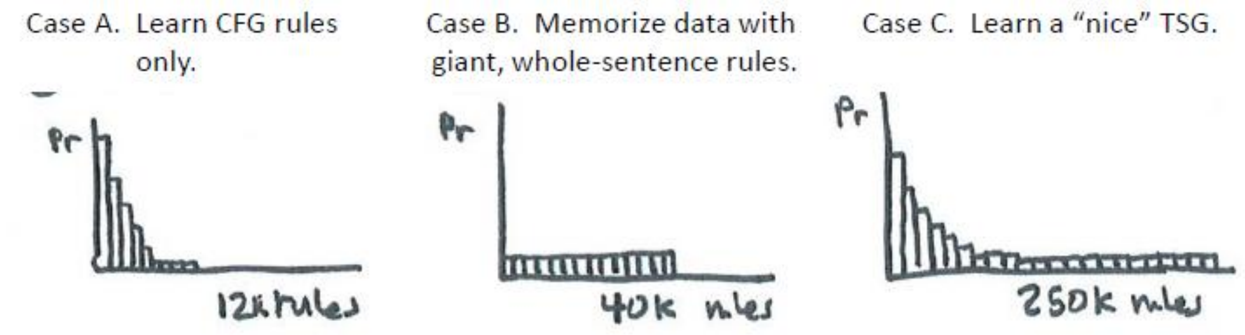
Exercise. Suppose the Treebank has only two rules, A → A A and A → a, each with probability 0.5, and further suppose that the Treebank has one million instances of each rule. What is $(0.5)^{2{,}000{,}000}$? Use Wolfram Alpha. Of course, the real Treebank has many more different kinds of rules, so the real P(Treebank) will be even lower.

Case B: P(Treebank) = $(1/40{,}000)^{40{,}000}$ (assuming Treebank has 40,000 unique sentences)

Exercise. What is $(1/40{,}000)^{40{,}000}$?

Case B is going to win, by a wide margin, and it will defeat Case C too. EM will head to Case B. It will end in tears.

Bayesian methods are going to try to help us with both problems: parameter storage and degenerate solutions. To start, let's make plots for solutions A, B, and C. Let's list rule types on the x-axis, and give rule frequencies on the y-axis:



Case A has 12,000 rules, and some of these occur extremely often. Case B has 40,000 rules, and each occurs only once. Case C has 250,000 rules, whose frequencies follow the same pattern as Case A.

Let's forget for a second whether these models assign good P(Treebank), and let's forget whether they result in good parsers or not. Take a thumbtack and pin this page to the wall across the room. Now, which looks like a grammar of English? Cases A and C look plausible. But case B looks bad. You'd expect some rules to be used more than others, some words to be used more than others. Why? Let's leave it as an exercise.

Exercise. Why do you think some words are frequent, and others are not? I mean, why aren't all words equally likely? (Full disclosure: this is a tough one).

But clearly this story of **re-use** is pervasive in natural language. Ask anyone named Zipf. If there are 42 parts of speech in English, you can bet that some are very popular (like noun), while others are less so (like superlative adjective). Some Chinese characters are popular, others not so much. If I use a certain word, or character, or part of speech, when I'm writing, there's a decent chance I'm going to re-use that word, character, or part of speech again as I write.

Case B's distribution just looks wrong.

10. Re-use, sparsity, and likelihood

Before you un-pin that paper from the far wall, let's also consider **model size**. All other things being equal, we'd like a compact (sparse) model. If you're talking to your Mum on the phone, and she asks, what rules are in your grammar, then you don't want to be on the phone all day. (Hey Natalie: Your Mum rates me!). Case A has an edge here -- the model is very compact, with only 12,000 small rules. Case B has fewer rules than Case C, but of course, the rules themselves are very big, so maybe it would take a long time to read them. All other things aren't equal, though. We'd also like a good P(Treebank) from our model. So we like three things in a model: (1) re-use of rules, (2) compact model size, and (3) good likelihood.

For now, let's focus on re-use, because EM incorrectly prefers Case B, which doesn't embody re-use.

11. New generative story

We shouldn't blame EM. It's just doing its thing. *We should blame our generative model instead*. Let's make a better generative model. First, to repeat old one:

Old generative story ("How the Treebank came to be")

Repeat 40,000 times, to build 40,000 trees:

a. Set root(rule) = S.

b. Pick a rule according to probability P(rule | root(rule)), and expand the tree with this rule.

c. Recursively pick rules to expand newly-introduced nodes, until the leaves of the tree are all words.

Notice that each rule selection depends only on the non-terminal symbol being expanded at that moment. No other context is used. Our new generative story is going to use more context for each decision. We replace step b like this:

New generative story:

b. Pick a rule according to probability

P(rule | root(rule) + all rule choices used so far in constructing the Treebank),

and expand the tree with this rule.

As our generative process builds up the Treebank, later decisions will now depend on earlier ones. Of course, adding more context is a standard thing to do when we build NLP models. The particular context here is unusually far-reaching -- it even crosses sentence boundaries! As always, we have to be careful to limit context conditioning, to avoid over-fitting. So we are going to condition rule selection on a summary of the history so far, namely the **counts of the rules** used so far in the generative Treebank construction:

P(rule | root(rule) + counts of rules used so far)

How do the counts influence the selection of the current rule? We're going to treat the history as a **cache** of rules, from which we'll select a rule according to its relative frequency in the cache.

P(rule | root(rule) + counts of rules used so far) =

$$\text{count(rule in cache)} \,/ \sum_{r \,\in\, \text{rules with same root as rule}} \text{count(r in cache)}$$

If we're trying to expand an NP, then we look back at our history of NP expansions, and we pick a rule from that list. (Note that this list contains duplicates). If we've used a particular NP rule a lot, then we're likely to select it again here. That's our re-use idea in action. This generative model is going to steer us away from Case B above, where 40,000 unique rules get generated, one after another. A dumb sequence of rule selections like that is going to get a very low probability.

The **rich get richer** under this scheme. If a rule happens to be picked several times early in the derivation process, it is more likely to be picked again later. (By the way, this is how academic prizes work – when you nominate someone for an award, you have to fill out a section called "previous awards received". Having received an award is supposed to be evidence that you are worthy of receiving an award.)

Now, how do we pick the very first rule in our Treebank derivation? Also, after that, won't we wind up picking the same rule over and over again? How do we ever get multiple types of rules into the cache? We solve these questions by adding a **base probability** for rules, called P_0.

P(rule | root(rule) + counts of rules used so far) =

$$\beta \;\; * P_0(\text{rule} \mid \text{root(rule)}) \; +$$

$$(1 - \beta) * \text{count(rule in cache)} \;/ \sum_{r \,\in\, \text{rules with same root as rule}} \text{count(r in cache)}$$

With probability β (sometimes called a **hyperparameter**), we'll generate the rule using the base distribution P_0, and with probability (1 - β), we'll use the cache. Note that the base distribution only conditions on the non-terminal being expanded, just like our TSG did, before all this cache business.

12. Base distribution

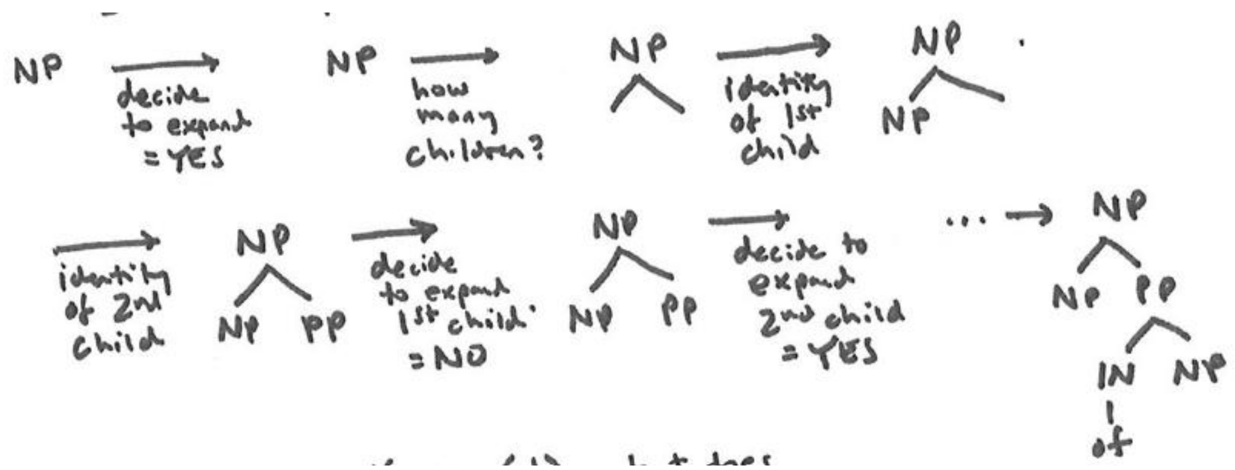
So how do we decide on the base distribution? If you ask the wrong person, they will jump up and down, and say "It can be anything you want!" Yeah! This is fun! But what do I type into my program? (By the way, and *come on*, it can't actually be anything you want.)

