# Penalized Expectation Propagation for Graphical Models over Strings*

**Ryan Cotterell** and **Jason Eisner**
Department of Computer Science, Johns Hopkins University
{ryan.cotterell,jason}@cs.jhu.edu

## Abstract

We present penalized expectation propagation (PEP), a novel algorithm for approximate inference in graphical models. Expectation propagation is a variant of loopy belief propagation that keeps messages tractable by projecting them back into a given family of functions. Our extension, PEP, uses a structured-sparsity penalty to encourage simple messages, thus balancing speed and accuracy. We specifically show how to instantiate PEP in the case of string-valued random variables, where we adaptively approximate finite-state distributions by variable-order $n$-gram models. On phonological inference problems, we obtain substantial speedup over previous related algorithms with no significant loss in accuracy.

## 1 Introduction

Graphical models are well-suited to reasoning about linguistic structure in the presence of uncertainty. Such models typically use discrete random variables, where each variable ranges over a finite set of values such as words or tags. But a variable can also be allowed to range over an *infinite* space of discrete *structures*—in particular, the set of all strings, a case first explored by Bouchard-Côté et al. (2007).

This setting arises because human languages make use of many word forms. These strings are systematically related in their spellings due to linguistic processes such as morphology, phonology, abbreviation, copying error and historical change. To analyze or predict novel strings, we can model the joint distribution of many related strings at once. Under a graphical model, the joint probability of an assignment tuple is modeled as a product of potentials on sub-tuples, each of which is usually modeled in turn by a weighted finite-state machine.

In general, we wish to infer the values of unknown strings in the graphical model. Deterministic approaches to this problem have focused on belief propagation (BP), a message-passing algorithm that is exact on acyclic graphical models and approximate on cyclic ("loopy") ones (Murphy et al., 1999). But in both cases, further heuristic approximations of the BP messages are generally used for speed.

In this paper, we develop a more principled and flexible way to approximate the messages, using variable-order $n$-gram models.

We first develop a version of expectation propagation (EP) for string-valued variables. EP offers a principled way to approximate BP messages by distributions from a *fixed* family—e.g., by trigram models. Each message update is found by minimizing a certain KL-divergence (Minka, 2001a).

Second, we generalize to variable-order models. To do this, we augment EP's minimization problem with a novel penalty term that keeps the number of $n$-grams finite. In general, we advocate penalizing more "complex" messages (in our setting, large finite-state acceptors). Complex messages are slower to construct, and slower to use in later steps.

Our penalty term is formally similar to regularizers that encourage structured sparsity (Bach et al., 2011; Martins et al., 2011). Like a regularizer, it lets us use a more expressive family of distributions, secure in the knowledge that we will use only as many of the parameters as we really need for a "pretty good" fit. But *why* avoid using more parameters? Regularization seeks better *generalization* by not overfitting the model to the data. By contrast, we already have a model and are merely doing inference. We seek better *runtime* by not over-fussing about capturing the model's marginal distributions.

Our "penalized EP" (PEP) inference strategy is applicable to any graphical model with complex messages. In this paper, we focus on strings, and show how PEP speeds up inference on the computational phonology model of Cotterell et al. (2015).

We provide further details, tutorial material, and results in the appendices (supplementary material).

*This material is based upon work supported by the National Science Foundation under Grant No. 1423276, and by a Fulbright Research Scholarship to the first author.

## 2 Background

Graphical models over strings are in fairly broad use. Linear-chain graphical models are equivalent to cascades of finite-state transducers, which have long been used to model stepwise derivational processes such as speech production (Pereira and Riley, 1997) and transliteration (Knight and Graehl, 1998). Tree-shaped graphical models have been used to model the evolution and speciation of word forms, in order to reconstruct ancient languages (Bouchard-Côté et al., 2007; Bouchard-Côté et al., 2008) and discover cognates in related languages (Hall and Klein, 2010; Hall and Klein, 2011). Cyclic graphical models have been used to model morphological paradigms (Dreyer and Eisner, 2009; Dreyer and Eisner, 2011) and to reconstruct phonological underlying forms (Cotterell et al., 2015). All of these graphical models, except Dreyer's, happen to be *directed* ones. And all of these papers, except Bouchard-Côté's, use *deterministic* inference methods—based on BP.

### 2.1 Graphical models over strings

A directed or undirected graphical model describes a joint probability distribution over a set of random variables. To perform inference *given* a setting of the model parameters *and* observations of some variables, it is convenient to construct a *factor graph* (Kschischang et al., 2001). A factor graph is a finite bipartite graph whose vertices are the random variables $\{V_1, V_2, \ldots\}$ and the factors $\{F_1, F_2, \ldots\}$. Each factor $F$ is a function of the variables that it is connected to; it returns a non-negative real number that depends on the values of those variables. We define our factor graph so that the posterior probability $p(V_1 = v_1, V_2 = v_2, \ldots \mid \text{observations})$, as defined by the original graphical model, can be computed as proportional to the product of the numbers returned by all the factors when $V_1 = v_1, V_2 = v_2, \ldots$.

In a graphical model *over strings*, each random variable $V$ is permitted to range over the strings $\Sigma^*$ where $\Sigma$ is a fixed alphabet. As in previous work, we will assume that each factor $F$ connected to $d$ variables is a $d$-way rational relation, i.e., a function that can be computed by a $d$-tape weighted finite-state acceptor (Elgot and Mezei, 1965; Mohri et al., 2002; Kempe et al., 2004). The weights fall in the semiring $(\mathbb{R}, +, \times)$: $F$'s return value is the *total* weight of all paths that accept the $d$-tuple of strings, where a path's weight is the *product* of its arcs' weights. So our model marginalizes over possible paths in $F$.

### 2.2 Inference by (loopy) belief propagation

Inference seeks the posterior marginal probabilities $p(V_i = v \mid \text{observations})$, for each $i$. BP is an iterative procedure whose "normalized beliefs" converge to exactly these marginals if the factor graph is acyclic (Pearl, 1988). In the cyclic case, the normalized beliefs still typically converge and can be used as approximate marginals (Murphy et al., 1999).

A full presentation of BP for graphical models over strings can be found in Dreyer and Eisner (2009). We largely follow their notation. $\mathcal{N}(X)$ represents the set of neighbors of $X$ in the factor graph.

For each edge in the factor graph, between a factor $F$ and a variable $V$, BP maintains two *messages*, $\mu_{V \to F}$ and $\mu_{F \to V}$. Each of these is a function over the possible values $v$ of variable $V$, mapping each $v$ to a non-negative score. BP also maintains another such function, the *belief* $b_V$, for each variable $V$.

In general, each message or belief should be regarded as giving only *relative* scores for the different $v$. Rescaling it by a positive constant would only result in rescaling other messages and beliefs, which would not change the final normalized beliefs. The *normalized belief* is the probability distribution $\hat{b}_V$ such that each $\hat{b}_V(v)$ is proportional to $b_V(v)$.

The basic BP algorithm is just to repeatedly select and update a function until convergence. The rules for updating $\mu_{V \to F}$, $\mu_{F \to V}$, and $b_V$, given the set of "neighboring" messages in each case, can be found as equations (2)–(4) of Dreyer and Eisner (2009). (We will give the EP variants in section 4.)

Importantly, that paper shows that for graphical models over strings, each BP update can be implemented via standard finite-state operations of composition, projection, and intersection. Each message or belief is represented as a weighted finite-state acceptor (WFSA) that scores all strings $v \in \Sigma^*$.

### 2.3 The need for approximation

BP is generally only used directly for short cascades of finite-state transducers (Pereira and Riley, 1997; Knight and Graehl, 1998). Alas, in other graphical models over strings, the BP messages—which are acceptors—become too large to be practical.

In cyclic factor graphs, where exact inference for strings can be *undecidable*, the WFSAs can become *unboundedly* large as they are iteratively updated around a cycle (Dreyer and Eisner, 2009). Even in an acyclic graph (where BP is exact), the finite-state operations quickly lead to large WFSAs. Each intersection or composition is a Cartesian product construction, whose output's size (number of automaton states) may be as large as the *product* of its inputs' sizes. Combining many of these operations leads to exponential blowup.

## 3 Variational Approximation of WFSAs

To address this difficulty through EP (section 4), we will need the ability to approximate any probability distribution $p$ that is given by a WFSA, by choosing a "simple" distribution from a family $\mathcal{Q}$.

Take $\mathcal{Q}$ to be a family of log-linear distributions

$$q_{\boldsymbol{\theta}}(v) \stackrel{\text{def}}{=} \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(v)) \,/\, Z_{\boldsymbol{\theta}} \quad (\forall v \in \Sigma^*) \qquad (1)$$

where $\boldsymbol{\theta}$ is a weight vector, $\boldsymbol{f}(v)$ is a feature vector that describes $v$, and $Z_{\boldsymbol{\theta}} \stackrel{\text{def}}{=} \sum_{v \in \Sigma^*} \exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(v))$ so that $\sum_v q_{\boldsymbol{\theta}}(v) = 1$. Notice that the featurization function $\boldsymbol{f}$ specifies the family $\mathcal{Q}$, while the variational parameters $\boldsymbol{\theta}$ specify a particular $q \in \mathcal{Q}$.[1]

We project $p$ into $\mathcal{Q}$ via inclusive KL divergence:

$$\boldsymbol{\theta} = \operatorname{argmin}_{\boldsymbol{\theta}} \mathrm{D}(p \mid\mid q_{\boldsymbol{\theta}}) \qquad (2)$$

Now $q_{\boldsymbol{\theta}}$ approximates $p$, and has support everywhere that $p$ does. We can get finer-grained approximations by expanding $\boldsymbol{f}$ to extract more features: however, $\boldsymbol{\theta}$ is then larger to store and slower to find.

### 3.1 Finding $\boldsymbol{\theta}$

Solving (2) reduces to maximizing $-H(p, q_{\boldsymbol{\theta}}) = \mathbb{E}_{v \sim p}[\log q_{\boldsymbol{\theta}}(v)]$, the log-likelihood of $q_{\boldsymbol{\theta}}$ on an "infinite sample" from $p$. This is similar to fitting a log-linear model to data (without any regularization: we want $q_{\boldsymbol{\theta}}$ to fit $p$ as well as possible). This objective is concave and can be maximized by following its gradient $\mathbb{E}_{v \sim p}[\boldsymbol{f}(v)] - \mathbb{E}_{v \sim q_{\boldsymbol{\theta}}}[\boldsymbol{f}(v)]$. Often it is also possible to optimize $\boldsymbol{\theta}$ in closed form, as we will see later. Either way, the optimal $q_{\boldsymbol{\theta}}$ matches $p$'s *expected* feature vector: $\mathbb{E}_{v \sim q_{\boldsymbol{\theta}}}[\boldsymbol{f}(v)] = \mathbb{E}_{v \sim p}[\boldsymbol{f}(v)]$. This inspired the name "expectation propagation."

### 3.2 Working with $\boldsymbol{\theta}$

Although $p$ is defined by an arbitrary WFSA, we can represent $q_{\boldsymbol{\theta}}$ quite simply by just storing the parameter vector $\boldsymbol{\theta}$. We will later take sums of such vectors to construct product distributions: observe that under (1), $q_{\boldsymbol{\theta}_1 + \boldsymbol{\theta}_2}(v)$ is proportional to $q_{\boldsymbol{\theta}_1}(v) \cdot q_{\boldsymbol{\theta}_2}(v)$.

We will also need to construct WFSA versions of these distributions $q_{\boldsymbol{\theta}} \in \mathcal{Q}$, and of other log-linear functions (messages) that may not be normalizable into distributions. Let $\textsc{Encode}(\boldsymbol{\theta})$ denote a WFSA that accepts each $v \in \Sigma^*$ with weight $\exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(v))$.

### 3.3 Substring features

To obtain our family $\mathcal{Q}$, we must design $\boldsymbol{f}$. Our strategy is to choose a set of "interesting" substrings $\mathcal{W}$. For each $w \in \mathcal{W}$, define a feature function "How many times does $w$ appear as a substring of $v$?" Thus, $\boldsymbol{f}(v)$ is simply a vector of counts (nonnegative integers), indexed by the substrings in $\mathcal{W}$.

A natural choice of $\mathcal{W}$ is the set of all $n$-grams for fixed $n$. In this case, $\mathcal{Q}$ turns out to be equivalent to the family of $n$-gram language models.[2] Already in previous work ("variational decoding"), we used (2) with this family to approximate WFSAs or weighted hypergraphs that arose at runtime (Li et al., 2009).

Yet a fixed $n$ is not ideal. If $\mathcal{W}$ is the set of bigrams, one might do well to add the trigram `the`—perhaps because `the` is "really" a bigram (counting the digraph `th` as a single consonant), or because the bigram model fails to capture how common `the` is under $p$. Adding `the` to $\mathcal{W}$ ensures that $q_\theta$ will now match $p$'s expected count for this trigram. Doing this should not require adding all $|\Sigma|^3$ trigrams.

By including strings of mixed lengths in $\mathcal{W}$ we get *variable-order* Markov models (Ron et al., 1996).