The simplest thing is to make the base distribution uniform. That is, when we need to expand an NP node according to P_0, then any NP rule is equally likely as any other one. How many NP rules are there in Tree Substitution Grammar? Again, the Treebank is so vast that it's difficult to list them all, and besides, someone might make up a new NP rule tomorrow. We can instead concisely capture P_0 as a little generative model of its own, one that generates TSG rules:

```
Generate-TSG-Rule(symbol)
1.  with probability δ, quit.
2.  with probability (1 - δ):
        a.  pick a number of children from a Poisson distribution (mean = λ).
        b.  pick the identities of those children from a uniform distribution over symbols.
        c.  call Generate-TSG-Rule(x) on each child x, to expand it further.
```

A symbol can be a non-terminal or an English word.

Here's an animation of Generate-TSG-Rule at work. It starts with a symbol, in this case NP, and ends with a TSG rule, in this case NP $\rightarrow$ NP PP(IN(of) NP).



Note that this model assigns higher probabilities to smaller rules. This model also assigns positive probability to an infinite number of TSG rules. We've saved a lot of RAM.

How do we decide on δ and λ? They can be anything you want! Ha, ha.

Recall, here's where we were a minute ago:

P(rule | root(rule) + counts of rules used so far) =
β * P_0(rule | root(rule)) +
$(1 - \beta)$ * count(rule in cache) / $\sum_{r \in \text{rules with same root as rule}}$ count(r in cache)

If we fire this model up in "Treebank generation" mode, it'll choose the first rule according to the base distribution, and then it'll choose each subsequent rule according to either (a) the base distribution, or (b) the cache, which we're building up over time.

13. Larger caches considered more reliable

All we have done is to create a new generative story for Treebank generation. Let's make one further change to the story. Let's imagine that as we build up more and more Treebank – i.e., a larger and larger cache – then we want to bias ourselves more towards using the cache, and less towards using the base distribution P_0. There are lots of ways we could accomplish that. Here's just one, where we replace β with (α/(α+H)), where H is the number of times that our non-terminal root(rule) has been expanded in the history of the cache:

P(rule | root(rule) + counts of rules used so far) =
$(\alpha/(\alpha+H))$ * P_0(rule | root(rule)) +
$(1 - \alpha/(\alpha+H)))$ * count(rule in cache) / $\sum_{r \in \text{rules with same root as rule}}$ count(r in cache)

You can verify that if the history H is short, then the base probability will be preferred, but if the history is long, then the cache model will be preferred. Instead of β between 0 and 1, we now have α between 0 and infinity. α is sometimes called a **concentration parameter**.

If you think (α/(α+H)) is an arbitrary way to bias more toward larger caches, or if you think biasing toward larger caches is itself an arbitrary move, then please hold that thought -- we'll return to it in Sections 31 and 32.

We can re-write (1 - (α/(α+H))) as (H/(α+H)) to get:

$$P(\text{rule} \mid \text{root(rule)} + \text{counts of rules used so far}) = \mathbf{(\alpha/(\alpha+H))} * P_0(\text{rule} \mid \text{root(rule)}) + \mathbf{(H/(\alpha+H))} * \text{count(rule in cache)} \,/ \sum_{r \,\in\, \text{rules with same root as rule}} \text{count(r in cache)}$$

Exercise. Use eighth-grade math to show that (1-(α/(α+H))) = (H/(α+H)).

Exercise. What happens if H=0, i.e., if we are generating the first rule of the Treebank?

Finally, we can simplify the last formula a bit more, since the numerator H is the same as the denominator sum:

$$\mathbf{P(\text{rule} \mid \text{root(rule)}) = \frac{\alpha * P_0(\text{rule} \mid \text{root(rule)}) + \text{count(rule in cache)}}{\alpha + \text{count(root(rule) expanded in cache)}}}$$

Exercise. Use eighth-grade math to derive this.

Exercise. Draw a $\boxed{\textbf{box}}$ around that last formula. It's critical! We'll refer to it as the **boxed formula**. Go ahead, draw the box now.

The boxed formula is our new generative story, and it describes a cache model. In fact, it's completely nailed down, except for how we decide on α. As you no doubt suspect, α can be anything! If α = 0, we always use the cache. If α = 1,000,000,000, we almost always use the base distribution. Let's set it to 42.

When you read natural language papers these days, you often see the boxed formula on the first page of the paper. The authors assume you are already familiar with it. To get that formula to where it made common sense to me, I back-fitted all the rest of what you just read.

An aside: People often describe the cache model as a **Chinese Restaurant Process**. The restaurant has an infinite number of round tables, each of which accommodates an infinite number of customers. Suppose there are H customers already sitting down at various tables, and a new customer walks in. With probability α/(α+H), she starts a new table, she uses P_0 to label the table with some rule, and she yells out that rule. Otherwise, she randomly picks an already-seated customer, sits at his table, and yells out that table's rule. Notice that as a table collects more customers, it becomes an ever more attractive target. All the yelling creates our rule sequence. For the TSG problem, there are actually several restaurants, one called “Delicious NP Expansions”, one called “Delicious VP Expansions”, etc. Depending on the non-terminal being expanded, the customer goes to whatever restaurant is appropriate. One thing I don’t understand about the Chinese Restaurant Process is why the tables have to be round. For those who like more violent analogies, the **Stick-Breaking Process** is an alternative.

14. We’re done?

We have our new generative model, and we’re ready to train it on the Treebank. We fixed the base distribution P_0. We decided on values for α, δ, and λ. Now all we have to do is learn the other parameters.

What other parameters? There are none! Remember the huge P(rule | root(rule)) EM parameter table that we couldn’t store? Well, we certainly have saved ourselves a lot of RAM. (Hmm, powerful computer-chip manufacturers will not be happy, and they may seek revenge. Bayesian researchers may start to disappear mysteriously. Say, Hal Daume III hasn’t blogged in several days …)

How can we possibly be done? With our old generative model, we could generate 40,000 trees stochastically, by consulting the vast P(rule | root(rule)) table, but we never actually stored or learned that table. With our new generative model, on the other hand, we can already generate 40,000 trees stochastically, anytime we want. Just fire it up! Select the first S rule for sentence 1, according to the base distribution. Now select the second rule, according to the boxed formula. Now select the third rule … et cetera, et cetera. Once the 40,000th tree is completed, stop and return the probability of the derivation. If we fire it up again, we’ll get a different set of 40,000 trees, and a different derivation probability.

Hmm. Shouldn’t we fiddle with some parameters, in order to “fit” the model to the Treebank? Shouldn’t we search for parameter values that maximize the likelihood of the Treebank? Well, we do have α, δ, and λ to play with … but learning three numbers from the giant Treebank seems silly. If we want a better P_0, then we’re back to square one, i.e., learning a nice TSG from an un-segmented Treebank.

15. Using the cache model

If the cache model is really finished, then we can answer the questions below. Set aside the question of efficiency for a minute.

- What derivations produce the observed Treebank, and what are their derivation probabilities, according to the cache model? Given a derivation, we can get its probability by mechanically applying the boxed formula to it.

- What is the sum of the probabilities of all derivations? That is, what is P(Treebank)?

- Of all derivations that produce the Treebank, which has the highest probability, according to the cache model? This is the Viterbi derivation. What is P(Viterbi-derivation)?

- Considering all derivations of the Treebank, what share of the probability mass does the Viterbi derivation possess? This will be P(Viterbi-derivation) / P(Treebank).

- Given all derivations of the Treebank, each with its own probability, what are the expected counts of all TSG rules? (At least for rules with expected counts greater than some threshold).

In fact, we can do all that, using the boxed formula! If we get TSG rules with expected counts, then we can use them to parse new sentences. So we really are done, except for figuring out how to answer the questions above more efficiently. We'll come back to that.

16. Add-one smoothing

Have you ever done **add-one smoothing**? If you have, then a man named Ken Church is very angry at you. Avoid him.

Say you have a sequence of N part-of-speech tags in some language. The tagging guidelines list 42 distinct tags. You only observe 39 of these in your data. How do you estimate the probability of a given tag, such as OJ? You could estimate P(OJ) = count(OJ) / N, but this would be zero if OJ was not among the 39. You don't really think the chance of OJ is zero, so you whip out some add-one smoothing:

P(OJ) = count(OJ)+1 / (N+42)

The "add-one" part is visible in the numerator, and the denominator is adjusted so that all the P(tag) values still sum to one. Now let's view your sequence of tags as a cache and try to predict what the next tag might be. Remember that our cache model says:

P(OJ) = **$\alpha/(\alpha$+N**) * P_0(OJ) + **(1 - $\alpha/(\alpha$+N))** * count(OJ)/N

Let's set the base probability $P_0(x)$ to be uniform, equaling 1/42 for all x. Let's also set α to be 42. Then:

P(OJ) = count(OJ)+1 / (N+42)

Exercise. Use eighth-grade math to derive this.