### 3.4 Arbitrary FSA-based features

More generally, let $\mathcal{A}$ be any *unambiguous and complete* finite-state acceptor: that is, any $v \in \Sigma^*$ has exactly one accepting path in $\mathcal{A}$. For each arc or final state $a$ in $\mathcal{A}$, we can define a feature function "How

[1] To be precise, we take $\mathcal{Q} = \{q_{\boldsymbol{\theta}} : Z_{\boldsymbol{\theta}} \text{ is finite}\}$. For example, $\boldsymbol{\theta} = \mathbf{0}$ is excluded because then $Z_{\boldsymbol{\theta}} = \sum_{v \in \Sigma^*} \exp 0 = \infty$. Aside from this restriction, $\boldsymbol{\theta}$ may be any vector over $\mathbb{R} \cup \{-\infty\}$. We allow $-\infty$ since it is a feature's optimal weight if $p(v) = 0$ for all $v$ with that feature: then $q_{\boldsymbol{\theta}}(v) = 0$ for such strings as well. (Provided that $\boldsymbol{f}(v) \geq \mathbf{0}$, as we will ensure.)

[2] Provided that we include special $n$-grams that match at the boundaries of $v$. See Appendix B.2 for details.

many times is $a$ used when $\mathcal{A}$ accepts $v$?" Thus, $\boldsymbol{f}(v)$ is again a vector of non-negative counts.

Section 6 gives algorithms for this general setting. We implement the previous section as a special case, constructing $\mathcal{A}$ so that its arcs essentially correspond to the substrings in $\mathcal{W}$. This encodes a variable-order Markov model as an FSA similarly to (Allauzen et al., 2003); see Appendix B.4 for details.

In this general setting, ENCODE($\boldsymbol{\theta}$) just returns a *weighted* version of $\mathcal{A}$ where each arc or final state $a$ has weight $\exp \theta_a$ in the $(+, \times)$ semiring. Thus, this WFSA accepts each $v$ with weight $\exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(v))$.

### 3.5 Adaptive featurization

How do we choose $\mathcal{W}$ (or $\mathcal{A}$)? Expanding $\mathcal{W}$ will allow better approximations to $p$—but at greater computational cost. We would like $\mathcal{W}$ to include just the substrings needed to approximate a *given* $p$ well. For instance, if $p$ is concentrated on a few high-probability strings, then a good $\mathcal{W}$ might contain those full strings (with positive weights), plus some shorter substrings that help model the rest of $p$.

To select $\mathcal{W}$ at runtime in a way that adapts to $p$, let us say that $\boldsymbol{\theta}$ is actually an infinite vector with weights for *all possible* substrings, and define $\mathcal{W} = \{w \in \Sigma^* : \theta_w \neq 0\}$. Provided that $\mathcal{W}$ stays finite, we can store $\boldsymbol{\theta}$ as a map from substrings to *nonzero* weights. We keep $\mathcal{W}$ small by replacing (2) with

$$\boldsymbol{\theta} = \operatorname{argmin}_{\boldsymbol{\theta}} \mathrm{D}(p \,||\, q_{\boldsymbol{\theta}}) + \lambda \cdot \Omega(\boldsymbol{\theta}) \qquad (3)$$

where $\Omega(\boldsymbol{\theta})$ measures the complexity of this $\mathcal{W}$ or the corresponding $\mathcal{A}$. Small WFSAs ensure fast finite-state operations, so ideally, $\Omega(\boldsymbol{\theta})$ should measure the size of ENCODE($\boldsymbol{\theta}$). Choosing $\lambda > 0$ to be large will then emphasize speed over accuracy.

Section 6.1 will extend section 6's algorithms to approximately minimize the new objective (3). Formally this objective resembles regularized log-likelihood. However, $\Omega(\boldsymbol{\theta})$ is not a regularizer—as section 1 noted, we have no statistical reason to avoid "overfitting" $\hat{p}$, only a computational one.

## 4 Expectation Propagation

Recall from section 2.2 that for each variable $V$, the BP algorithm maintains several nonnegative functions that score $V$'s possible values $v$: the messages $\mu_{V \to F}$ and $\mu_{F \to V}$ ($\forall F \in \mathcal{N}(V)$), and the belief $b_V$.

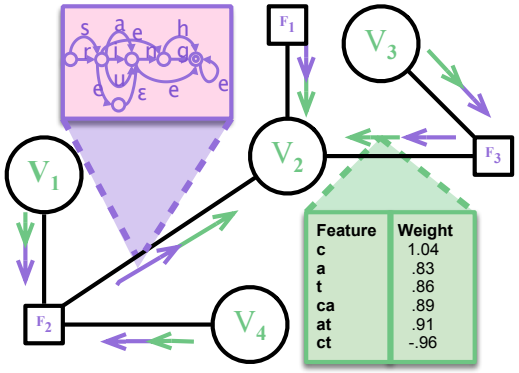


Figure 1: Information flowing toward $V_2$ in EP (reverse flow not shown). The factors work with purple $\mu$ messages represented by WFSAs, while the variables work with green $\boldsymbol{\theta}$ messages represented by log-linear weight vectors. The green table shows a $\boldsymbol{\theta}$ message: a sparse weight vector that puts high weight on the string `cat`.

EP is a variant in which all of these are forced to be log-linear functions from the same family, namely $\exp(\boldsymbol{\theta} \cdot \boldsymbol{f}_V(v))$. Here $\boldsymbol{f}_V$ is the featurization function we've chosen for variable $V$.[3] We can represent these functions by their parameter vectors—let us call those $\boldsymbol{\theta}_{V \to F}$, $\boldsymbol{\theta}_{F \to V}$, and $\boldsymbol{\theta}_V$ respectively.

### 4.1 Passing messages through variables

What happens to BP's update equations in this setting? According to BP, the belief $b_V$ is the pointwise product of all "incoming" messages to $V$. But as we saw in section 3.2, pointwise products are *far easier* in EP's restricted setting! Instead of intersecting several WFSAs, we can simply add several vectors:

$$\boldsymbol{\theta}_V = \sum_{F' \in \mathcal{N}(V)} \boldsymbol{\theta}_{F' \to V} \qquad (4)$$

Similarly, the "outgoing" message from $V$ to factor $F$ is the pointwise product of all "incoming" messages *except* the one from $F$. This message $\boldsymbol{\theta}_{V \to F}$ can be computed as $\boldsymbol{\theta}_V - \boldsymbol{\theta}_{F \to V}$, which adjusts (4).[4] We never store this but just compute it on demand.

[3] A single graphical model might mix categorical variables, continuous variables, orthographic strings over (say) the Roman alphabet, and phonological strings over the International Phonetic Alphabet. These different data types certainly require different featurization functions. Moreover, even when two variables have the same type, we could choose to approximate their marginals differently, e.g., with bigram vs. trigram features.

[4] If features can have $-\infty$ weight (footnote 1), this trick might need to subtract $-\infty$ from $-\infty$ (the log-space version of $0/0$). That gives an undefined result, but it turns out that any result will do—it makes no difference to the subsequent beliefs.

### 4.2 Passing messages through factors

Our factors are weighted finite-state machines, so their messages still require finite-state computations, as shown by the purple material in Figure 1. These computations are just as in BP. Concretely, let $F$ be a factor of degree $d$, given as a $d$-tape machine. We can compute a belief *at this factor* by joining $F$ with $d$ WFSAs that represent its $d$ incoming messages, namely $\text{ENCODE}(\boldsymbol{\theta}_{V' \to F})$ for $V' \in \mathcal{N}(F)$. This gives a new $d$-tape machine, $b_F$. We then obtain each outgoing message $\mu_{F \to V}$ by projecting $b_F$ onto its $V$ tape, but removing (dividing out) the weights that were contributed (multiplied in) by $\boldsymbol{\theta}_{V \to F}$.[5]

### 4.3 Getting from factors back to variables

Finally, we reach the only tricky step. Each resulting $\mu_{F \to V}$ is a possibly large WFSA, so we must *force it back into our log-linear family* to get an updated approximation $\boldsymbol{\theta}_{F \to V}$. One cannot directly employ the methods of section 3, because KL divergence is only defined between probability distributions. ($\mu_{F \to V}$ might not be normalizable into a distribution, nor is its best approximation necessarily normalizable.)

The EP trick is to use section 3 to instead approximate the belief at $V$, which *is* a distribution, and *then* reconstruct the approximate message to $V$ that would have produced this approximated belief. The "unapproximated belief" $\hat{p}_V$ resembles (4): it multiplies the unapproximated message $\mu_{F \to V}$ by the current values of all *other* messages $\boldsymbol{\theta}_{F' \to V}$. We know the product of those *other* messages, $\boldsymbol{\theta}_{V \to F}$, so

$$\hat{p}_V := \mu_{F \to V} \odot \mu_{V \to F} \tag{5}$$

where the pointwise product $\odot$ is carried out by WFSA intersection and $\mu_{V \to F} \stackrel{\text{def}}{=} \text{ENCODE}(\boldsymbol{\theta}_{V \to F})$.

We now apply section 3 to choose $\boldsymbol{\theta}_V$ such that $q_{\boldsymbol{\theta}_V}$ is a good approximation of the WFSA $\hat{p}_V$. Finally, to preserve (4) as an invariant, we reconstruct

$$\boldsymbol{\theta}_{F \to V} := \boldsymbol{\theta}_V - \boldsymbol{\theta}_{V \to F} \tag{6}$$

[5]This is equivalent to computing each $\mu_{F \to V}$ by "generalized composition" of $F$ with the $d-1$ messages to $F$ from its *other* neighbors $V'$. The operations of join and generalized composition were defined by Kempe et al. (2004).

In the simple case $d = 2$, $F$ is just a weighted finite-state transducer mapping $V'$ to $V$, and computing $\mu_{F \to V}$ reduces to composing $\text{ENCODE}(\boldsymbol{\theta}_{V' \to F})$ with $F$ and projecting the result onto the output tape. In fact, one can assume WLOG that $d \leq 2$, enabling the use of popular finite-state toolkits that handle at most 2-tape machines. See Appendix B.10 for the construction.

In short, EP combines $\mu_{F \to V}$ with $\boldsymbol{\theta}_{V \to F}$, then approximates the result $\hat{p}_V$ by $\boldsymbol{\theta}_V$ before removing $\boldsymbol{\theta}_{V \to F}$ again. Thus EP is approximating $\mu_{F \to V}$ by

$$\boldsymbol{\theta}_{F \to V} := \operatorname*{argmin}_{\boldsymbol{\theta}} \mathrm{D}(\underbrace{\mu_{F \to V} \odot \mu_{V \to F}}_{= \hat{p}_V} \,||\, \underbrace{q_{\boldsymbol{\theta}} \odot \mu_{V \to F}}_{= \boldsymbol{\theta}_V}) \tag{7}$$

in a way that updates not only $\boldsymbol{\theta}_{F \to V}$ but also $\boldsymbol{\theta}_V$.

Wisely, this objective focuses on approximating the message's scores for the *plausible* values $v$. Some values $v$ may have $\hat{p}_V(v) \approx 0$, perhaps because *another* incoming message $\boldsymbol{\theta}_{F' \to V}$ rules them out. It does not much harm the objective (7) if these $\mu_{F \to V}(v)$ are poorly approximated by $q_{\boldsymbol{\theta}_{F \to V}}(v)$, since the overall belief is still roughly correct.

Our *penalized* EP simply adds $\lambda \cdot \Omega(\boldsymbol{\theta})$ into (7).

### 4.4 The EP algorithm: Putting it all together

To run EP (or PEP), initialize all $\boldsymbol{\theta}_V$ and $\boldsymbol{\theta}_{F \to V}$ to $\mathbf{0}$, and then loop repeatedly over the nodes of the factor graph. When visiting a factor $F$, ENCODE its incoming messages $\boldsymbol{\theta}_{V \to F}$ (computed on demand) as WFSAs, construct a belief $b_F$, and update the outgoing WFSA messages $\mu_{F \to V}$. When visiting a variable $V$, iterate $K \geq 1$ times over its incoming WFSA messages: for each incoming $\mu_{F \to V}$, compute the unapproximated belief $\hat{p}_V$ via (5), then update $\boldsymbol{\theta}_V$ to approximate $\hat{p}_V$, then update $\boldsymbol{\theta}_{F \to V}$ via (6).

For possibly faster convergence, one can alternate "forward" and "backward" sweeps. Visit the factor graph's nodes in a fixed order (given by an approximate topological sort). At a factor, update the outgoing WFSA messages to *later* variables only. At a variable, approximate only those incoming WFSA messages from *earlier* factors (all the outgoing messages $\boldsymbol{\theta}_{V \to F}$ will be recomputed on demand). Note that both cases examine all incoming messages. After each sweep, reverse the node ordering and repeat.

If gradient ascent is used to find the $\boldsymbol{\theta}_V$ that approximates $\hat{p}_V$, it is wasteful to optimize to convergence. After all, the optimization problem will keep changing as the messages change. Our implementation improves $\boldsymbol{\theta}_V$ by only a single gradient step on each visit to $V$, since $V$ will be visited repeatedly.

See Appendix A for an alternative view of EP.

## 5 Related Approximation Methods

We have presented EP as a method for simplifying a variable's incoming messages during BP. The vari-

able's outgoing messages are pointwise products of the incoming ones, so they become simple too. Past work has used approximations with a similar flavor.

Hall and Klein (2011) heuristically predetermine a short, fixed list of plausible values for $V$ that were observed elsewhere in their dataset. This list is analogous to our $\boldsymbol{\theta}_V$. After updating $\mu_{F\to V}$, they force $\mu_{F\to V}(v)$ to 0 for all $v$ outside the list, yielding a *finite* message that is analogous to our $\boldsymbol{\theta}_{F\to V}$.

Our own past papers are similar, except they *adaptively* set the "plausible values" list to $\bigcup_{F'\in\mathcal{N}(V)} k\text{-BEST}(\mu_{F'\to V})$. These strings are favored by at least one of the *current* messages to $V$ (Dreyer and Eisner, 2009; Dreyer and Eisner, 2011; Cotterell et al., 2015). Thus, simplifying one of $V$'s incoming messages considers all of them, as in EP.

The above methods *prune* each message, so may prune correct values. Hall and Klein (2010) avoid this: they fit a full bigram model by inclusive KL divergence, which refuses to prune *any* values (see section 3). Specifically, they minimized $\mathrm{D}(\mu_{F\to V}\odot\tau \mid\mid q_{\boldsymbol{\theta}}\odot\tau)$, where $\tau$ was a simple fixed function (a 0-gram model) included so that they were working with distributions (see section 4.3). This is very similar to our (7). Indeed, Hall and Klein (2010) found their procedure "reminiscent of EP," hinting that $\tau$ was a surrogate for a real $\mu_{V\to F}$ term. Dreyer and Eisner (2009) had also suggested EP as future work.

EP has been applied only twice before in the NLP community. Daumé III and Marcu (2006) used EP for query summarization (following Minka and Lafferty (2003)'s application to an LDA model with fixed topics) and Hall and Klein (2012) used EP for rich parsing. However, these papers inferred a single structured variable connected to all factors (as in the traditional presentation of EP—see Appendix A), rather than inferring many structured variables connected in a sparse graphical model.

We regard EP as a generalization of loopy BP for just this setting: graphical models with large or unbounded variable domains. Of course, we are not the first to use such a scheme; e.g., Qi (2005, chapter 2) applies EP to linear-chain models with both continuous and discrete hidden states. We believe that EP should also be broadly useful in NLP, since it naturally handles joint distributions over the kinds of structured variables that arise in NLP.

## 6 Two Methods for Optimizing $\boldsymbol{\theta}$

We now fill in details. If the feature set is defined by an unambiguous FSA $\mathcal{A}$ (section 3.4), two methods exist to max $\mathbb{E}_{v\sim p}[\log q_{\boldsymbol{\theta}}(v)]$ as section 3.1 requires.

**Closed-form.** Determine how often $\mathcal{A}$ would traverse each of its arcs, in expectation, when reading a random string drawn from $p$. We would obtain an optimal $\text{ENCODE}(\boldsymbol{\theta})$ by, at each state of $\mathcal{A}$, setting the weights of the arcs from that state to be proportional to these counts while summing to 1.[6] Thus, the logs of these arc weights give an optimal $\boldsymbol{\theta}$.

For example, in a trigram model, the probability of the `c` arc from the `ab` state is the expected count of `abc` (according to $p$) divided by the expected count of `ab`. Such expected substring counts can be found by the method of Allauzen et al. (2003). For general $\mathcal{A}$, we can use the method sketched by Li et al. (2009, footnote 9): intersect the WFSA for $p$ with the unweighted FSA $\mathcal{A}$, and then run the forward-backward algorithm to determine the posterior count of each arc in the result. This tells us the expected total number of traversals of each arc in $\mathcal{A}$, if we have kept track of which arcs in the intersection of $p$ with $\mathcal{A}$ were derived from which arcs in $\mathcal{A}$. That bookkeeping can be handled with an expectation semiring (Eisner, 2002), or simply with backpointers.

**Gradient ascent.** For any given $\boldsymbol{\theta}$, we can use the WFSAs $p$ and $\text{ENCODE}(\boldsymbol{\theta})$ to exactly compute $\mathbb{E}_{v\sim p}[\log q_{\boldsymbol{\theta}}(v)] = -H(p, q_{\boldsymbol{\theta}})$ (Cortes et al., 2006). We can tune $\boldsymbol{\theta}$ to globally maximize this objective.

The technique is to intersect $p$ with $\text{ENCODE}(\boldsymbol{\theta})$, after lifting their weights into the expectation semiring via the mappings $k \mapsto \langle k, 0\rangle$ and $k \mapsto \langle 0, \log k\rangle$ respectively. Summing over all paths of this intersection via the forward algorithm yields $\langle Z, r\rangle$ where $Z$ is the normalizing constant for $p$. We also sum over paths of $\text{ENCODE}(\boldsymbol{\theta})$ to get the normalizing constant $Z_{\boldsymbol{\theta}}$. Now the desired objective is $r/Z - \log Z_{\boldsymbol{\theta}}$. Its *gradient* with respect to $\boldsymbol{\theta}$ can be found by back-propagation, or equivalently by the forward-backward algorithm (Li and Eisner, 2009).

An overlarge gradient step can leave the feasible space (footnote 1) by driving $Z_{\boldsymbol{\theta}_V}$ to $\infty$ and thus driving (2) to $\infty$ (Dreyer, 2011, section 2.8.2). In this case, we try again with reduced stepsize.

[6] This method always yields a probabilistic FSA, i.e., the arc weights are locally normalized probabilities. This does not sacrifice any expressiveness; see Appendix B.7 for discussion.

### 6.1 Optimizing $\boldsymbol{\theta}$ with a penalty

Now consider the penalized objective (3). Ideally, $\Omega(\boldsymbol{\theta})$ would count the number of nonzero weights in $\boldsymbol{\theta}$—or better, the number of arcs in $\textsc{Encode}(\boldsymbol{\theta})$. But it is not known how to efficiently minimize the resulting discontinuous function. We give two approximate methods, based on the two methods above.

**Proximal gradient.** Leaning on recent advances in sparse estimation, we replace this $\Omega(\boldsymbol{\theta})$ with a convex surrogate whose partial derivative with respect to each $\theta_w$ is undefined at $\theta_w = 0$ (Bach et al., 2011). Such a penalty tends to create sparse optima.

A popular surrogate is an $\ell_1$ penalty, $\Omega(\boldsymbol{\theta}) \stackrel{\text{def}}{=} \sum_w |\boldsymbol{\theta}_w|$. However, $\ell_1$ would not recognize that $\boldsymbol{\theta}$ is simpler with the features $\{\texttt{ab}, \texttt{abc}, \texttt{abd}\}$ than with the features $\{\texttt{ab}, \texttt{pqr}, \texttt{xyz}\}$. The former leads to a smaller WFSA encoding. In other words, it is cheaper to add `abd` once `abc` is already present, as a state already exists that represents the context `ab`.

We would thus like the penalty to be the number of distinct *prefixes* in the set of nonzero features,

$$|\{u \in \Sigma^* : (\exists x \in \Sigma^*)\, \theta_{ux} \neq 0\}|, \tag{8}$$

as this is the number of ordinary arcs in $\textsc{Encode}(\boldsymbol{\theta})$ (see Appendix B.4). Its convex surrogate is

$$\Omega(\boldsymbol{\theta}) \stackrel{\text{def}}{=} \sum_{u \in \Sigma^*} \sqrt{\sum_{x \in \Sigma^*} \theta_{ux}^2} \tag{9}$$

This *tree-structured group lasso* (Nelakanti et al., 2013) is an instance of group lasso (Yuan and Lin, 2006) where the string $w = \texttt{abd}$ belongs to four groups, corresponding to its prefixes $u = \epsilon, u = \texttt{a}, u = \texttt{ab}, u = \texttt{abd}$. Under group lasso, moving $\theta_w$ away from 0 increases $\Omega(\boldsymbol{\theta})$ by $\lambda|\theta_w|$ (just as in $\ell_1$) *for each group in which $w$ is the only nonzero feature*. This penalizes for the new WFSA arcs needed for these groups. There are also increases due to $w$'s other groups, but these are smaller, especially for groups with many strongly weighted features.

Our objective (3) is now the sum of a differentiable convex function (2) and a *particular* non-differentiable convex function (9). We minimize it by proximal gradient (Parikh and Boyd, 2013). At each step, this algorithm first takes a gradient step as in section 6 to improve the differentiable term, and then applies a "proximal operator" to jump to a nearby point that improves the non-differentiable term. The proximal operator for tree-structured group lasso (9) can be implemented with an efficient recursive procedure (Jenatton et al., 2011).

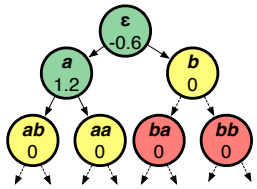


Figure 2: Active set method, showing the infinite tree of all features for the alphabet $\Sigma = \{a, b\}$. The green nodes currently have non-zero weights. The yellow nodes are on the frontier and are *allowed* to become non-zero, but the penalty function is still keeping them at 0. The red nodes are not yet considered, forcing them to remain at 0.

What if $\boldsymbol{\theta}$ is $\infty$-dimensional because we allow all $n$-grams as features? Paul and Eisner (2012) used just this feature set in a dual decomposition algorithm. Like them, we rely on an *active set* method (Schmidt and Murphy, 2010). We fix `abcd`'s weight at 0 until `abc`'s weight becomes nonzero (if ever);[7] only then does feature `abc` become "active." Thus, at a given step, we only have to compute the gradient with respect to the currently nonzero features (green nodes in Figure 2) and their immediate children (yellow nodes). This hierarchical inclusion technique ensures that we only consider a small, finite subset of all $n$-grams at any given iteration of optimization.

**Closed-form with greedy growing.** There are existing methods for estimating variable-order $n$-gram language models from data, based on either "shrinking" a high-order model (Stolcke, 1998) or "growing" a low-order one (Siivola et al., 2007).

We have designed a simple "growing" algorithm to estimate such a model from a WFSA $p$. It approximately minimizes the objective (3) where $\Omega(\boldsymbol{\theta})$ is given by (8). We enumerate all $n$-grams $w \in \Sigma^*$ in decreasing order of expected count (this can be done efficiently using a priority queue). We add $w$ to $\mathcal{W}$ if we *estimate* that it will decrease the objective. Every so often, we measure the *actual* objective (just as in the gradient-based methods), and we stop if it is no longer improving. Algorithmic details are given in Appendices B.8–B.9.

[7] Paul and Eisner (2012) also required `bcd` to have nonzero weight, observing that `abcd` is a *conjunction* `abc`$\wedge$`bcd` (McCallum, 2003). This added test would be wise for us too.

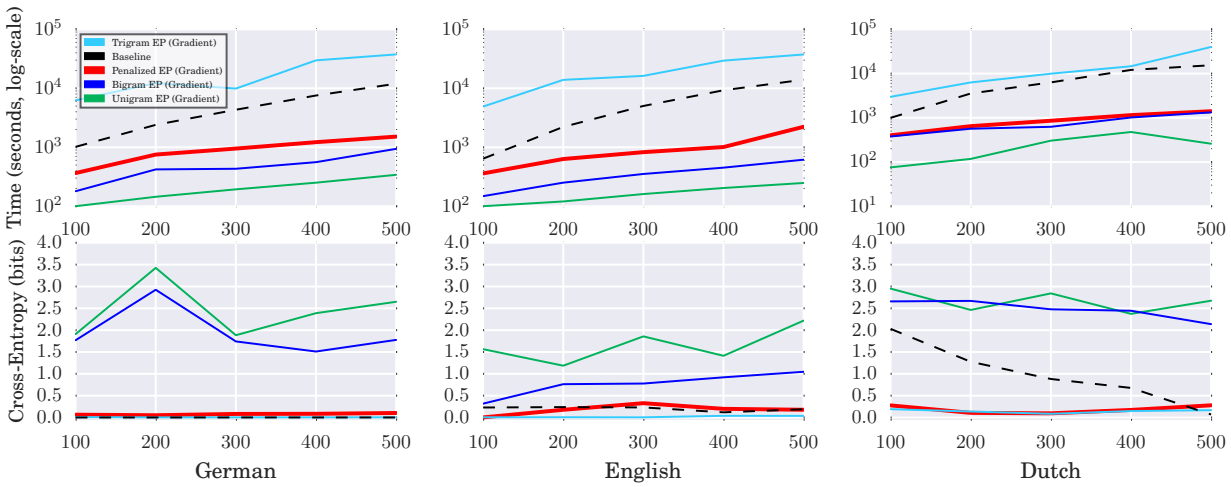


Figure 3: Inference on 15 factor graphs (3 languages $\times$ 5 datasets of different sizes). The first row shows the total runtime (logscale) of each inference method. The second row shows the accuracy, as measured by the negated log-probability that the inferred belief at a variable assigns to its gold-standard value, averaged over "underlying morpheme" variables. At this penalty level ($\lambda = 0.01$), PEP [thick line] is faster than the pruning baseline of Cotterell et al. (2015) [dashed line] and much faster than trigram EP, yet is about as accurate. (For Dutch with sparse observations, it is considerably more accurate than baseline.) Indeed, PEP is nearly as fast as bigram EP, which has terrible accuracy. An ideal implementation of PEP would be faster yet (see Appendix B.5). Further graphs are in Appendix C.

## 7 Experiments and Results

Our experimental design aims to answer three questions. (1) Is our algorithm able to beat a strong baseline (adaptive pruning) in a non-trivial model? (2) Is PEP actually better than ordinary EP, given that the structured sparsity penalty makes it more algorithmically complex? (3) Does the $\lambda$ parameter successfully trade off between speed and accuracy?

All experiments took place using the graphical model over strings for the discovery of underlying phonological forms introduced in Cotterell et al. (2015). They write: "Comparing `cats` ([kæts]), `dogs` ([dɔgz]), and `quizzes` ([kwɪzɪz]), we see the English plural morpheme evidently has at least three pronunciations." Cotterell et al. (2015) sought a unifying account of such variation in terms of phonological underlying forms for the morphemes.

In their Bayes net, morpheme underlying forms are latent variables, while word surface forms are observed variables. The factors model underlying-to-surface phonological changes. They learn the factors by Expectation Maximization (EM). Their first E step presents the hardest inference problem because the factors initially contribute no knowledge of the language; so that is the setting we test on here.