Wow, so add-one smoothing is somehow related to the cache model, at least for some particular choices of P_0 and α. We've also seen that 42 can indeed be a reasonable setting for α!

Exercise. What happens to P(OJ) if α=0? (i.e., always use the cache)

Exercise. In a futile effort to placate Ken Church, some people do add-one-half smoothing, instead of add-one smoothing. Can you simulate this with a different choice of α?

17. Chinese segmentation

Let's switch to another application and see some actual results. We observe a long sequence of Chinese characters written without spaces, as is the custom. We'd like to divide it up into words, for use in indexing, language modeling, and translation modeling. A Chinese word is itself a short sequence of characters. We'd also like to learn an efficient segmenter for dividing up new, previously unseen Chinese sequences into words.

Our old-style generative story for EM ("How the stream of characters got there") would go like this:

```
1. Generate a Chinese word according to a finite table of unigram word probabilities p(w)
2. Write down the word just chosen.  Do not hit the space bar!
3. With probability 0.99,
      go to step 1 (and continue outputting words)
   With probability 0.01, quit.
```

Starting with uniform probabilities, EM training will work poorly here. It will conclude that the 1,000,000-character training sequence is actually just one big Chinese word.

By contrast, our new cache-based generative story goes like this:

```
0. H=0.   /* H is the length of the history, the number of decisions taken so far */
1. With probability $\alpha/(\alpha+H)$, generate a Chinese word according to the base distribution $P_0$
   With probability $H/(\alpha+H)$, generate a Chinese word using the cache of words generated so far
2. Write down the word just chosen.
3. With probability 0.99,
      $H \leftarrow H + 1$
      go to step 1 (and continue outputting words)
   With probability 0.01, quit.
```

This model assigns high probability to derivations that re-use Chinese words. When we work through the cache model mathematics (remember the boxed formula!), we see that the probability of a particular derivation that generates word sequence $w_1 \ldots w_n$ is:

$$\prod_{i=1}^{n} \frac{\alpha\, P_0(w_i) + \text{count}(w_i \text{ in i-1 previous word selections})}{\alpha + (i - 1)} * 0.99^{n-1} * 0.01$$

Note that this formula scores a *word* sequence, not a *character* sequence. The probability of a character sequence is something different. Don't get confused.

Let's pick $\alpha=1$. For the base distribution P_0 that generates individual words, let's use this **micro-story** (assume there are C total Chinese characters):

```
0. Word = empty
1. Pick a Chinese character from the uniform distribution (1/C)
2. Add the chosen character to the end of Word
3. With probability 0.5, go to 1
   With probability 0.5, quit and output Word.
```

This base distribution prefers short words to long words, though it assigns positive probability to an infinity of arbitrarily long words.

Exercise. Pretend there are only three distinct Chinese characters in the world: a, b, and c. What probabilities does the base distribution assign to these words: a, aa, aaa, bcb?

Let’s say we observe a very short Chinese character sequence: ab. According to our generative story, there are two ways this sequence could have gotten produced. Under one derivation, the word “ab” was generated from the base distribution, and then the **quit** decision was taken. The probability of this derivation is:

$$\frac{\alpha * \mathbf{P_0(ab)} + 0}{\alpha + 0} * 0.01 = \mathbf{(1/3 * 1/2 * 1/3 * 1/2)} * 0.01 = 0.00028$$

Under the other derivation, the single-character word “a” was generated, then the **continue** decision was taken, then the word “b” was generated, then the **quit** decision was taken:

$$\frac{\alpha * P_0(a) + 0}{\alpha + 0} * 0.99 * \frac{\alpha * P_0(b) + 0}{\mathbf{\alpha + 1}} * 0.01 = (1/3 * 1/2) * 0.99 * (1/3 * 1/2) \mathbf{/ 2} * 0.01 = 0.00013$$

Exercise. Why is there a “+ 1” in the second denominator?

Each derivation corresponds to a segmentation of the observed sequence “ab”. The reason their probabilities don’t sum to one is because our generative story can produce lots of other strings besides “ab”. If we normalize these two derivations’ probabilities, however, we get:

P(derivation-ab | ab) = 0.00028 / (0.00028 + 0.00013) = 0.68 /* winner */
P(derivation-a-b | ab) = 0.32

Notice that we got these numbers without any iterative, EM-style training. Magic! Now let’s say we instead observe the sequence “aa”. Again, there are two possible derivations. The probability of the first is:

$$\frac{\alpha * P_0\,(aa) + 0}{\alpha + 0} * 0.01 = (1/3 * 1/2 * 1/3 * 1/2) * 0.01 = 0.00028$$

while the probability of the second is:

$$\frac{\alpha * P_0\,(a) + 0}{\alpha + 0} * 0.99 * \frac{\alpha * P_0\,(a) \mathbf{+ 1}}{\alpha + 1} * 0.01 = (1/3 * 1/2) * 0.99 * ((1/3 * 1/2) \mathbf{+ 1}) / 2 * 0.01 = 0.00096$$

Exercise. Why is there a “+ 1” in the second numerator?

After normalizing, we get:

P(derivation-aa | aa) = 0.23
P(derivation-a-a | aa) = 0.77 /* winner */

A different result this time -- now the model prefers to imagine that the sequence is composed of two separate word tokens (“a” and “a”), rather than a single multi-character word (“aa”). The model rewards re-use.

Exercise. How many derivations does the sequence “abab” have?

Exercise. For three derivations of “abab” (namely: abab, a-b-a-b, and ab-ab), compute un-normalized P(derivation) scores. Which scores highest? Here are a couple of other derivations with scores you can confirm:

derivation	P(derivation)
abab	??
a-b-a-b	??
ab-ab	??
aba-b	0.0000038 (you can confirm this)
a-b-ab	0.0000013 (you can confirm this)

Exercise. Repeat the last exercise with some α other than 1. What do you get?

We can contrast the cache model’s behavior with that of plain EM under the old-style generative model. With plain EM, we don’t have an α or a base distribution. We have the following parameters:

p(a) = ?
p(b) = ?
p(ab) = ?
p(aa) = ?
…
p(ababba) = ?
…
p(continue) = 0.99
p(quit) = 0.01

With training data like T = “ababba”, EM will fit the parameters to the data by converging to p(ababba) = 1.0, and setting all the other word unigram probabilities to zero. That will make the likelihood of the data P(T) = p(ababba) * p(quit) = 0.01, which is unbeatable via any other parameter settings. In other words, EM will analyze the whole sequence as a single word, every time. Another way of saying this is that EM is very bad at mediating **big rule versus small rule** competition – big rules always win.

The cache model gives more interesting behavior. (But should α=42?)

Unfortunately, we can’t yet crank up our cache model on a very long Chinese string. That’s because we can’t physically list out the exponential number of derivations and find the highest-scoring one, much less use all of the derivations to compute expected word counts. There are too many derivations. With our old generative story (no cache, no long-ranging context), we were able to use some nice dynamic programming solutions (Viterbi algorithm, forward-backward algorithm, etc). Don’t worry, we’ll get there. For now, let’s drop Chinese segmentation and push on to a third and final application, part-of-speech tagging. There we will finally resolve the efficiency question.

18. Part-of-Speech Tagging

We are given an English sentence and want to assign a **part-of-speech** (NN=noun, VB=verb, etc) to every word token. Words are ambiguous, so we have to take context into account. One unsupervised set-up for this problem is that we are given (1) a large sample of un-annotated English, and (2) a dictionary that gives all legal tags for each word type. An extract from the dictionary looks like this:

comment:	VB, NN
way:	NN, RB
live:	VB, VBP, JJ, RB
platypus:	NN

We want to use the dictionary (2) in order to tag the text (1), and we'll evaluate how good the resulting tagging is, against a gold standard. One strategy is to pick, for each word token, a random tag from its list of legal tags. This gives 65% accuracy. We can do better with unsupervised training. Suppose we observe this sentence:

The platypus made the comment.

Since "platypus" is unambiguously NN, we can deduce that NN sometimes follows DT (assuming we know that "the" is a DT here). We might then tentatively label "comment" as NN rather than VB, since it also follows DT. Of course, "the" is also ambiguous – believe it or not, it appears with six different tags in the Penn Treebank. So we can make progress, but there are a lot of intertwined inferences we need to make.

EM to the rescue!