Their data are surface phonological forms from the CELEX database (Baayen et al., 1995). For each of 3 languages, we run 5 experiments, by observing the surface forms of 100 to 500 words and running EP to infer the underlying forms of their morphemes. Each of the 15 factor graphs has $\approx$ 150–700 latent variables, joined by 500–2200 edges to 200–1200 factors of degree 1–3. Variables representing suffixes can have extremely high degree ($> 100$).

We compare PEP with other approximate inference methods. As our main baseline, we take the approximation scheme actually used by Cotterell et al. (2015), which restricts the domain of a belief to that of the union of 20-best strings of its incoming messages (section 5).We also compare to unpenalized EP with unigram, bigram, and trigram features.

We report both speed and accuracy for all methods. Speed is reported in seconds. Judging accuracy is a bit trickier. The best metric would to be to measure our beliefs' distance from the true marginals or even from the beliefs computed by vanilla loopy BP. Obtaining these quantities, however, would be extremely expensive—even Gibbs sampling is infeasible in our setting, let alone 100-way WFSA intersections. Luckily, Cotterell et al. (2015) provide gold-standard values for the latent variables (underlying

forms). Figure 3 shows the negated log-probabilities of these gold strings according to our beliefs, averaged over variables in a given factor graph. Our accuracy is weaker than Cotterell et al. (2015) because we are doing inference with their *initial* (untrained) parameters, a more challenging problem.

Each update to $\boldsymbol{\theta}_V$ consisted of a single step of (proximal) gradient descent: starting at the current value, improve (2) with a gradient step of size $\eta = 0.05$, then (in the adaptive case) apply the proximal operator of (9) with $\lambda = 0.01$. We chose these values by preliminary exploration, taking $\eta$ small enough to avoid backtracking (section 6.1).

We repeatedly visit variables and factors (section 4.4) in the forward-backward order used by Cotterell et al. (2015). For the first few iterations, when we visit a variable we make $K = 20$ passes over its incoming messages, updating them iteratively to ensure that the high probability strings in the initial approximations are "in the ballpark". For subsequent iterations of message passing we take $K = 1$. For similar reasons, we constrained PEP to use only unigram features on the first iteration, when there are still many viable candidates for each morph.

### 7.1 Results

The results show that PEP is much faster than the baseline pruning method, as described in Figure 3 and its caption. It mainly achieves better cross-entropy on English and Dutch, and even though it loses on German, it still places almost all of its probability mass on the gold forms. While EP with unigram and bigram approximations are both faster than PEP, their accuracy is poor. Trigram EP is nearly as accurate but even slower than the baseline. The results support the claim that PEP has achieved a "Goldilocks number" of $n$-grams in its approximation—just enough $n$-grams to approximate the message well while retaining speed.

Figure 4 shows the effect of $\lambda$ on the speed-accuracy tradeoff. To compare apples to apples, this experiment fixed the set of $\mu_{F \to V}$ messages for each variable. Thus, we held the set of beliefs fixed, but measured the size and accuracy of different approximations to these beliefs by varying $\lambda$.

These figures show only the results from gradient-based approximation. Closed-form approximation is faster and comparably accurate: see Appendix C.

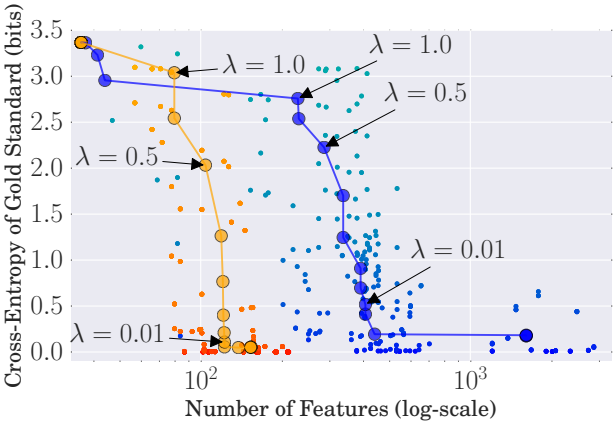


Figure 4: Increasing $\lambda$ will greatly reduce the number of selected features in a belief—initially without harming accuracy, and then accuracy degrades gracefully. (Number of features has 0.72 correlation with runtime, and is shown on a *log scale* on the $x$ axis.)

Each point shows the result of using PEP to approximate the belief at some latent variable $V$, using $\mu_{F \to V}$ messages from running the baseline method on German. Lighter points use larger $\lambda$. Orange points are affixes (shorter strings), blue are stems (longer strings). Large circles are averages over all points for a given $\lambda$.

## 8 Conclusion and Future Work

We have presented penalized expectation propagation (PEP), a novel approximate inference algorithm for graphical models, and developed specific techniques for string-valued random variables. Our method integrates structured sparsity directly into inference. Our experiments show large speedups over the strong baseline of Cotterell et al. (2015).

In future, instead of choosing $\lambda$, we plan to reduce $\lambda$ as PEP runs. This serves to gradually refine the approximations, yielding an *anytime algorithm* whose beliefs approach the BP beliefs. Thanks to (7), the coarse messages from early iterations guide the choice of finer-grained messages at later iterations. In this regard, "Anytime PEP" resembles other coarse-to-fine architectures such as generalized A* search (Felzenszwalb and McAllester, 2007).

As NLP turns its attention to lower-resource languages and social media, it is important to model the rich phonological, morphological, and orthographic processes that interrelate words. We hope that the introduction of faster inference algorithms will increase the use of graphical models over strings. We are releasing our code package (see Appendix D).

## References


Cyril Allauzen, Mehryar Mohri, and Brian Roark. 2003. Generalized algorithms for constructing statistical language models. In *Proceedings of ACL*, pages 40–47.

Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. OpenFst: A general and efficient weighted finite-state transducer library. In *Proceedings of the 12th International Conference on Implementation and Application of Automata*, volume 4783 of *Lecture Notes in Computer Science*, pages 11–23. Springer.

R. Harald Baayen, Richard Piepenbrock, and Leon Gulikers. 1995. The CELEX lexical database on CD-ROM.

Francis Bach, Rodolphe Jenatton, Julien Mairal, Guillaume Obozinski, et al. 2011. Convex optimization with sparsity-inducing norms. In S. Sra, S. Nowozin, and S. J. Wright, editors, *Optimization for Machine Learning*. MIT Press.

Alexandre Bouchard-Côté, Percy Liang, Thomas L Griffiths, and Dan Klein. 2007. A probabilistic approach to diachronic phonology. In *Proceedings of EMNLP-CoNLL*, pages 887–896.

Alexandre Bouchard-Côté, Percy Liang, Thomas Griffiths, and Dan Klein. 2008. A probabilistic approach to language change. In *Proceedings of NIPS*.

Victor Chahuneau. 2013. PyFST. `https://github.com/vchahun/pyfst`.

Corinna Cortes, Mehryar Mohri, Ashish Rastogi, and Michael D Riley. 2006. Efficient computation of the relative entropy of probabilistic automata. In *LATIN 2006: Theoretical Informatics*, pages 323–336. Springer.

Ryan Cotterell, Nanyun Peng, and Jason Eisner. 2014. Stochastic contextual edit distance and probabilistic FSTs. In *Proceedings of ACL*, pages 625–630.

Ryan Cotterell, Nanyun Peng, and Jason Eisner. 2015. Modeling word forms using latent underlying morphs and phonology. *Transactions of the Association for Computational Linguistics*. To appear.

Hal Daumé III and Daniel Marcu. 2006. Bayesian query-focused summarization. In *Proceedings of ACL*, pages 305–312.

Markus Dreyer and Jason Eisner. 2009. Graphical models over multiple strings. In *Proceedings of EMNLP*, pages 101–110, Singapore, August.

Markus Dreyer and Jason Eisner. 2011. Discovering morphological paradigms from plain text using a Dirichlet process mixture model. In *Proceedings of EMNLP*, pages 616–627, Edinburgh, July.

Markus Dreyer. 2011. *A Non-Parametric Model for the Discovery of Inflectional Paradigms from Plain Text Using Graphical Models over Strings*. Ph.D. thesis, Johns Hopkins University, Baltimore, MD, April.

Jason Eisner. 2002. Parameter estimation for probabilistic finite-state transducers. In *Proceedings of ACL*, pages 1–8.

C.C. Elgot and J.E. Mezei. 1965. On relations defined by generalized finite automata. *IBM Journal of Research and Development*, 9(1):47–68.

Gal Elidan, Ian Mcgraw, and Daphne Koller. 2006. Residual belief propagation: Informed scheduling for asynchronous message passing. In *Proceedings of UAI*.

P. F. Felzenszwalb and D. McAllester. 2007. The generalized A* architecture. *Journal of Artificial Intelligence Research*, 29:153–190.

David Hall and Dan Klein. 2010. Finding cognate groups using phylogenies. In *Proceedings of ACL*.

David Hall and Dan Klein. 2011. Large-scale cognate recovery. In *Proceedings of EMNLP*.

David Hall and Dan Klein. 2012. Training factored PCFGs with expectation propagation. In *Proceedings of EMNLP*.

Tom Heskes and Onno Zoeter. 2002. Expectation propagation for approximate inference in dynamic Bayesian networks. In A. Darwiche and N. Friedman, editors, *Uncertainty in Artificial Intelligence: Proceedings of the Eighteenth Conference (UAI-2002)*, pages 216–233, San Francisco, CA. Morgan Kaufmann Publishers. A more detailed technical report can be found at `http://www.cs.ru.nl/~tomh/techreports/heskes03extended.pdf`.

Rodolphe Jenatton, Julien Mairal, Guillaume Obozinski, and Francis Bach. 2011. Proximal methods for hierarchical sparse coding. *The Journal of Machine Learning Research*, 12:2297–2334.

André Kempe, Jean-Marc Champarnaud, and Jason Eisner. 2004. A note on join and auto-intersection of $n$-ary rational relations. In Loek Cleophas and Bruce Watson, editors, *Proceedings of the Eindhoven FASTAR Days (Computer Science Technical Report 04-40)*, pages 64–78. Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, Netherlands, December.

Kevin Knight and Jonathan Graehl. 1998. Machine transliteration. *Computational Linguistics*, 24(4).

F. R. Kschischang, B. J. Frey, and H. A. Loeliger. 2001. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, February.

Zhifei Li and Jason Eisner. 2009. First- and second-order expectation semirings with applications to minimum-risk training on translation forests. In *Proceedings of EMNLP*, pages 40–51.

Zhifei Li, Jason Eisner, and Sanjeev Khudanpur. 2009. Variational decoding for statistical machine translation. In *Proceedings of ACL*, pages 593–601.

André F. T. Martins, Noah A. Smith, Pedro M. Q. Aguiar, and Mário A. T. Figueiredo. 2011. Structured sparsity in structured prediction. In *Proceedings of EMNLP*, pages 1500–1511.

Andrew McCallum. 2003. Efficiently inducing features of conditional random fields. In *Proceedings of UAI*.

Thomas Minka and John Lafferty. 2003. Expectation-propagation for the generative aspect model. In *Proceedings of UAI*.

Thomas P Minka. 2001a. Expectation propagation for approximate Bayesian inference. In *Proceedings of UAI*, pages 362–369.

Thomas P. Minka. 2001b. *A Family of Algorithms for Approximate Bayesian Inference*. Ph.D. thesis, Massachusetts Institute of Technology, January.

Thomas Minka. 2005. Divergence measures and message passing. Technical Report MSR-TR-2005-173, Microsoft Research, January.

Mehryar Mohri, Fernando Pereira, and Michael Riley. 2002. Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69–88.

Mehryar Mohri. 2000. Minimization algorithms for sequential transducers. *Theoretical Computer Science*, 324:177–201, March.

Mehryar Mohri. 2005. Local grammar algorithms. In Antti Arppe, Lauri Carlson, Krister Lindèn, Jussi Piitulainen, Mickael Suominen, Martti Vainio, Hanna Westerlund, and Anssi Yli-Jyrä, editors, *Inquiries into Words, Constraints, and Contexts: Festschrift in Honour of Kimmo Koskenniemi on his 60th Birthday*, chapter 9, pages 84–93. CSLI Publications, Stanford University.

Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. 1999. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of UAI*, pages 467–475.

Anil Nelakanti, Cédric Archambeau, Julien Mairal, Francis Bach, Guillaume Bouchard, et al. 2013. Structured penalties for log-linear language models. In *Proceedings of EMNLP*, pages 233–243.

Neal Parikh and Stephen Boyd. 2013. Proximal algorithms. *Foundations and Trends in Optimization*, 1(3):123–231.

Michael Paul and Jason Eisner. 2012. Implicitly intersecting weighted automata using dual decomposition. In *Proceedings of NAACL-HLT*, pages 232–242, Montreal, June.

Judea Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, California.

Fernando C. N. Pereira and Michael Riley. 1997. Speech recognition by composition of weighted finite automata. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*. MIT Press, Cambridge, MA.

Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. 2006. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of COLING-ACL*, pages 433–440, July.

Yuan Qi. 2005. *Extending Expectation Propagation for Graphical Models*. Ph.D. thesis, Massachusetts Institute of Technology, February.

Dana Ron, Yoram Singer, and Naftali Tishby. 1996. The power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning*, 25(2-3):117–149.

Mark W Schmidt and Kevin P Murphy. 2010. Convex structure learning in log-linear models: Beyond pairwise potentials. In *Proceedings of AISTATS*, pages 709–716.

Vesa Siivola, Teemu Hirsimäki, and Sami Virpioja. 2007. On growing and pruning Kneser-Ney smoothed $n$-gram models. *IEEE Transactions on Audio, Speech, and Language Processsing*, 15(5):1617–1624.

Andreas Stolcke. 1998. Entropy-based pruning of backoff language models. In *Proceedings of the DARPA Broadcast News Transcription and Understanding Workshop*, pages 270–274.

Ming Yuan and Yi Lin. 2006. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67.

# Appendices

## A Expectation Propagation

In this appendix, we give a different perspective on EP and its relationship to BP. We begin with Minka's original presentation of EP, where $p$ is a product distribution but not necessarily a factor graph.

### A.1 What is EP?

What is EP in general? Suppose we hope to approximate some complex distribution $p(x)$ by fitting a log-linear model (i.e., an exponential-family model),

$$q_{\boldsymbol{\theta}}(x) \stackrel{\text{def}}{=} (1/Z)\exp(\boldsymbol{\theta} \cdot \boldsymbol{f}(x))$$

where $\boldsymbol{\theta}$ is a parameter vector and $\boldsymbol{f}(x)$ is a feature vector. It is well-known that the KL divergence $\mathrm{D}(p \mid\mid q_{\boldsymbol{\theta}})$ is convex and can be minimized by following its gradient, $\mathbb{E}_{x\sim q_{\boldsymbol{\theta}}}[\boldsymbol{f}(x)] - \mathbb{E}_{x\sim p}[\boldsymbol{f}(x)]$.

However, what if $p$ is defined as some product of factors? In this case, it may be intractable to compute $\mathbb{E}_{x\sim p}[\boldsymbol{f}(x)]$. EP is a specific iterative algorithm for *approximately* fitting $q_{\boldsymbol{\theta}}$ in this setting, by solving a succession of *simpler* min-KL problems. Each of these simpler problems does not use $p$ but rather a partially approximated version that combines a single factor of $p$ with the previous iteration's estimate of $q_{\boldsymbol{\theta}}$. The algorithm and its justification can be found in (Minka, 2001a; Minka, 2001b).

### A.2 From EP to exact BP

Now suppose that $X = (V_1, V_2, \ldots)$, with each factor in $p$ depending on only some of the variables $V_i$. In other words, $p$ is specified by a factor graph. Furthermore, suppose we define the log-linear model $q_{\boldsymbol{\theta}}$ to use all features of the form $V_i = v_i$: that is, one indicator feature for every possible variable-value pair, but no features that consider pairs of variables. In this case, it is not hard to see that $\mathbb{E}_{x\sim p}[\boldsymbol{f}(x)]$ encodes the *exact* marginals of $p$. If we could *exactly* minimize $\mathrm{D}(p \mid\mid q_{\boldsymbol{\theta}})$, then we would have $\mathbb{E}_{x\sim q_{\boldsymbol{\theta}}}[\boldsymbol{f}(x)] = \mathbb{E}_{x\sim p}[\boldsymbol{f}(x)]$, recovering these exact marginals. This is the problem that EP approximately solves, thus recovering *approximate* marginals of $p$—just like BP.

In fact, Minka (2001b) shows that **EP in this special case is equivalent to loopy BP**. Minka goes on to construct *more accurate* EP algorithms by *lengthening* the feature vector. However, recall from section 2.3 that BP is already too slow in our setting! So we instead derive a *faster approximation* by *shortening* the EP feature vector.

### A.3 From EP to approximate BP

Put differently, when there are infinitely many values (e.g., $\Sigma^*$), we cannot afford the BP strategy of a separate indicator feature for each variable-value pair. However, we can still use a finite set of backed-off features (e.g., $n$-gram features) that *inspect* the values. Recall that in section 4, we designed a featurization function $\boldsymbol{f}_V$ for each variable $V$. We can concatenate the results of these functions to get a *global* featurization function $\boldsymbol{f}$ that computes features of $x = (v_1, v_2, \ldots)$, just as in section A.2. Each feature still depends on just one variable-value pair.

In this backed-off case, EP reduces to the algorithm that we presented in section 4—essentially "BP with log-linear approximations"—which exploits the structure of the factor graph. We suppress the proof, as showing the equivalence would require first presenting EP in terms of (Minka, 2001b). However, it is a trivial extension of Minka's proof for the previous section. Minka presumably regarded the reduction as obvious, since his later presentation of EP in Minka (2005), where he relates it to other message-passing algorithms, also exploits the structure of the factor graph and hence is essentially the same as section 4. We have merely tried to present this algorithm in a more concrete and self-contained way.

*Heskes and Zoeter (2002) were the first to make this connection, recasting EP as a variant of BP that uses a "collapse-product rule." We found their paper just after the present paper appeared; we apologize for the omission.*

The approximate BP algorithm can alternatively be viewed as applying EP *within the BP algorithm*, using it separately *at each variable* $V$. Exact BP would need to compute the belief at $V$ as the product of the incoming messages $\mu_{F\to V}$. Since this is a product distribution, EP can be used to approximate it by a log-linear function, and this is exactly how our method finds $\boldsymbol{\theta}_V$ (section 4.3). Exact BP would also need to compute each outgoing message $\mu_{V\to F}$, as a product of all the incoming messages but one. We recover approximations to these as a side effect.[8]

[8] Rather than running EP separately to approximate each of these products, which would be more accurate but less efficient.

### A.4 From approximate BP to EP

Conversely, *any* EP algorithm can be viewed as an instance of "BP with log-linear approximations" as presented in section 4. Recall that EP always approximates a distribution $p(x)$ that is defined by a product of factors. That corresponds to a trivial factor graph where all factors are connected to a single variable $X$. The standard presentation of EP simply runs our algorithm on that trivial graph, maintaining a complex belief $q_X(x)$. This belief is a log-linear distribution in which any feature may look at the entire value $x$.

For readers who wish to work through the connection, the notation of Minka (2001b, p. 20)

$$t_i,\ \tilde{t}_i,\ \hat{p}_i,\ q^{\text{new}},\ q_i$$

(where Minka drops the subscripts on the temporary objects $\hat{p}$ and $q$ "for notational simplicity") corresponds respectively to our section 4's notation

$$\mu_{F_i \to X},\ q_{\boldsymbol{\theta}_{F_i \to X}},\ \hat{p}_X,\ q_{\boldsymbol{\theta}_X},\ q_{\boldsymbol{\theta}_{X \to F_i}}$$

All of these objects are functions over some value space. We use $v \in \Sigma^*$ to refer to the values. Minka uses $\theta \in \mathbb{R}$, which is unrelated to our $\boldsymbol{\theta}$.

### A.5 So is there any advantage?

In short, the algorithm of section 4 constitutes an alternative general presentation of EP, one that builds on understanding of BP and explicitly considers the factor graph structure.

The difference in our presentation is that we start with a finer-grained factor graph in which $X$ is decomposed into variables, $X = (V_1, V_2, \ldots)$, and each factor $F$ only looks at some of the variables. Our features are constrained to be local to single variables.

What would happen if we performed inference on the trivial graph *without* taking advantage of this finer-grained structure? Then the belief $q_X$, which is a log-linear distribution parameterized by $\boldsymbol{\theta}_X$, would take a factored form: $q_X(v_1, v_2, \ldots) = q_{V_1}(v_1) \cdot q_{V_2}(v_2) \cdot \cdots$. Here each $q_{V_i}$ is a log-linear distribution parameterized by a subvector $\boldsymbol{\theta}_{V_i}$ of $\boldsymbol{\theta}_X$. The messages between $X$ and a factor $F$ would formally be functions of the entire value of $X$, but would only actually depend on the dimensions $V_i \in \mathcal{N}(F)$. Their representations $\boldsymbol{\theta}_{X \to F}$ and $\boldsymbol{\theta}_{F \to X}$ would be 0 except for features of $V_i \in \mathcal{N}(F)$.

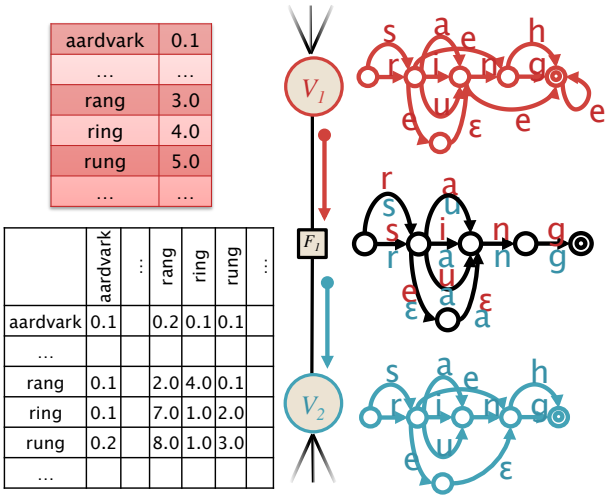


Figure 5: A sketch of one step of finite-state BP in a fragment of a factor graph. The red and blue machines are weighted finite-state acceptors that encode messages, while the black machine is a weighted finite-state transducer that encodes a factor. The blue message $\mu_{F_1 \to V_2}$ is computed by NEWFV from the red message $\mu_{V_1 \to F_1}$ and the black factor $F_1$. Specifically, it is the result of *composing* the red message with the factor and *projecting* the result onto the blue tape. The tables display some of the scores assigned by the red message and by the factor.

The resulting run of inference would be isomorphic to a run of our (approximate) BP algorithm. However, it would lose some space and time efficiency from failing to exploit the sparsity of these vectors. It would also fail to exploit the factored form of the beliefs. That factored form allows our algorithm (section 4.3) to visit a single variable and update just its belief parameters $\boldsymbol{\theta}_{V_i}$ (considering all factors $F \in \mathcal{N}(V_i)$). Under our algorithm, one can visit a variable $V_i$ more often if one wants to spend more time refining its belief. Our algorithm can also more carefully choose the order in which to visit variables (section 4.4), by considering the topology of the factor graph (Pearl, 1988), or considering the current message values (Elidan et al., 2006).

## B Variable-order $n$-gram models and WFSAs

In this appendix, we describe automaton constructions that are necessary for the main paper.

### B.1 Final arcs

We standardly use $h \xrightarrow{c/k} h'$ to denote an arc from state $h$ to state $h'$ that consumes symbol $c$ with weight $k$.

To simplify our exposition, we will use a non-standard WFSA format where all weights appear on arcs, never on final states. WFSAs can be trivially converted between our expository format and the traditional format used for implementation (Appendix D).

Wherever the traditional WFSA format would designate a state $h$ as a final state with stopping weight $k$, we instead use an arc $h \xrightarrow{\text{EOS}/k} \square$. This arc allows the WFSA to stop at state $h$ by reading the distinguished end-of-string symbol EOS and transitioning to the distinguished final state $\square$.

For a WFSA in our format, we say formally that an accepting path for string $v \in \Sigma^*$ is a path from the initial state to $\square$ that is labeled with $v\,\text{EOS}$. The weight of this path is the product of its arc weights, including the weight of the EOS arc. We use $\Sigma' \stackrel{\text{def}}{=} \Sigma \cup \{\text{EOS}\}$ to denote the set of possible arc labels.

### B.2 The substring set $\mathcal{W}$

Section 3.3 defines a class of log-linear functions based on a fixed finite set of strings $\mathcal{W}$.

We allow strings in $\mathcal{W}$ to include the symbols BOS ("beginning of string") and EOS ("end of string"). These special symbols are not in $\Sigma$ but can be used to match against the edges of the string. To be precise, the feature function $f_w$ is defined such that $f_w(v)$ counts the number of times that $w$ appears as a substring of $\text{BOS}\,v\,\text{EOS}$ (for any $w \in \mathcal{W}, v \in \Sigma^*$). By convention, $f_\epsilon(v) \stackrel{\text{def}}{=} |v| + 1$, so $\epsilon$ is considered to match once for each character of $v\,\text{EOS}$.

If $\mathcal{W}$ is too small, then the family $\mathcal{Q}$ that is defined from $\mathcal{W}$ could be empty: that is, $\mathcal{Q}$ may not contain any proper distributions $q_{\boldsymbol{\theta}}$ (see footnote 1). To prevent this, it suffices to insist that $\mathcal{W}$ includes all $n$-grams for some $n$ (a valid choice is $n = 0$, which requires only that $\mathcal{W}$ contains $\epsilon$). Now $\mathcal{Q}$ is not empty because giving sufficiently negative weights to these features (and weight 0 to all other features) will ensure that $q_{\boldsymbol{\theta}}$ has a finite normalizing constant $Z_\theta$.

Our experiments in this paper work with particular cases of $\mathcal{W}$. For our $n$-gram EP method,

(a) We take $\mathcal{W}$ to consist of all strings of length $n$ in the regular language $\Sigma^*\,\text{EOS}^?$, plus all non-empty strings of length $\leq n$ in the regular language $\text{BOS}\,\Sigma^*\,\text{EOS}^?$.

For penalized EP with variable-length strings, the set $\mathcal{W}$ corresponds to some specific collection of log-linear weights that we are encoding or updating:

(b) During EP message passing, we must construct $\text{ENCODE}(\boldsymbol{\theta})$ (where $\boldsymbol{\theta}$ represents a variable-to-factor message). Here we have $\mathcal{W} = \text{support}(\boldsymbol{\theta}) = \{w : \theta_w \neq 0\}$, as stated in section 3.5. Such WFSA encodings happen in sections 4.2 and 4.3.

(c) When we use the greedy growing method (section 6.1) to estimate $\boldsymbol{\theta}$ as an approximation to a variable belief, $\mathcal{W}$ is set explicitly by that method, as detailed in Appendix B.9 below.