Generatively speaking, we ask "How did the raw English corpus get here?" We imagine someone produced a sequence of part-of-speech tags, according to a grammar, and then they converted each tag into a word, independent of context. Stranger things happen in that head of yours. Let t be a tag sequence, and let w be a word sequence, both of length n. Assuming a bigram tag grammar, our old-style generative model looks like this:

$$P(t, w) = P(t)\, P(w \mid t) = \prod_{i=1}^{n} P(t_i \mid t_{i-1}) * \prod_{i=1}^{n} P(w_i \mid t_i)$$

EM can learn the tag bigram probabilities $P(t_i \mid t_{i-1})$ and the dictionary probabilities $P(w_i \mid t_i)$ like this:

```
0.  Assume uniform $P(t_i \mid t_{i-1})$ and $P(w_i \mid t_i)$
1.  For every derivation of w (conceivable tag sequence), compute P(derivation) using the formula above
2.  Normalize those values to get P(derivation | w) for each derivation
3.  Collect fractional counts off each derivation, weighted by P(derivation | w)
4.  Normalize counts to get new values for $P(t_i \mid t_{i-1})$ and $P(w_i \mid t_i)$
5.  Unless tired, go to 2
```

This algorithm is exponential as written, because step 1 enumerates the 42^n unique derivations. But there is a linear-time algorithm ("forward backward") that accomplishes the same thing with dynamic programming. With the values trained by EM on a sequence of 24,000 words, the Viterbi tag sequence is about **81% correct**.

By contrast, our new-style cache model looks like this:

$$P(t, w) = \prod_{i=1}^{n} \frac{\alpha_1 * P_0(t_i \mid t_{i-1}) + \text{count}(\text{“}t_{i-1}\, t_i\text{” in cache})}{\alpha_1 + \text{count}(\text{“}t_{i-1}\text{” in cache})} * \prod_{i=1}^{n} \frac{\alpha_2 * P_0(w_i \mid t_i) + \text{count}(\text{“}t_i\ w_i\text{” in cache})}{\alpha_2 + \text{count}(\text{“}t_i\text{” in cache})}$$

Here we have one α for the tag bigrams and a different α for the dictionary.

Exercise. Compare this equation to the boxed formula and make sure you understand it.

If we pick certain values for P_0, α_1, and α_2, we can simplify the formula some more. I'm not sure why simplifying the formula will lead to better a natural language processing system, but we should try to get into the spirit of this.

Let's set $P_0(t_i \mid t_{i-1}) = 1/T$, where T is the total number of distinct tags (NN, VB, IN …), say 42. Let's set $P_0(w_i \mid t_i) = 1/W$, where W is the total number of distinct words, say 50,000. If we do that, we can eliminate P_0:

$$P(t, w) = \prod_{i=1}^{n} \frac{\beta_1 + \text{count}(\text{“}t_{i-1}\, t_i\text{” in cache})}{\beta_1\, T + \text{count}(\text{“}t_{i-1}\text{” in cache})} * \prod_{i=1}^{n} \frac{\beta_2 + \text{count}(\text{“}t_i\ w_i\text{” in cache})}{\beta_2\, W + \text{count}(\text{“}t_i\text{” in cache})}$$

Exercise. What values of β_1 and β_2 do we need to pick to get this simplification?

How do we train this model? Simple: we collect expected counts for the tag bigrams and tag/word pairs, and we normalize those counts. How do we get the expected counts? We just run steps 1-4 as above:

1. For every derivation of w (conceivable tag sequence), compute P(derivation) using the formula above
2. Normalize those values to get P(derivation | w) for each derivation
3. Collect fractional counts off each derivation, weighted by P(derivation | w)
4. Normalize counts to get new values for $P(t_i \mid t_{i-1})$ and $P(w_i \mid t_i)$

No need for any messy iterations. We could even simply grab the highest-scoring derivation and collect whole counts off of that – again, no messy iterations.

Tagging accuracy for the cache model is in the **85% range**, so the model that rewards re-use wins out (85% > 81% for plain EM). If you are pretty psyched about that, then check this out: with a *trigram* tag model, EM's accuracy is actually **75%**, worse than bigram. EM abuses all those extra parameters and does weird things. The cache model, on the other hand, doesn't require the designer to make such careful choices.

Unfortunately, we can't use the forward-backward trick for cache-model training, because P(t, w) has long-ranging dependencies. The probabilities associated with labeling the thousandth word as NN will depend on (the counts of) all the choices made on the first 999 words. Okay, let's keep working on this.

19. Sampling

We have so far assumed that we should enumerate *all* corpus derivations (either explicitly, or implicitly via dynamic programming), and that we should collect fractional counts over events in all derivations. Well, changes are coming, my friend:

enumerate	→ sample
all derivations	→ some derivations
fractional counts	→ whole counts

We're going to generate derivations by pulling them one by one from a hat. After we sample a derivation, we put it back in the hat. This is **sampling with replacement**. We rig things so that the probability of pulling a certain derivation from the hat is proportional to P(derivation). Furthermore, when we pull an alignment out of the hat, we collect **whole counts** from it. Finally, for efficiency, we only sample some derivations, not all.

Suppose there were two derivations consistent with our observed data:

P(derivation1) = 0.001
P(derivation2) = 0.002

Therefore:

P(derivation1 | observed-data) = 1/3
P(derivation2 | observed-data) = 2/3

With EM, we'd collect events off derivation1 and toss them into our count table with weight 1/3, and likewise for derivation2, with weight 2/3. Then we'd normalize our count table.

With sampling, we randomly draw a derivation (with probabilities proportional to 0.001 and 0.002), collect whole counts from that derivation, and repeat. The "random draw" will retrieve derivation2 two-thirds of the time, and derivation1 one-third of the time. After we've sampled some number of times, we normalize our count table.

Exercise. Convince yourself that both methods yield the same results, supposing enough samples are drawn.

20. Gibbs sampling

How do we choose sample derivations, and how do we ensure that we pull derivations in proportion to their P(derivation) scores? One answer is **Gibbs sampling**. Here's the general plan:

1. Start with some initial sample (e.g., a random tagging of the corpus).

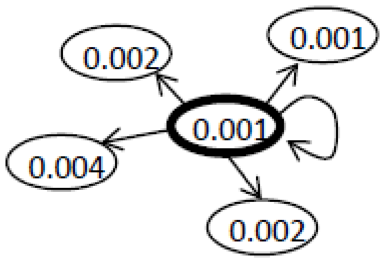
2. Make some **small change** to the sample (e.g., change the current tag of a single word to one of NN, VB, IN, …, or RB – the word's current tag included), by weighted coin flip. The probability with which we make the change should be proportional to the entire P(derivation) with the change.

3. Collect whole counts off the new sample (which might be the same as the old sample, if the word's tag didn't change).

4. Until tired, go to 2

So we wander around drunk, hitting one sample after another, perhaps walking in circles now and then. Notice, however, that the choice at step 2 is not *uniformly* random. Changing a word's tag to NN may yield a high P(derivation), while changing that same word's tag to VB may yield a low P(derivation). Because the choice at step 2 is not uniform, we ultimately wander over to where the good whole-corpus derivations are, and we ultimately spend more time there, collecting counts. Although, if we wander around long enough, we'll spend at least a little time everywhere.

Here's a picture. The circles are derivations. Each derivation represents a whole-corpus tag sequence. P(derivation) values are shown inside.



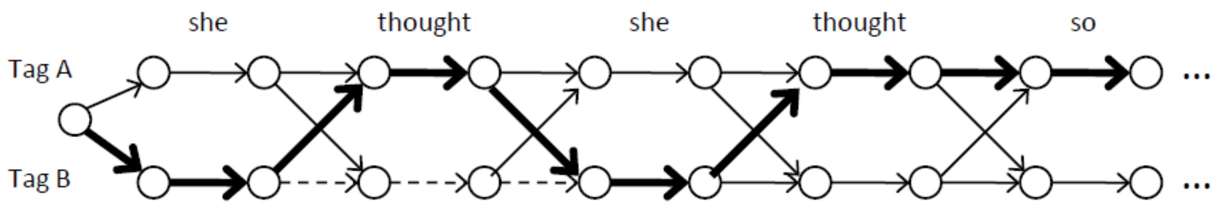
The bold circle is the current sample. Pretend we are considering re-tagging the 74th word. There are 42 ways to change the sample, 5 of which are shown in the picture (including **no change** – that's the little loop on the right). To execute step 2 above, we flip a coin. There is a 4/10 chance of moving to the lower-left sample, there is a 1/10 chance of moving to the upper-right sample, etc. After we make this change, we collect whole counts off the new sample, and repeat. When we repeat, we may consider re-tagging the 75th word instead of the 74th word, which would yield a new set of neighboring derivations and a chance to explore out further. How many samples do we explore altogether? Thousands or millions.