(d) When we use proximal gradient (section 6.1) to estimate $\boldsymbol{\theta}$ as an approximation to a variable belief, $\mathcal{W}$ is the current active set of substrings $w$ for which we would *allow* $\theta_w \neq 0$. (Then the resulting WFSA encoding is used to help compute $\partial \text{D}(p \mid\mid q_{\boldsymbol{\theta}})/\partial\theta_w$ and so update $\theta_w$.) This active set consists of all strings $w$ that currently have $\theta_w \neq 0$, together with all single-character extensions of these $w$ and of $\epsilon$, and also all prefixes of these $w$.

### B.3 Modifying $\mathcal{W}$ into $\overline{\mathcal{W}}$

For generality, we will give our WFSA construction below (in Appendix B.4) so that it works with *any* set $\mathcal{W}$. In case $\mathcal{W}$ lacks certain properties needed for the construction, we construct a modified set $\overline{\mathcal{W}}$ that has these properties. Each substring in $\overline{\mathcal{W}}$ will correspond to a different arc in the WFSA.

Each $w \in \overline{\mathcal{W}}$ must consist of a **history** $h(w) \in \text{BOS}^?\Sigma^*$ concatenated with a **next character** $n(w) \in \Sigma'$. (These statements again use regular expression notation.) That is, $w = h(w)\,n(w)$.

Also, $\overline{\mathcal{W}}$ must be closed under prefixes. That is, if $w \in \overline{\mathcal{W}}$, then $h(w) \in \overline{\mathcal{W}}$, unless $h(w) \in \{\text{BOS}, \epsilon\}$ since these are not legal elements of $\overline{\mathcal{W}}$. For example, to include $w = \texttt{abc}$ in $\mathcal{W}$, we must include it in $\overline{\mathcal{W}}$ along with `ab` and `a`; or to include $w = \text{BOS}\,\texttt{ab}$ in $\mathcal{W}$, we must include it in $\overline{\mathcal{W}}$ along with $\text{BOS}\,\texttt{a}$.

This ensures that our simple construction below will give a WFSA that is able to detect $w$ one character at a time.

We convert the given $\mathcal{W}$ into $\overline{\mathcal{W}}$ by removing all strings that are not of the form $\text{BOS}^?\Sigma^*\Sigma'$, and then taking the closure under prefixes.

Again, let us be concrete about the particular $\mathcal{W}$ that we use in our experiments:

(a) For an $n$-gram model, we skip the above conversion and simply set $\overline{\mathcal{W}} = \mathcal{W}$. Even though $\overline{\mathcal{W}}$ does not then satisfy the prefix-closure property, the construction below will still work (see footnote 9).

(b) When we are encoding an infinite $\boldsymbol{\theta}$ during message passing, sparsity in $\boldsymbol{\theta}$ may occasionally mean that we must add strings when enlarging $\mathcal{W}$ into $\overline{\mathcal{W}}$.

(c,d) When we are estimating a new $\boldsymbol{\theta}$, the $\mathcal{W}$ that we use already has the necessary properties, so we end up with $\overline{\mathcal{W}} = \mathcal{W}$ just as in the $n$-gram model. We will use this fact in section B.6.

### B.4 Encoding $(\mathcal{W}, \boldsymbol{\theta})$ as a WFSA

Given $\mathcal{W}$ and a parameter vector $\boldsymbol{\theta}$ (indexed by $\mathcal{W}$, not by $\overline{\mathcal{W}}$), we will now build $\text{ENCODE}(\boldsymbol{\theta})$, a WFSA that can be used to score strings $v \in \Sigma^*$ (section 3.2). Related constructions are given by Mohri (2005) for the unweighted case and by Allauzen et al. (2003) for language modeling.

For any non-empty string $w \in \text{BOS}^?\Sigma^*$, define the **backoff** $b(w)$ to be the longest proper suffix of $w$ that is a proper prefix of some element of $\overline{\mathcal{W}}$. Define the **bridge** $\bar{b}(w)$ similarly except that it need not be a *proper* suffix of $w$ (it may equal $w$ itself).[9] In addition, we define $\bar{b}(w)$ to be $\square$ when $w$ is any string ending in EOS.

Finally, for any $w \in \overline{\mathcal{W}}$, define the **weight** $k(w) = \exp \sum_{w' \in (\text{suffixes}(w) \cap \mathcal{W})} \theta_{w'}$. This weight will be associated with the arc that consumes the final character of a copy of substring $w$, since consuming that character (while reading the input string $v$) means that the WFSA has detected an instance of substring $w$, and thus all features in $\mathcal{W}$ fire that match against suffixes of $w$.

We can now define the WFSA $\text{ENCODE}(\boldsymbol{\theta})$, with weights in the $(+, \times)$ semiring. Note that this WFSA is unambiguous (in fact deterministic).[10]

- The **set of states** is $H = \{h(w) : w \in \overline{\mathcal{W}}\}$ together with the distinguished **final state** $\square$ (see Appendix B.1).
- The **initial state** is BOS, if $\text{BOS} \in H$. Otherwise the initial state is $\epsilon$, which is in $H$ thanks to the prefix-closure property.
- The **ordinary arcs** are $\{h(w) \xrightarrow{n(w)/k(w)} \bar{b}(w) : w \in \overline{\mathcal{W}}\}$. As explained in Appendix B.1, the notation $n(w)/k(w)$ means that the arc reads the character $n(w)$ (possibly EOS) with weight $k(w)$. Note that we have one ordinary arc for every $w \in \overline{\mathcal{W}}$.
- The **failure arcs** are $\{h \xrightarrow{\phi/1} b(h)\}$ where $h \in H$ and $h \neq \epsilon$.
- If $\epsilon$ is in $H$, there is also a **default arc** $\epsilon \xrightarrow{\rho/k(\epsilon)} \epsilon$.

A **default arc** is one that has the special label $\rho$. It is able to consume any character in $\Sigma'$ that is *not* the label of any ordinary arc leaving the same state. To avoid further discussion of this special case, we will assume from here on that the single default arc has been implemented by replacing it with an explicit collection of ordinary arcs labeled with the various non-matching characters (perhaps including EOS), and each having the same weight $k(\epsilon)$. Thus, the state $\epsilon$ has $|\Sigma'|$ outgoing arcs in total.

A **failure arc** (Allauzen et al., 2003) is one that has the special label $\phi$. An automaton can traverse it if—and only if—no other option is available. That is, when the next input character is $c \in \Sigma'$, the automaton may traverse the $\phi$ arc from its current state

[9] $b(w)$ or $\bar{b}(w)$ may become a state in the WFSA. If so, we need to ensure that there is a path from this state that reads the remaining characters of any "element of $\overline{\mathcal{W}}$"—call it $w'$—of which it is a proper prefix. The prefix closure property guarantees this by stating that if $w' \in \overline{\mathcal{W}}$ then $h(w') \in \overline{\mathcal{W}}$ (and so on recursively). A weaker property would do: we only need $h(w') \in \overline{\mathcal{W}}$ if there actually exists some $w \in \overline{\mathcal{W}}$ such that $b(w)$ or $\bar{b}(w)$ is a *proper* prefix of $h(w')$. This condition never applies for the $n$-gram model.

[10] As a result, the + operator is never actually needed to define this WFSA's behavior. However, by specifying the $(+, \times)$ semiring, we make it possible to combine this WFSA with other WFSAs (such as $p$) that have weights in the same semiring.

$h$ unless there exists an ordinary $c$ arc from $h$. In contrast to $\rho$, traversing the $\phi$ arc does not actually consume the character $c$; the automaton must try again to consume it from its new state.

The construction allows the automaton to back off repeatedly by following a path of multiple $\phi$ arcs, e.g., abc $\xrightarrow{\phi/1}$ bc $\xrightarrow{\phi/1}$ c $\xrightarrow{\phi/1}$ $\epsilon$. Thus, the automaton can always manage to read the next character in $\Sigma'$, if necessary by backing off all the way to the $\epsilon$ state and then using the $\rho$ arc.

In the case of a fixed-order $n$-gram model, each state has explicit outgoing arcs for all symbols in $\Sigma'$, so the failure arcs are never used and can be omitted in practice. For the variable-order case, however, failure arcs can lead to a considerable reduction in automaton size. It is thanks to failure arcs that ENCODE($\boldsymbol{\theta}$) has only $|\overline{\mathcal{W}}|$ ordinary arcs (counting the default arc if any). Indeed, this is what is counted by (8).[11]

### B.5 Making do without failure arcs

Unfortunately, our current implementation does not use failure arcs, because we currently rely on a finite-state infastructure that does not fully support them (Appendix D). Thus, our current implementation enforces another closure property on $\overline{\mathcal{W}}$: if $w \in \overline{\mathcal{W}}$, then $h(w)\,c \in \overline{\mathcal{W}}$ for *every* $c \in \Sigma'$. This ensures that failure arcs are unnecessary at all states for just the same reason that they are unnecessary at $\epsilon$: *every* state now has explicit outgoing arcs for all symbols in $\Sigma'$.

*This slows down our current PEP implementation*, potentially by a factor of $O(|\Sigma|)$, because it means that adding the abc feature forces us to construct an ab state with outgoing arcs for all characters in $\Sigma$, rather than outgoing arcs for just c and $\phi$. By constrast, there is no slowdown for $n$-gram EP, because then $\mathcal{W}$ already has the new closure property. Our current experiments therefore *underestimate* the speedup that is possible with PEP.

We expect a future implementation to support failure arcs; so in the following sections, we take care to give constructions and algorithms that handle them.

[11] The number of arcs is a good measure of the size of the encoding. The worst-case runtime of our finite-state operations is generally proportional to the number of arcs. The number of states $|H|$ is smaller, and so is the number of failure arcs, since each state has at most one failure arc.

### B.6 $\mathcal{W}$ and $\mathcal{A}$ parameterizations are equivalent

Given $\mathcal{W}$ (without $\boldsymbol{\theta}$), we can construct the unweighted FSA $\mathcal{A}$ exactly in the section above, except that we omit the weights. How do we then find $\boldsymbol{\theta}$? Recall that our optimization methods (section 6) actually tune parameters $\boldsymbol{\theta}^{\mathcal{A}}$ associated with the arcs of $\mathcal{A}$. In this section, we explain why this has the same effect as tuning parameters $\boldsymbol{\theta}^{\mathcal{W}}$ associated with the substrings in $\mathcal{W}$.

Given a log-linear distribution $q$ defined by the $\mathcal{W}$ features with a particular setting of $\boldsymbol{\theta}^{\mathcal{W}}$, it is clear that the same $q$ can be obtained using the $\mathcal{A}$ features with some setting of $\boldsymbol{\theta}^{\mathcal{A}}$. That is essentially what the previous section showed: for each arc or final state $a$ in $\mathcal{A}$, take $\theta^{\mathcal{A}}_a$ to be the log of $a$'s weight under the construction of the previous section.

The converse is also true, provided that $\overline{\mathcal{W}} = \mathcal{W}$. That is, given a log-linear distribution $q$ defined by the $\mathcal{A}$ features with a particular setting of $\boldsymbol{\theta}^{\mathcal{A}}$, we can obtain the same $q$ using the $\mathcal{W}$ features with some setting of $\boldsymbol{\theta}^{\mathcal{W}}$. This is done as follows:

1. Produce a weighted version of $\mathcal{A}$ such that the weight of each arc or final state $a$ is $\exp \boldsymbol{\theta}^{\mathcal{A}}_a$.

2. Modify this WFSA such that it defines the same $q$ function but all $\phi$ arcs have weight 1.[12] This can be done by the following "weight pushing" construction, similar to (Mohri, 2000). For each state $h$, let $k_h > 0$ denote the product weight of the maximum-length path from $h$ labeled with $\phi^*$.[13] Now for every arc, from some $h$ to some $h'$, multiply its weight by $k_{h'}/k_h$. This preserves the weight of all accepting paths in the WFSA (or more precisely, divides them all by the constant $k_{h_0}$ where $h_0$ is the initial state), and thus preserves the distribution $q$.

3. For each $w \in \mathcal{W}$, let $k(w)$ denote the modified weight of the ordinary arc associated with $w$ in the topology. Recall that the previous section constructed these arc weights $k(w)$ from $\theta^W$. Reversing that construction, we put $\theta^{\mathcal{W}}_w =$

[12] Remark: Since this is possible, we would lose no generality by eliminating the features corresponding to the $\phi$ arcs. However, including those redundant features may speed up gradient optimization.

[13] Such paths always have finite length in our topology (possibly 0 length, in which case $k_h = 1$).

$\log k(w) - \log k(w')$, where $w'$ is the longest proper suffix of $w$ that appears in $\overline{\mathcal{W}}$. If there is no such suffix, we put $\theta_w^{\mathcal{W}} = \log k(w)$.

So we have seen that it is possible to convert back and forth between $\boldsymbol{\theta}^{\mathcal{W}}$ and $\boldsymbol{\theta}^{\mathcal{A}}$. Hence, the family $\mathcal{Q}$ that is defined by $\mathcal{A}$ (as in section 3.4) is identical to the family $\mathcal{Q}$ that would have been defined by $\mathcal{W}$ (as in section 3.3), provided that $\overline{\mathcal{W}} = \mathcal{W}$. It is merely a reparameterization of that family.

Therefore, we can use the method of section 6 to find our optimal distribution

$$\operatorname*{argmin}_{q \in \mathcal{Q}} \mathrm{D}(p \mid\mid q) \tag{10}$$

In other words, we represent $q$ by its $\boldsymbol{\theta}^{\mathcal{A}}$ parameters rather than its $\boldsymbol{\theta}^{\mathcal{W}}$ parameters. We optimize $q$ by tuning the $\boldsymbol{\theta}^{\mathcal{A}}$ parameters. (It is not necessary to *actually* convert back to $\boldsymbol{\theta}^{\mathcal{W}}$ by the above construction; we are simply showing the equivalence of two parameterizations.)

### B.7 Weighted and probabilistic FSAs are equivalent

Section 6 gives methods for estimating the weights $\boldsymbol{\theta}$ associated with a WFSA $\mathcal{A}$. In the experiments of this paper, $\mathcal{A}$ is always derived from some $\mathcal{W}$. However, section 3.4 explains that our EP method can be used with features derived from any arbitrary $\mathcal{A}$ (e.g., arbitrary regular expressions and not just $n$-grams). So we now return to that general case.

The gradient methods in section 6 search for arbitrary WFSA parameters. However, the closed-form methods in that section appear at first to be more restrictive: they always choose weights $\boldsymbol{\theta}$ such that $\textsc{Encode}(\boldsymbol{\theta})$ is actually a probabilistic FSA. In other words, the weights yield "locally normalized" probabilities on the arcs and final states of $\mathcal{A}$ (section 6, footnote 6).

**Definition of probabilistic FSAs.** This property means that at each state $h \neq \square$, the WFSA defines a probability distribution over the next character $c \in \Sigma'$. Thus, one can sample a string from the distribution $q_{\boldsymbol{\theta}}$ by taking a random walk on the WFSA from the initial state to $\square$.

Unlike previous papers (Eisner, 2002; Cotterell et al., 2014), we cannot simply say that the arcs from state $h$ are weighted with probabilities that sum to 1. The difficulty has to do with failure arcs, which may even legitimately have weight > 1.

The following extended definition is general enough to cover cases where the WFSA may be nondeterministic as well as having failure arcs. This general definition may be useful in future work. Even for this paper, $\mathcal{A}$ is permitted to be nondeterministic—section 3.4 only requires $\mathcal{A}$ to be unambiguous and complete.)

Define an **augmented transition** with **signature** $h \overset{c/k}{\rightsquigarrow} h'$ to be any path from state $h$ to state $h'$, with product weight $k$, that is labeled with a sequence in $\phi^* c$ (where $c \in \Sigma'$), such that there is no path from $h$ that is labeled with a *shorter* sequence in $\phi^* c$. This augmented transition can be used to read symbol $c$ from state $h$.

We say that the WFSA is a **probabilistic finite-state acceptor (PFSA)** if for each state $h$, the augmented transitions from $h$ have total weight of 1.

Note that one can eliminate failure arcs from a WFSA by replacing augmented transitions with actual transitions. However, that would expand the WFSA and enlarge the number of parameters. Our goal here is to discuss local normalization within machines that retain the compact form using failure arcs.

**Performing local normalization.** We now claim that restricting to PFSAs does not involve any loss of generality.[14] More precisely, we will show that *any* WFSA that defines a (possibly unnormalized) probability distribution, such as $\textsc{Encode}(\boldsymbol{\theta})$, can be converted to a PFSA of the same topology that defines a normalized version of the same probability distribution. This is done by modifying the weights as follows.

For every state $h$ of the WFSA, define the *backward probability* $\beta(h)$ to be the total weight of all suffix paths from $h$ to $\square$. Note that if $h_0$ is the initial state, then $\beta(h_0)$ is the WFSA's normalizing constant $Z_{\boldsymbol{\theta}}$, which is finite by assumption.[15] It follows that $\beta(h)$ is also finite at any state that is reachable

[14] Eisner (2002) previously pointed this out (in the setting of WFSTs rather than WFSAs). However, here we generalize the claim to cover WFSAs with failure arcs.

[15] Our PFSA will define a normalized distribution, so it will not retain any memory of the value $Z_{\boldsymbol{\theta}}$.

from the initial state (assuming that all arc weights are positive).

One can compute $\beta(h)$ using the recurrence $\beta(h) = \sum_i k_i \cdot \beta(h_i)$ where $i$ ranges over the augmented transitions from $h$ and the $i$th augmented transition has signature $h \overset{c_i/k_i}{\rightsquigarrow} h_i$. As the base case, $\beta(\square) = 1$. This gives a linear system of equations[16] that can be solved for the backward probabilities. The system has a unique solution provided that the WFSA is trim (i.e., all states lie on some accepting path).

It is now easy to modify the weights of the ordinary arcs. For each ordinary arc $h \overset{c/k}{\rightarrow} h'$, change the weight to $\frac{k \cdot \beta(h')}{\beta(h)}$.

Finally, consider each failure arc $h \overset{\phi/k}{\longrightarrow} h'$. Let $k' = \sum_i k_i \cdot \beta(h_i)$, where $i$ ranges over the "blocked" augmented transitions *from* $h'$—those that *cannot* be taken after this failure arc. The $i$th augmented transition has signature $h' \overset{c_i/k_i}{\rightsquigarrow} h_i$, and is blocked if $c_i \in \Sigma'$ can be read directly at $h$. It follows that the paths from $h'$ that *can* be taken after this failure arc will have total probability $1 - \frac{k'}{\beta(h')}$ in the new PFST. Change the weight of the failure arc to $\frac{k \cdot \beta(h')}{\beta(h)} \,/\, (1 - \frac{k'}{\beta(h')})$. As a result, the total probability of all suffix paths from $h$ that start with this failure arc will be $\frac{k \cdot \beta(h')}{\beta(h)}$ as desired.

**When to use the above algorithms.** For working with approximate distributions during EP or PEP, it is not necessary to *actually* compute backward probabilities on parameterized versions of $\mathcal{A}$ or convert these WFSAs to PFSA form. We are simply showing the equivalence of the WFSA and PFSA parameterizations.

Even so, these are fundamental computations for WFSAs. They are needed in order to compute a WFSA's normalizing constant $Z_{\boldsymbol{\theta}}$, to compute its expected arc counts (the forward-backward algorithm), or to sample strings from it.

Indeed, such computations are used in section 6, though they are not applied to $\mathcal{A}$ but rather to other WFSAs that are built by combining $\mathcal{A}$ with the distribution $p$ that is to be approximated. If *these* WFSAs contained failure arcs, then we *would* need the extended algorithms above. This could happen, in principle, if $p$ as well as $\mathcal{A}$ were to contain failure arcs.

### B.8 Fitting PFSA parameters

In order to estimate a WFSA with given topology $\mathcal{A}$ that approximates a given distribution $p$, the previous section shows that it suffices to estimate a PFSA. Recall from section 3.1 that we are looking for maximum-likelihood parameters.

Section 6 sketched a closed-form method for finding the maximum-likelihood PFSA parameters. Any string $v \in \Sigma^*$ has a single accepting path in $\mathcal{A}$, leading to an integer feature vector $\boldsymbol{f}(v)$ that counts the number of times this path traverses each arc of $\mathcal{A}$ (including the final EOS arc as described in Appendix B.1). As section 6 explains, it is possible to compute the expected feature vector $\mathbb{E}_{v \sim p}[\boldsymbol{f}(v)]$ using finite-state methods. Now, at each state $h$ of $\mathcal{A}$, set the outgoing arcs' weights to be proportional to these expected counts, just as in section 6. The parameters $\boldsymbol{\theta}$ are then the logs of these weights.

Unfortunately, this construction does not work when $\mathcal{A}$ has failure arcs—which are useful, e.g., for defining variable-order Markov models. In this case we do *not* know of a closed-form method for finding the maximum-likelihood parameters under a local normalization constraint. The difficulty arises from the fact that a single arc (ordinary arc or failure arc) may be used as part of several augmented transitions. The constrained maximum-likelihood problem can be formulated using the method of Lagrange multipliers, which leads to an elegant system of equations. Unfortunately, this system is not linear, and we have not found an efficient way to solve it exactly. (Using iterative update does not work because the desired fixpoint is unstable.)

[16] For greater efficiency, it is possible to set up this system of equations in a way that is as sparse as the WFSA itself. For each state $h$, we constrain $\beta(h)$ to equal a linear combination of other $\beta$ values. For a state with just a few ordinary arcs plus a failure arc, we would like to have just a few summands in this linear combination (*not* one summand for each $c \in \Sigma'$).

The linear combination includes a summand $k_j \cdot \beta(h_j)$ for each ordinary arc $h \overset{c_j/k_j}{\longrightarrow} h_j$. It also includes a summand $k \cdot \beta(h')$ for each failure arc $h \overset{\phi/k}{\longrightarrow} h'$. However, we must correct this last summand by recognizing that some augmented transitions from the backoff state $h'$ *cannot* be taken after this failure arc. Thus, the linear combination also includes a corrective summand $-k \cdot k_i \cdot \beta(h_i)$ for each augmented transition $h' \overset{c_i/k_i}{\rightsquigarrow} h_i$ that is "blocked" in the sense that $c_i \in \Sigma'$ can be read directly at $h$.

To rescue the idea of closed-form estimation, we have two options. One is to eliminate failure arcs from $\mathcal{A}$, which expands the parameter set and leads to a richer family of distributions, at some computational cost. Our current experiments do this for reasons explained in Appendix B.5.

The other option is to apply a conventional approximation from backoff language modeling. Consider a backoff trigram model. At the state `b`, it is conventional to estimate the probabilities of the outgoing arcs $c$ according to the relative counts of the bigrams `b`$c$. This is an approximation that does not quite find the maximum-likelihood parameters: it ignores the fact that `b` is a backoff state, some of whose outgoing transitions may be "blocked" depending on how `b` was reached. For example, some tokens of `b`$c$ are actually part of a trigram `ab`$c$, and so would be read by the $c$ arc from `ab` rather than the $c$ arc from `b`. However, the approximation counts them in both contexts.

It is straightforward to generalize this approximation to any $\mathcal{A}$ topology with $\phi$ arcs (provided that there are no cycles consisting solely of $\phi$ arcs). Simply find the expected counts $\mathbb{E}_{v\sim p}[\boldsymbol{f}(v)]$ as before, but using a *modified* version of $\mathcal{A}$ where the $\phi$ arcs are treated as if they were $\epsilon$ arcs. This means that $\mathcal{A}$ is no longer unambiguous, and a single string $v$ will be accepted along multiple paths. This leads to the double-counting behavior described above, where the expected features count both ordinary paths and backoff paths.

As before, at each state $h$ of $\mathcal{A}$, set the outgoing arcs' weights to be proportional to these (inflated) expected counts.

Finally, because the semantics of $\phi$ results in blocked arcs, we must now adjust the weights of the failure arcs so that the augmented transitions from each state will sum to 1. Mark each failure arc as "dirty," i.e., its weight has not yet been adjusted. To "clean" the failure arc $h \xrightarrow{\phi/k} h'$, divide its weight by $1 - k'$ where $k'$ is the total weight of all blocked augmented transitions from $h'$. When computing the weight of an augmented transition from $h'$, it is necessary to first clean any failure arcs that are themselves part of the augmented transition. This recursive process terminates provided that there are no $\phi$-cycles.

### B.9 Greedily growing $\mathcal{W}$

Section 6.1 sketches a "closed-form with greedy growing" method for approximately minimizing the objective (3), where $\Omega(\boldsymbol{\theta})$ is given by (8).

The method is conceptually similar to the active set method. Both methods initialize $\mathcal{W} \supseteq \{\epsilon, \text{BOS}\}$, and then repeatedly expand the current $\mathcal{W}$ with new strings (yellow nodes in Figure 2) that are rightward extensions of the current strings (green nodes).[17]

The active set method relies on a proximal gradient step to update the weights of the yellow nodes. This also serves to select the yellow nodes that are worth adding—we remove those whose weight remains at 0.

In contrast, the closed-form method updates the weights of the yellow nodes by adding them to $\mathcal{W}$ and running the closed-form estimation procedure of Appendix B.8. This procedure has no structured-sparsity penalty, so it is not able to identify less useful nodes by setting their weights to 0. As an alternative, it would be reasonable to identify and remove less useful nodes by entropy pruning (Stolcke, 1998), similar to the split-merge EM procedure of Petrov et al. (2006). At present we do not do this. Rather, we use a heuristic test to decide whether to add each yellow node in the first place.

Our overall method initializes $\mathcal{W}$ and then enumerates strings $w \in \text{BOS}^?\Sigma^*\text{EOS}^?$ in a heuristic order. We add $w$ to $\mathcal{W}$ if we *estimate* that this will improve the objective. Every so often, we evaluate the *actual* objective using the current $\mathcal{W}$, and test whether it has improved since the last evaluation. In other words, has it helped to add the latest batch of yellow strings? (Our current implementation uses batches of size 20.) If not, then we stop and return the current parameters. Stopping means that the average entropy reduction per newly added string $w$ has diminished below the penalty rate $\lambda$ in (3).

**Enumerating strings.** We take care to ensure that $\mathcal{W}$ remains closed under prefixes. A step of enumeration consists of popping the highest-priority element $w$ from the priority queue. Whenever we elect to add a string $w$ to $\mathcal{W}$ (including when we initialize

[17] We require $\text{BOS} \in \mathcal{W}$ only so that we can extend it rightward into BOS `a`, BOS `ab`, etc. When constructing a WFSA, Appendix B.3 will remove BOS from $\mathcal{W}$ to obtain $\overline{\mathcal{W}}$.

$\mathcal{W}$ by adding its initial members), we enqueue all possible rightward extensions $wc$ for $c \in \Sigma'$.