Note that the small change is user-defined and problem-specific. This is annoying. Plain EM doesn't require so much of your personal involvement.

21. Scoring derivations

The inefficient part of the previous algorithm is scoring the neighboring derivations. We have nice formulas for P(derivation), for both the cache model and the old-style (no cache) model. But we can't afford to calculate a whole-corpus P(derivation) from scratch if our data set is very large, certainly not 42 times at each of the millions of sampling steps we take. We want an **incremental scoring** scheme. We compute the very first P(initial-derivation) from scratch, but after that, since neighboring samples are so similar-looking, we'd like to compute P(new-derivation) as a quick-to-compute modification of P(current-derivation).

Here we have a picture of our tagging problem. The word sequence is shown across the top ("she thought she thought so …"). Possible tags are shown at the left (here, just A and B). Paths through the network represent derivations that are consistent with the word sequence.



Consider the bold path, our initial sample derivation. This sequence of bold arrows represents these decisions:

Choose tag B given START	START→B
Choose word "she" given tag B	B→she
Choose tag A given tag B	B→A
Choose word "thought" given tag A	A→thought
Choose tag B given tag A	A→B
Choose word "she" given tag B	B→she
Choose tag A given tag B	B→A
Choose word "thought" given tag A	A→thought
Choose tag A given tag A	A→A
Choose word "so" given tag A	A→so
etc.	

Notice that we have interleaved tag-bigram choices with word-selection choices. Our old-style generative model would assign to the following score to this path.

P(derivation) = P(B | START) * P(she | B) * P(A | B) * P(thought | A) * P(B | A) * P(she | B) *
P(A | B) * P(thought | A) * P(A | A) * P(so | A) …

Let's see how Gibbs sampling works, first with the old-style generative model.

Note: Of course, with the old-style model, the forward-backward algorithm is the way to go. But in a universe where the forward-backward algorithm doesn't exist … whoa! … sampling might be a reasonable thing to do. This is just a warm-up -- we'll soon do Gibbs sampling with the cache model.

We pick a word, such as the second word token, "thought". It is tagged as "A" in the current (bold) sample derivation. We consider how to re-tag it. We could either switch its tag to "B", changing our derivation to include the dotted arrows, or we could leave its tag as "A". We need to execute a weighted coin flip to select the next sample. Suppose we already know P(current-derivation) for the current sample. Let P(new-derivation) reflect the small modification in which we flip the tag of the second word token from "A" to "B". Then:

$$\text{P(new-derivation)} = \text{P(current-derivation)} * \frac{P(B \mid B)}{P(A \mid B)} * \frac{P(\text{thought} \mid B)}{P(\text{thought} \mid A)} * \frac{P(B \mid B)}{P(B \mid A)}$$

Exercise. Verify that this is correct.

So even if our sequence is 1,000,000 words long, we can score neighboring derivations very quickly! With that in mind, we select our new sample like this:

$$\text{with probability } \frac{\text{P(current-derivation)}}{\text{P(current-derivation) + P(new-derivation)}} \text{ , choose bold path (again).}$$

$$\text{with probability } \frac{\text{P(new-derivation)}}{\text{P(current-derivation) + P(new-derivation)}} \text{ , choose new path with dotted arrows.}$$

If we had 42 parts of speech (instead of just "A" and "B"), we'd make a 42-way decision to select the next sample.

Note: P(current-derivation) and P(new-derivation) are not the chances with which the weighted coin flip will move to a new sample. These must be normalized against each other first. We move to a new sample with probability *proportional to* P(derivation).

Exercise. On any given night, Jerzy eats Italian food with 95% probability (he opens a thick dictionary at random, and if the last two digits of the page number are less than 95, then it's pasta). What's the chance he eats Italian every day of the year? Call that X. What's the chance Jerzy eats Italian every day of the year, except March 1? Call that Y. How much more likely is X than Y? What's a short incremental formula to get Y from X?

22. Whole counts

Recall that we collect whole counts off each sample we encounter. Is there also an incremental scheme for updating our count table? Yes. The idea is to note that the new sample's counts differ only slightly from the old sample's counts:

one less count for (A→thought)
one less count for (B→A)
one less count for (A→B)
one more count for (B→thought)
two more counts for (B→B)

Exercise. Why *two* more counts for (B→B)?

Exercise. Show how to cheaply maintain a count table as we sample. Hint: time-stamp the incremental changes and "settle up" next time that same count table entry is incrementally changed. This is referred to as the "perceptron trick". (Full disclosure: this exercise is harder than the others.)

23. Incremental scoring for the cache model

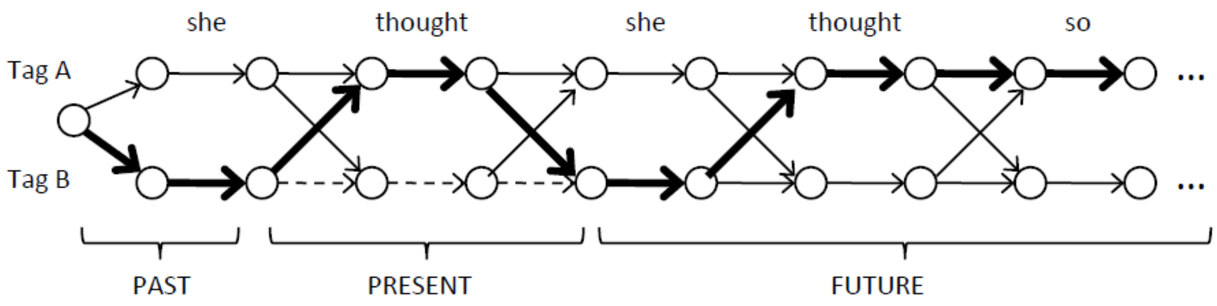
So much for the old-style generative model. Let's move to the cache model. Recall from Section 18:

$$P(t, w) = \prod_{i=1}^{n} \frac{\beta_1 + \text{count}(\text{“}t_{i-1}\, t_i\text{” in cache})}{\beta_1 T + \text{count}(\text{“}t_{i-1}\text{” in cache})} * \prod_{i=1}^{n} \frac{\beta_2 + \text{count}(\text{“}t_i\ w_i\text{” in cache})}{\beta_2 W + \text{count}(\text{“}t_i\text{” in cache})}$$

For this model, we'd also like:

P(new-derivation) = P(current-derivation) * <SSIF>

where <SSIF> stands for **some short incremental formula**. Here again is our current sample (in bold) and the little switch we'd like to make (dotted lines):



The two samples share the same PAST links and link-costs, so let's not worry about the PAST. For the PRESENT, we can surely devise some incremental scoring formula – no problem. However, the FUTURE is terrible. The two samples share the same FUTURE *links*, but if we change the PRESENT, we unwittingly change the *costs* of those FUTURE links. That's because the same FUTURE decisions in the new sample are operating under a **different cache** than they were in the old sample.

Argh! Things were going so well.

Exercise. Thomas is figuring out where to eat dinner. If he's feeling playful, he flips a coin: heads for Italian, tails for Japanese. If he's *not* feeling playful, he looks back at the year to date and just uses his experience as a guide. That is, if he's eaten Italian 82% of the days so far, then he'll eat Italian tonight with an 82% probability. A couple more points: (1) Thomas always eats Italian on January 1st, and (2) Thomas only feels playful on Fridays. (*Are you with me, reader? If you are zoning out, you might be getting hungry. If you are zoning in, you might think that Thomas should feel less and less playful as the year wears on!*) Anyway … what's the chance Thomas eats Italian every day for a whole year? Call that X. What's the chance he eats Italian every day for a whole year except for March 1? Call that Y. That should be a little painful to compute, and hmm, it's not so easy to get to Y from X incrementally, is it?

24. Exchangeability

"The factors are **exchangeable**!" People will tell you that. I personally believe that "exchangeable" is very hard to spell. I typed the word for the first time just now, and it took me three tries. Here are the Google counts: exchangible (327), exchangable (189,000), exchangeable (5,000,000). Well, let's just forge ahead.

First of all, let's separate out the **word/tag events** from the tag-bigram events. The cache of tag bigram events is irrelevant for a given word/tag event. Only the cache of word/tag events is relevant. Here are the word/tag events in our current tagging sample:

PAST	PRESENT	FUTURE
(B→she)	(A→thought)	(B→she) (A→thought) (A→so)

The portion of P(derivation) contributed by these events is:

$$\prod_{i=1}^{n} \frac{\beta_2 + \text{count}(\text{“}t_i /w_i\text{” in cache})}{\beta_2 W + \text{count}(\text{“}t_i\text{” in cache})} = \frac{\beta_2 + 0}{\beta_2 W + 0} * \frac{\beta_2 + 0}{\beta_2 W + 0} * \frac{\beta_2 + 1}{\beta_2 W + 1} * \frac{\beta_2 + 1}{\beta_2 W + 1} * \frac{\beta_2 + 0}{\beta_2 W + 2}$$

Exercise. What do the integers in the numerators and denominators refer to?

Now for the big trick. Let's move (exchange) the PRESENT to the very end of the sequence:

PAST	**+**	**FUTURE**	**PRESENT**
(B→she)		**(B→she) (A→thought) (A→so)**	**(A→thought)**

It turns out that this doesn't change the score. The new score is:

$$\prod_{i=1}^{n} \frac{\beta_2 + \text{count}(\text{“}t_i /w_i\text{” in cache})}{\beta_2 W + \text{count}(\text{“}t_i\text{” in cache})} = \frac{\beta_2 + 0}{\beta_2 W + 0} * \frac{\beta_2 + 1}{\beta_2 W + 1} * \frac{\beta_2 + 0}{\beta_2 W + 0} * \frac{\beta_2 + 0}{\beta_2 W + 1} * \frac{\beta_2 + 1}{\beta_2 W + 2}$$

In general, the terms will change. But while the numerators and denominators shift around, we still have the same bag of numerators and the same bag of denominators. So the overall score doesn't change. This is true no matter how big FUTURE is. We state this without proof.

Exercise. Noodle this thing around until it becomes clear.

We can use this result to do incremental scoring of neighboring samples. Let's say we want to re-tag the second word from "A" to "B". We set about constructing that new derivation sequence from our current derivation:

1. Start with current sample. We already know its score to be P(current-derivation).

2. Move PRESENT to end. The score is still the same.

3. Remove (A→thought) from the end. It's easy to add or remove stuff at the end of a fixed **PAST+FUTURE** sequence, if we have the counts of PAST+FUTURE events. The score is now:

$$\text{P(current-derivation)} * \frac{\beta_2 W + \text{count(“A” in PAST+FUTURE)}}{\beta_2 + \text{count(“(A}\rightarrow\text{thought)” in PAST+FUTURE)}}$$

4. Add (B→thought) to the end. This completes the construction of our new derivation. The score of P(new-derivation) is:

$$\text{P(current-derivation)} * \frac{\beta_2 \text{ W + count(“A” in PAST+FUTURE)}}{\beta_2 \text{ + count(“(A}\rightarrow\text{thought)” in PAST+FUTURE)}} \; \frac{\beta_2 \text{ + count(“(B}\rightarrow\text{thought)” in PAST+FUTURE)}}{\beta_2 \text{ W + count(“B” in PAST+FUTURE)}}$$

5. Move PRESENT back to the middle of the derivation, to where it was before. This doesn't change the score.

To summarize, we created a new derivation that re-tags the second word from “A” to “B”, and we demonstrated a short incremental scoring formula for obtaining the probability of this new derivation, given the probability of the old derivation. That's the magic, baby.

You might have noticed that in step 3, it says “if we have the counts of PAST+FUTURE events”. No problem. For current samples, we not only maintain P(derivation), but we also maintain event counts for the whole derivation. You can think of these as PAST+PRESENT+FUTURE counts. If we're about to re-sample the tag for a particular PRESENT word, then we can get PAST+FUTURE counts by subtracting PRESENT counts from our always-maintained PAST+PRESENT+FUTURE counts.

25. Tag-bigram events

What about the **tag-bigram events**? We can do the same trick, exchanging them around. Here, we are actually modifying two bigrams when we change a tag in the PRESENT. To get from our current derivation to the new derivation, we again take the following conceptual steps.

1. Current derivation:

PAST	|	PRESENT	|	FUTURE
(START→B)	|	(B→A) (A→B)	|	(B→A) (A→A)

2. Move PRESENT to end:

PAST	+	FUTURE	|	PRESENT
(START→B)		(B→A) (A→A)	|	(B→A) (A→B)

3. Remove PRESENT:

PAST	+	FUTURE	|
(START→B)		(B→A) (A→A)	|

4. Add new derivation's PRESENT:

PAST	+	FUTURE	|	NEW PRESENT
(START→B)		(B→A) (A→A)	|	(B→B) (B→B)

5. Move new PRESENT back to middle:

PAST	NEW PRESENT	FUTURE
(START→B)	(B→B) (B→B)	(B→A) (A→A)

We only need to determine the cost of removing "(B→A) (A→B)" from the end of the PAST+FUTURE cache, and the cost adding "(B→B) (B→B)" to the end of the PAST+FUTURE cache. Let's look at the latter, just step 4 above. You might think that the incremental scoring factors would be something like this:

$$\text{current-score} * \frac{\beta_1 + \text{count(“(B→B)” in PAST+FUTURE)}}{\beta_1 T + \text{count(“B” in PAST+FUTURE)}} * \frac{\beta_1 + \text{count(“(B→B)” in PAST+FUTURE)}}{\beta_1 T + \text{count(“B” in PAST+FUTURE)}}$$

That's not right! Deciding to add the first "(B→B)" event **actually modifies the cache** for the second "(B→B)" event. So we have to track the cache even while inside PRESENT. The correct formula is:

$$\text{current-score} * \frac{\beta_1 + \text{count(“(B→B)” in PAST+FUTURE)}}{\beta_1 T + \text{count(“B” in PAST+FUTURE)}} * \frac{\beta_1 + \text{count(“(B→B)” in PAST+FUTURE)} + 1}{\beta_1 T + \text{count(“B” in PAST+FUTURE)} + 1}$$

Exercise. Explain the "+1" in the last numerator. Explain the "+1" in the last denominator.

Exercise. We just determined the incremental cost of adding the two bigrams "(B→B) (B→B)" to the end of the PAST+FUTURE cache in step 4 above. What's the incremental cost of removing the two bigrams "(B→A) (A→B)" in step 3?

Exercise. Put together an entire incremental scoring formula, including both tag bigram and tag/word events. Make it general. Assume we have a word sequence $w_1 \ldots w_n$ and a current sample $t_1 \ldots t_n$, and assume we consider changing the tag of the ith word from NN to VB. (Hard.)

26. The whole algorithm

Here's an overall plan for unsupervised tagging with the cache model. For your enjoyment, I've indicated various kinds of new **black magic in bold**.

1. Obtain an initial tagging sample for a string of length n. **This could be a random tag sequence, or it could be the tag sequence generated by plain EM (forward-backward).** Compute and store P(current-derivation) and the current event counts.

2. C = a storage bin with whole counts from the initial sample.

3. burn-in = 20. **We're going to make many passes over the data. "Burn in" means: don't collect counts in the first few passes. The theory is that the sampler will focus a good chunk of time initially just to escape the bad initial tagging. In steady state, it might not actually visit near there very much.**

4. collection-rate = 0.01. **Only collect counts from every hundredth sample. People do this to avoid collecting over samples that are too similar to each other.**

5. total-iterations = 200. **Just like EM, we need to know how many passes to make over the data. With EM, we can also use likelihood change as a stopping criterion, but that's tougher to do here.**

```
6. for i = 1 to total-iterations
      for j = 1 to n   /* total number of word tokens */
         for each tag T in tag set
            incrementally compute P(T-derivation), where we change jth word's tag to T
         choose new sample by 42-way weighted coin flip, with probability proportional to P(T-derivation)
         update P(current-derivation) and current event counts
         if (i ≥ burn-in) and (random(0,1) ≤ collection-rate) then
            incrementally update C with whole counts off the new sample
```

7. Normalize parameter counts in C, if desired.

Another black magic sampling idea is **annealing**. When choosing the next sample by weighted coin flip, we can cube-root all the P(derivation) values. That will make for a flatter distribution. For example, instead of 0.008 versus 0.027 (a roughly 25/75 coin flip), we'd have 0.2 versus 0.3 (a 40/60 coin flip). The sampler will effectively wander around even more drunkenly and maybe hit regions it would accidentally overlook otherwise. As iterations proceed, we "sober up" and start square-rooting instead of cube-rooting, and finally we stop rooting altogether.