Our strategy is to approximate $p$ by learning nonzero feature weights for its most common substrings first. So the priority of $w$ is $e(w) \stackrel{\text{def}}{=} \mathbb{E}_{v \sim p}[f_w(v)]$, the expected count of the substring $w$.

For simplicity, let us assume that $p$ is given by an $\epsilon$-free WFSA. At the start of our method, we run the forward-backward algorithm on $p$ to compute $\alpha(s)$ and $\beta(s)$ for each state $s$ of this WFSA. $\alpha(s)$ is the total probability of all **prefix paths** from the initial state $h_0$ to $s$, while $\beta(s)$ is the total probability of all **suffix paths** from $s$ to $\square$. Since $p$ may be cyclic, it is necessary in general to solve a linear system to obtain these values (Eisner, 2002). We set $Z = \alpha(h_0)$, the normalizing constant of $p$.

We need to be able to compute the priority of a string when we add it to the queue. To enable this, the entry for $w$ on the priority queue also maintains a map $m_w$ that is indexed by states $s$ of the WFSA for $p$. The entry $m_w[s]$ is the total weight of all prefix paths that reach state $s$ on a string with suffix $w$.[18] (The key $s$ may be omitted from the map if there are no such paths, i.e., if $m_w[s] = 0$.) From $m_w$, we can easily compute the priority of $w$ as $e(w) = \sum_s m_w[s] \cdot \beta(s)/Z$. When adding $w$ to $\mathcal{W}$ causes us to enqueue $wc$, we must create the map $m_{wc}$. This is derived from $m_w$ by setting $m_{wc}[s'] = \sum_s m_w[s] \cdot$ (total weight of all augmented transitions $s \stackrel{c/k}{\rightsquigarrow} s'$). The base case $m_\epsilon$ is given by $m_\epsilon[s] = \alpha(s)$. The base case $m_{\text{BOS}}$ is given by $m_{\text{BOS}}[h_0] = 1$.

**Testing whether to add $w$.** When we pop $w$ from the priority queue, is it worth adding to $\mathcal{W}$? Recall that we have already computed $e(w)$ as the priority of $w$. Adding $w$ means that we will be able to model the final character $n(w)$ in the context of the history $h(w)$ without backing off. Under the methods of the previous section, this will change (3) by roughly $e(w) \cdot (\log P_{\text{old}} - \log P_{\text{new}}) + \lambda$, where $P_{\text{new}} = e(w)/e(h(w))$ and $P_{\text{old}} = e(w')/e(h(w'))$, where $w'$ is the longest proper suffix of $w$ that is currently in $\mathcal{W}$.[19] Our heuristic is to add $w$ if this estimated change is negative (i.e., an improvement). In other words, the increase in the model size penalty $\lambda \cdot |\mathcal{W}|$ needs to be outweighted by a reduction in perplexity for the last character of $w$.

**Evaluating the objective.** Given $\mathcal{W}$, we could evaluate the objective (3) using the construction given in section 6. This would require estimating $\boldsymbol{\theta}$ and constructing $\text{ENCODE}(\boldsymbol{\theta})$ using the methods of Appendix B.8.

Fortunately, it is possible to use a shortcut, since we are specifically doing variable-order Markov modeling and we have already computed $e(w)$ for all $w \in \mathcal{W}$. If we estimate the parameters using the backoff language model technique suggested in Appendix B.8, then the minimization objective is given by a constant plus

$$-\left(\sum_{w \in \mathcal{W}} e(w) \cdot \left(\log \frac{e(w)}{e(h(w))} - \log \frac{e(w')}{e(h(w'))}\right)\right) + |\mathcal{W}| \tag{11}$$

where $w'$ denotes the longest proper suffix of $w$ that is also in $\mathcal{W}$.[20]

The summand for $w$ claims that for each of the $e(w)$ instances of $w$, the model will estimate the probability of the last character of $w$ using $e(w)/e(h(w))$. This summand overrides the $w'$ summand, which incorrectly claimed to estimate the same $e(w)$ cases using the backed-off estimate $e(w')/e(h(w'))$. Thus, the $w$ summand subtracts the backed-off estimate based on $w'$ and replaces it with its own estimate based on $w$. Of course, this summand may be corrected in turn by even longer strings in $\mathcal{W}$.

In principle, one can maintain (11) incrementally as $\mathcal{W}$ changes. Adding a new string to $\mathcal{W}$ will add a new term, and it may modify old terms for which the new string replaces the old value of $w'$.

**Making do without failure arcs.** Since our current implementation cannot use failure arcs (see Appendix B.5), we cannot extend $w \in \mathcal{W}$ by adding

[18] Ordinarily, this means all paths of the form $h_0 \xrightarrow{\Sigma^* w} s$. However, if $w$ has the form $\text{BOS}\, w'$, then it means all paths of the form $h_0 \xrightarrow{w'} s$, meaning that that $w'$ must match at the *start* of the path.

[19] In the case $w' = \epsilon$, we take $P_{\text{old}} = 1/|\Sigma'|$, the 0-gram probability of an unknown character.

[20] The summand for $w = \epsilon$ must be handled as a special case, namely $e(w) \cdot \log(1/|\Sigma'|)$.

a single string $wc$ for $c \in \Sigma'$. We must add *all* of these strings at once. This triggers a slight change to the architecture. When we add $w$ to $\mathcal{W}$, we do not enqueue all extensions $wc$ to the priority queue. Rather, we enqueue $w$ itself, with priority $e(w)$. This now represents a "bundle of extensions." When we pop $w$, we decide whether to add the full set of extensions $wc$ to $\mathcal{W}$, by estimating the total benefit of doing so.

### B.10 Making do without $d$-tape WFSAs

Footnote 5 noted that it is possible to reformulate any factor graph to use only factors of degree 2. This follows a standard transformation that reformulates any problem in constraint satisfaction programming (CSP) to use only binary constraints.

**General construction for factor graphs.** To eliminate a factor $F$ that depends on variables $V_1, \ldots, V_d$, one can introduce a new variable $V_F$ whose value is a $d$-tuple. $V_F$ is related to the original variables by binary factors that ensure that $V_F$ is specifically the $d$-tuple $(V_1, \ldots, V_d)$. (That is, the first component of $V_F$ must equal $V_1$, the second must equal $V_2$, etc.) The old factor $F$ that examined $d$ variables is now replaced by a unary factor that examines the $d$-tuple.

**Construction in the finite-state case.** In the case of graphical models over strings, a factor of degree $d$ is a $d$-tape weighted finite-state machine. To eliminate a factor $F$ of degree $d > 2$ from the factor graph, introduce a new variable $V_F$ that encodes a path in the machine $F$. Since a path is just a sequence of arcs, the value of this variable is a *string* over a special alphabet, namely the set of arcs in $F$.

Let $V_1, \ldots, V_d$ denote the neighbors of $F$, so that a path in $F$ accepts a tuple of strings $(v_1, \ldots, v_d)$. For each $1 \leq i \leq d$, introduce a new binary factor connected to $V_F$ and $V_i$. This factor scores the pair of strings $(v_F, v_i)$ as 1 or 0 according to whether $v_i$ is the $i$th element of the tuple accepted by the path $v_F$.

Finally, replace $F$ with a new unary factor $F'$ connected to $V_F$. This unary factor scores an arc sequence $v_F$ as 0 if $v_F$ is not a path from the initial state of $F$ to a final state of $F$. Oherwise it returns the weight of $v_F$ as an accepting path in $F$.

| $n$-gram | weight | level | $n$-gram | weight | level |
|---|---|---|---|---|---|
| z | -0.05 | 1 | ip | 0.88 | 2 |
| iz | 0.0 | 2 | Sip | 0.85 | 3 |
| p | 0.84 | 1 | tSip | 0.84 | 4 |

Table 1: Some of the active $n$-gram features in PEP's belief about the underlying representation of the word *chip*. The correct answer is tSip.

**Implementation of the new finite-state factors.**[21] The binary factor connected to $V_F$ and $V_i$ implements a simple homomorphism. (Specifically, if an arc $a$ of $F$ is labeled as $\xrightarrow{c_1:c_2:\cdots:c_d/k}$, then this binary factor always maps the symbol $a$ to the string $c_i$.) Therefore, this binary factor can be implemented as a 1-state deterministic finite-state transducer.

The unary factor $F'$ can be implemented as a WFSA that has the same topology as $F$, the same initial and final states, and the same weights. Only the labels are different. If an arc $a$ of $F$ is labeled as $\xrightarrow{c_1:c_2:\cdots:c_d/k}$, then the corresponding arc $a'$ in $F'$ is labeled as $\xrightarrow{a/k}$.

## C Further Results

Table 1 illustrates a sample of the $n$-grams that PEP pulls out when approximating one belief.

Figures 6 and 7 compare the different algorithms from section 6.

## D Code Release

Code for performing EP and PEP as approximations to loopy BP will be released via the first author's website. This implementation includes a generic library for our automaton constructions, e.g. construction of variable length $n$-gram machines and minimization of the KL-divergence between two machines, built on top of PyFST (Chahuneau, 2013), a Python wrapper for OpenFST (Allauzen et al., 2007).

While OpenFST supports failure and default arcs,[22] PyFST currently does not. We hope to resolve this in future, for reasons discussed in Appendix B.5.

[21] Because we are now working with multi-tape machines, we drop Appendix B.1 and assume the usual machine format.

[22] http://www.openfst.org/twiki/bin/view/FST/FstAdvancedUsage#Matchers

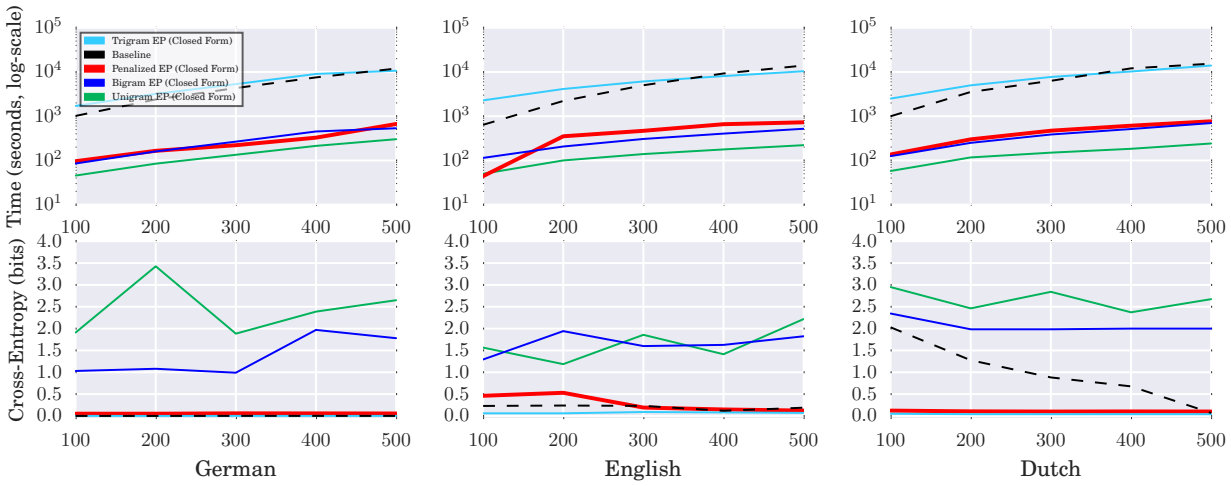


Figure 6: A version of Figure 3 that uses the closed-form methods to estimate $\boldsymbol{\theta}$, rather than the gradient-based methods. The same general pattern is observed.

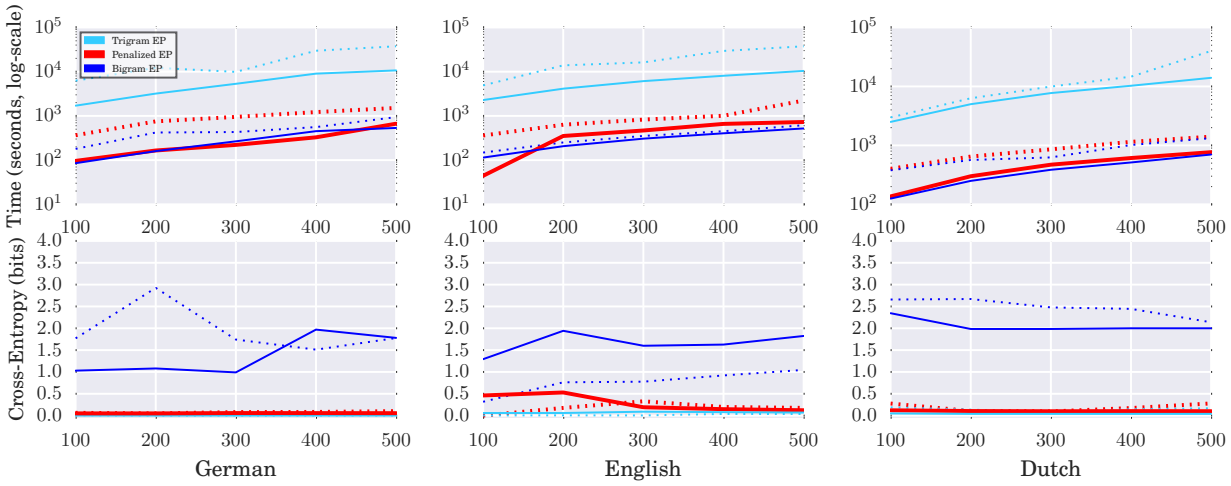


Figure 7: A comparison of the key curves from Figures 3 and 6. The dotted lines show the gradient-based methods, while the solid lines show the closed-form methods. The closed-form method is generally a bit faster, and has comparable accuracy.