27. Fiddling

Want to do better? **Fiddle with your α's and β's!** Of course, this is unsupervised learning, so it's very naughty to look at task accuracy while you fiddle with your α's and β's. If you were naughty, you might get accuracy results like these for cache-model tagging with a bigram tag model:

β_1 β_2 →	.001	.003	.01	.03	.1	.3	1.0	3.0	10.0
.001	84.71%	84.62%	84.54%	84.73%	84.54%	84.25%	83.89%	83.40%	84.64%
.003	84.90%	84.67%	84.75%	84.64%	84.51%	84.09%	83.80%	83.38%	84.67%
.01	84.70%	84.69%	84.36%	84.46%	84.46%	84.07%	83.84%	83.34%	84.68%
.03	84.99%	84.93%	84.71%	84.69%	84.79%	84.42%	84.14%	83.25%	84.30%
.1	84.68%	84.67%	84.85%	84.79%	84.99%	84.93%	84.55%	83.55%	82.72%
.3	84.19%	84.20%	84.28%	84.13%	**85.88%**	85.72%	85.09%	84.73%	83.08%
1.0	83.59%	83.70%	83.89%	83.72%	83.71%	83.52%	82.68%	84.41%	84.02%
3.0	83.27%	82.67%	82.05%	82.36%	82.59%	82.70%	82.45%	82.28%	84.76%
10.0	81.60%	81.73%	81.96%	82.39%	82.37%	82.43%	82.13%	83.12%	83.92%

The question of how to set α's and β's always comes up in this sort of work. It seems reasonable and practical to set them in order to optimize some end-to-end task accuracy. Bayesian people look at you sadly when you suggest this. They will tell you to embrace your uncertainty about what values α and β should take on, and reason under that uncertainty. For example, the distribution of α's values can be captured by a Gaussian distribution. Instead of choosing a specific value for α, we need only specify a mean (μ) and standard deviation (σ) for its distribution, and sample from that. Haven't we just replaced α with μ and σ? Yes, it's **turtles all the way down**.

28. What are we learning?

Step 7 says "if desired". That brings up the whole question of what we are learning. Of course, you and I want to learn things like P(NN | VB) = 0.32, so that we can tag new sentences with the Viterbi algorithm. But be careful saying that out loud. Bayesian people will get sad, or possibly go berserk. They will say, we want to learn a *distribution over possible values that P(NN | VB) might take*. Yeah! Then when we tag new sentences, we do it keeping that whole distribution in mind. Yeah!

Someone else might say, look, we just want to tag our unsupervised training set and get some accuracy result, like 85%, so we can tell our friends. Forget about tagging new sentences, and forget about distributions over values. Well, we could use our simple P(NN | VB) = 0.32 for that too. Or – and you'll find this in a lot of natural language work – we could forget about steps 2 and 7, and just use the **final sample** as our tag sequence. We could compute the tagging accuracy on that final sample. We could even collect counts off the final sample only. That would obviate the need for the counting trick from Section 22.

Another question. How do we track the progress of the learning? With plain EM, we report P(data) after each iteration, and we watch it improve. If P(data) suddenly gets worse after an iteration, we know we have a bug. With Gibbs sampling for the cache model, however, it's not easy to get P(data). We don't have access to all derivations, nor can we approximate easily with a subset of derivations. Programming bugs in Gibbs samplers may therefore be tougher to root out.

29. Other unsupervised problems

Now let's go back and think about how to do Chinese segmentation efficiently. We start with some initial segmentation of the Chinese characters into words. What kind of small change can we define, in order to get our Gibbs sampler to work? Consider the sliver of whitespace between each character. In the current sample, this sliver of whitespace is either *inside* of a (multi-character) word token, or else it is *between* two word tokens. The small change we contemplate is whether to flip that bit. In effect, a small change has the power to either split up an existing (multi-character) word token, or else join two separate word tokens. We repeatedly scan through the character sequence, splitting and joining, to sample the whole-corpus derivations.

Learning a Tree Substitution Grammar is similar. Every node in a tree is either the root of a TSG rule instance, or it isn't. We can flip such bits to sample TSG derivations of a corpus. Rules get split and joined by this method.

30. Small change operators

The more complex the problem, the more we have to design clever change operators. To knock a Gibbs sampler over into the good zone, we may need to provide it with the ability to make larger changes. A larger change could mean "change any pair of adjacent tags" (for tagging) or "move a word boundary to the right" (for segmenting). ~~Changing multiple variables at once is called **collapsed sampling**.~~ (*not!*)

Or we could re-sample at the whole-sentence level. This is sometimes called **block sampling**. Just like the Viterbi algorithm finds the best derivation sequence for a sentence, there are algorithms to randomly pull derivation sequences with probability proportional to P(derivation). With this plan, we may have to gloss over the fact that early decisions in the sentence-specific derivation sequence change the cache (a little bit) for later decisions in the same sentence-specific derivation.

> (I mentioned the Carmel finite-state toolkit in Section 2. A new version of Carmel contains switches that implement cache models and block-sampling as described in this workbook. As

with regular Carmel, the user simply sets up the problem as a cascade of arbitrary finite-state transducers and supplies the input/output strings for training).

For very hard problems, our sample-generator may be only approximately correct. Instead of generating samples proportional to their P(derivation) scores, it generates them proportional to some more-easily-computable Q(derivation) scores. For example, a block sampler might ignore the cache effects within the block itself. With **rejection sampling**, we re-score a proposed new sample by the true P(derivation), and we compare it to the true P(derivation) of our current sample. Then we may decide to reject the proposed sample – that is, don't move to it or collect counts from it. **Metropolis-Hastings** is a kind of rejection sampling. **Importance sampling** is like rejection sampling, but softer. It tries to steer the sampler into the truly good part of the space.

31. Different cache models & exchangeability

Remember Thomas, the guy we met in an earlier exercise, who ate Italian and Japanese food for dinner? Well, Margo is a friend of his. Margo is planning to eat out every night this week. She'll eat Italian on Monday. The other nights, if she's feeling playful, she'll flip an unbiased coin -- heads for Italian, and tails for Japanese. If she's not feeling playful, she'll consult the list of food choices she made so far that week, and go by that. If she's eaten mostly Italian so far, she's liable to eat Italian again.

Unlike Thomas, however, Margo **gets less playful** as the week proceeds.

Let H be the number of days that have passed (the history). The chance that Margo feels playful is α/α+H. Let's set α to 1. So on Tuesday, there's a 1/2 chance that Margo feels playful and will consult her coin. On Wednesday, there's a 1/3 chance she'll feel playful. By Sunday, there's only a 1/6 chance she'll feel playful. At that point, she's much more likely to use the week's history as a guide for what to eat next. Margo fits our cache model:

$$\mathbf{P(Italian) = \alpha/(\alpha+H) * P_0(Italian) + H/(\alpha+H) * count(Italian\ so\ far\ this\ week)/H}$$

Boxed-formula version: $[\alpha\ P_0(\text{Italian}) + \text{count(Italian so far)}] / [\alpha + H]$

Here, P_0(Italian) = 1/2 (that's Margo's unbiased coin). Let's investigate three possible meal sequences and their associated probabilities. The first sequence is all Italian, every day. (Stop the gnocchi!). The second sequence is the same, except for one Japanese meal, on Thursday. The third sequence also has one Japanese meal, but it's on Sunday.

Day	Sequence 1		Sequence 2		Sequence 3	
	meal	P(meal)	meal	P(meal)	meal	P(meal)
Monday	Italian	1	Italian	1	Italian	1
Tuesday	Italian	3/4	Italian	3/4	Italian	3/4
Wednesday	Italian	5/6	Italian	5/6	Italian	5/6
Thursday	Italian	7/8	**Japanese**	1/8	Italian	7/8
Friday	Italian	9/10	Italian	7/10	Italian	9/10
Saturday	Italian	11/12	Italian	9/12	Italian	11/12
Sunday	Italian	13/14	Italian	11/14	**Japanese**	1/14
P(sequence)		0.420		0.032		0.032

Sequence 2 and Sequence 3 have the same probability. It doesn't matter when Margo has her Japanese meal. It's exchangeable. That also means we can quickly and incrementally score Sequence 2, given the probability of Sequence 1. We take Sequence 1, replace Sunday's Italian meal with Japanese, giving us Sequence 3, then we move that Japanese meal to Thursday, giving us Sequence 2. The cost of that Sunday meal replacement has two factors:

Remove Sunday Italian meal – divide P(Sequence 1) by 1/7 * 1/2 + 6/7 * 6/6 = 13/14
Add Sunday Japanese meal – multiply the result by 1/7 * 1/2 + 6/7 * 0 = 1/14

So the incremental cost is a factor of 13, which is corroborated by the last row of the table: Sequence 2 is thirteen times less likely than Sequence 1, overall.

Now recall that back in Section 11, we first proposed a simpler, more intuitive cache model:

P(Italian) = β * P_0(Italian) + (1-β) * count(Italian so far this week)/H

This model just interpolates the base probability with the cache. It does not bias the model to pull more from the cache as H gets bigger.

Margo has a friend named Jenni. Unlike Margo, Jenni **does not get less playful** as the week progresses. On any given day, there's a fixed 50% chance that she feels playful. So for Jenni, we can use the P(Italian) formula above, with β = 0.5. Here are the sequence probabilities:

Day	Sequence 1		Sequence 2		Sequence 3	
	meal	P(meal)	meal	P(meal)	meal	P(meal)
Monday	Italian	1	Italian	1	Italian	1
Tuesday	Italian	3/4	Italian	3/4	Italian	3/4
Wednesday	Italian	3/4	Italian	3/4	Italian	3/4
Thursday	Italian	3/4	**Japanese**	1/4	Italian	3/4
Friday	Italian	3/4	Italian	5/8	Italian	3/4
Saturday	Italian	3/4	Italian	13/20	Italian	3/4
Sunday	Italian	3/4	Italian	2/3	**Japanese**	1/4
P(sequence)		0.178		0.038		0.059

Sequence 2 and Sequence 3 now have different probabilities. If Jenni is going to eat one Japanese meal this week, it's more likely to be on Sunday than on Thursday! Not exchangeable. That means, if we had stuck with our simpler, more intuitive cache model, we'd have been in trouble when we arrived to the incremental-scoring guts of our Gibbs sampling plan.

So is **$\alpha/(\alpha+H)$** better than **β**? I'm not sure. We ought to judge generative models on how well they empirically explain natural language data -- accuracy, perplexity, all that. But, hmm … maybe the math trick associated with **$\alpha/(\alpha+H)$** is a sign from the gods. Could this be their not-so-subtle way of steering us towards a subtly superior **$\alpha/(\alpha+H)$** cache model? Hmm. I actually think that Brown et al (speaking of gods) were more on the right track. If you have read Brown et al 93, or the "Statistical MT Tutorial Workbook", then you witnessed the math trick associated with IBM alignment Model 1. (It moved a product over a sum, resulting in a quadratic-time EM algorithm). But Brown et al always had linguistically better models in mind -- Models 3, 4, 5, and beyond -- and they only exploited Model 1 to get them into position to learn those nicer models, for which nice math tricks don't exist.

32. Long tail

One consequence of **α/(α+H)** is that the cache eventually takes over -- after we have generated enough data, we (pretty much) no longer consult the base distribution. We reach a point where "there's nothing new under the sun".

You may insist that natural language doesn't work that way. No matter how much text you look at, you keep finding new things! There's a super-long tail. You may insist that **β** captures this phenomenon better than **α/(α+H)** does, since it keeps on using the base distribution. After all, the base distribution often can have its own micro-generative-story that spews out an infinite number of new item types (words/rules/whatever), thus strrrrrrrretching out that long tail.

Okay, you've made a good point.

How about this, then. Instead of consulting the base distribution with probability **α/(α+H)**, we can consult it with probability

$$\frac{\boldsymbol{\alpha + d * (\text{\# of times base distribution has been consulted so far in building the cache})}}{\boldsymbol{\alpha + H}}$$

d is a "discount" factor between 0 and 1. How to set d? You've got to be kidding me.

Here, each time we consult the base distribution, we actually raise the chance that we will consult the base distribution again in the future. That way, we can keep a long tail going. This story is called the **Pitman-Yor process**.

This generative story is looking pretty cool. It is really intent on making a long tail. Imagine trying to stop it. We can certainly set α=0. That will reduce the chance of consulting the base distribution, but not kill it. Suppose further that we get "lucky" and manage to avoid the base distribution while generating data items 2 through 999,999. Even so, remember that we had to use the base distribution to generate item number 1, so there will still be some tiny probability of consulting it for item number 1,000,000. Can you figure out what that probability is? (Hint: it's a function of d and 1,000,000). Long live the tail.

Of course, if we *really* don't want a long tail (as with Penn Treebank non-terminals), then we can simply set d=0, in which case our formula reverts to **α/(α+H)**. So this new business is a true generalization of the **α/(α+H)** story.

To flesh this generative story out:

Let **u** be the number of times the base distribution has been consulted in building the cache.
Let **c(item)** be the count of an item (word/rule/whatever) in the cache.
Generative process:

1. Generate the first data item from the base distribution P_0.

2. To generate each additional data item:
   With probability (α+du)/(α+H), generate an item from the base distribution P_0.
   With probability (1 - (α+du)/(α+H)), pick an item from the cache, according to (c(item)-d)/(H-du).

Sorry that we can't do the usual c(item)/H at the end there, when we select from the cache. You can see some discounting going on, to balance out how much the P_0 is getting allocated.

The formula is:

$$P(\text{item}) = (\alpha+du)/(\alpha+H) * P_0(\text{item}) + (1 - (\alpha+du)/(\alpha+H)) \begin{cases} (c(\text{item})-d) / (H-du) & : \text{ if item is in cache} \\ \text{zero} & : \text{ otherwise} \end{cases}$$

Exercise. Derive this simplified "boxed formula" for the new story:

$$\mathbf{P(item)} = \frac{\alpha + du}{\alpha + H} * P_0(\text{item}) + \left\{ \frac{c(\text{item}) - d}{\alpha + H} \text{ if item is in cache; zero otherwise} \right\}$$

You might see this formula in papers that use the Pitman-Yor process. Good to know where it comes from!

Exercise. Show that P(item) sums to one, considering all items that one might generate.

Exercise. Do we get exchangeability with the new story?

33. Subtract 1/2

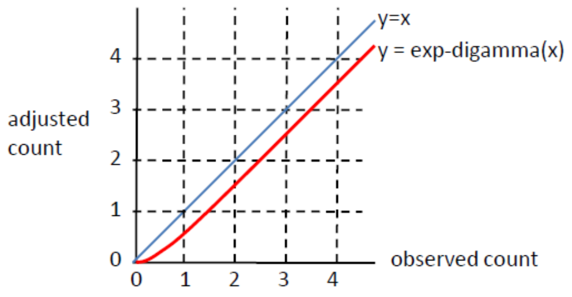
Now for a little light reading to round things out. Remember that the whole idea of the cache model is to encourage re-use. The rich get richer, and the poor get zeroed out, which leads to sparse models.

Are there other ways to implement the rich-get-richer idea? Sure. We can just steal from the poor during EM … Sheriff-of-Nottingham style! Let's go back to plain EM, without the cache model, and let's say we're on the first iteration. We observe A eleven times, and we observe B once. We then set P(B) = 1/12. But, hmm, that pesky B is still hanging around, clogging up our parameter table. Even if EM eventually zeroes B out, it might take many iterations. Here's an idea: let's **subtract 1/2** from everyone's counts right now. So we pretend to observe A 10.5 times and we pretend to observe B 0.5 times, so P(B) = 0.5/11 = 1/22. Any derivation with B in it will now receive a much lower score on the next EM iteration, and B will then likely receive an even lower count. B is headed for oblivion. Sparse models!

This idea seems like the opposite of smoothing, doesn't it? In smoothing, we steal counts from the rich and give them to the poor. The poorest are the ones with zero counts. Actually, good smoothing methods are pretty funny. In re-assigning counts to the poor, they don't actually steal from the rich, they steal from the lower-middle class. The one- and two-count items get massive cuts (perhaps 1/3 of their count taken away), while the 100,000-count items get a relatively tiny cut.

Anyway, the Subtract-1/2 strategy is easy to implement. Just use plain EM, but at each iteration, subtract 1/2 from every parameter count. Do this just after count collection is finished, right before normalization.

A necessary twist: EM does fractional count collection, so a parameter might wind up with a total count of 1.55, or 0.72, or even 0.16. But it doesn't make sense to subtract 1/2 from 0.16, does it? Apparently, stealing from the poor has its limits! Instead, we use the **exp-digamma function** (whoa!!). Here's a picture adapted from Mark Johnson:



If your count is high, exp-digamma subtracts 1/2, but if your count is low, it subtracts less. It never returns a negative. This graph looks like a flat tax proposal, doesn't it? Need to build a new highway? Everybody pitch in 50 dollars. Some people don't have 50 dollars? Okay, they can pitch in 10 dollars.

Exercise. Go find the formula for exp-digamma.

The Subtract-1/2 idea falls under a topic called **Variational Bayes.** There's a long proof, and at the end of the proof, it says: "Subtract 1/2".

Does Subtract-1/2 lead to better task-accuracy than EM? I've seen accuracy improve by a percentage point, and I've also seen accuracy degrade by a percentage point. Here are some results for unsupervised tagging with a tag bigram model:

EM:	80.8%
EM with "Subtract 1/2" strategy:	81.3%
EM with 100 random restarts:	82.3%
EM with "Subtract 1/2" strategy, with 100 restarts:	82.0%

So results are mixed. Subtract-1/2 certainly converges faster, with less dilly-dallying. The poor die off quickly. If you've ever run the kind of EM where you only collect counts over the Viterbi derivation (sometimes called "Viterbi training" or "Hard EM"), then you already know this effect. By the way, random restarts are good! If you use EM, please do random restarts.

34. The end

Nice work. Time for beer!

35. About the author

Kevin Knight will serve as President of the Association for Computational Linguistics in 2011. His primary responsibilities will be to grant wishes and pardons. If you've been bad, or are thinking about being bad, you can go ahead and put in for a pardon. If you have a wish, it's best just to come right out and say it.