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),

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

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., Σ^*), 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.

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

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

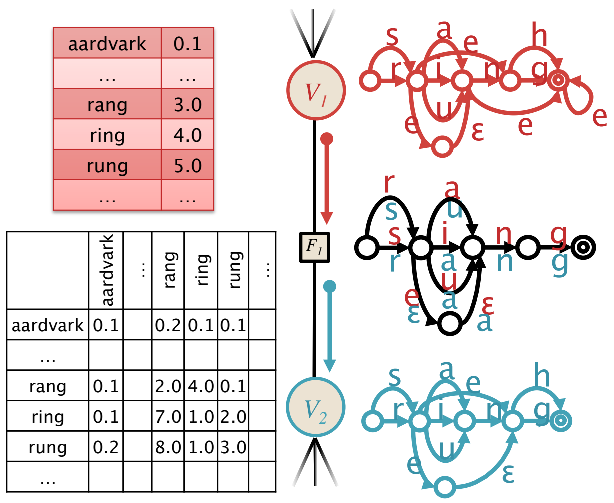
All of these objects are functions over some value space. We use $v \in \Sigma^*$ to refer to the values. Minka uses $\theta \in \mathbb{R}$, which is unrelated to our $\boldsymbol{\theta}$.

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 $\boldsymbol{\theta}_{X \to F}$



aardvark	0.1
…	…
rang	3.0
ring	4.0
rung	5.0
…	…

	aardvark	…	rang	ring	rung	…
aardvark	0.1		0.2	0.1	0.1	
…						
rang	0.1		2.0	4.0	0.1	
ring	0.1		7.0	1.0	2.0	
rung	0.2		8.0	1.0	3.0	
…						

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.

and $\boldsymbol{\theta}_{F \to X}$ would be 0 except for features of $V_i \in \mathcal{N}(F)$.

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 $\Sigma' \overset{\text{def}}{=} \Sigma \cup \{\text{EOS}\}$ 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 Σ 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, $f_\epsilon(v) \overset{\text{def}}{=} |v| + 1$, so ϵ 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 ϵ). 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_θ.

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 nonempty 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 θ_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 ϵ, 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 $\texttt{ab}$ and $\texttt{a}$; 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 ϵ, 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 ϵ 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 ρ. It is able to consume any character in Σ' 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 ϵ has $|\Sigma'|$ outgoing arcs in total.

A **failure arc** (Allauzen et al., 2003) is one that has the special label ϕ. 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 ϕ 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 ρ, traversing the ϕ 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 ϕ arcs, e.g., $\texttt{abc} \xrightarrow{\phi/1} \texttt{bc} \xrightarrow{\phi/1} \texttt{c} \xrightarrow{\phi/1} \epsilon$. Thus, the automaton can always manage to read the next character in Σ', if necessary by backing off all the way to the ϵ state and then using the ρ arc.

In the case of a fixed-order n-gram model, each state has explicit outgoing arcs for all symbols in Σ', 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 $\textsc{Encode}(\boldsymbol{\theta})$ 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 ϵ: *every* state now has explicit outgoing arcs for all symbols in Σ'.

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 Σ, rather than outgoing arcs for just `c` and ϕ. 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 $\theta_a^{\mathcal{A}}$ 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 $\exp \boldsymbol{\theta}_a^{\mathcal{A}}$.

2. Modify this WFSA such that it defines the same q function but all ϕ 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 ϕ^*.[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 θ^{W}. Reversing that construction, we put $\theta_w^{\mathcal{W}} =$

[12]Remark: Since this is possible, we would lose no generality by eliminating the features corresponding to the ϕ 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 $\theta^{\mathcal{W}}_w = \log k(w)$.

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 $\text{ENCODE}(\boldsymbol{\theta})$ 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 $\text{ENCODE}(\boldsymbol{\theta})$, 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 ith 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 $h \xrightarrow{\phi/k} h'$. 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 ith 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 β 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 $h \xrightarrow{c_j/k_j} h_j$. It also includes a summand $k \cdot \beta(h')$ for each failure arc $h \xrightarrow{\phi/k} h'$. 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 $\texttt{b}$, it is conventional to estimate the probabilities of the outgoing arcs c according to the relative counts of the bigrams $\texttt{b}c$. This is an approximation that does not quite find the maximum-likelihood parameters: it ignores the fact that $\texttt{b}$ is a backoff state, some of whose outgoing transitions may be "blocked" depending on how $\texttt{b}$ was reached. For example, some tokens of $\texttt{b}c$ are actually part of a trigram $\texttt{ab}c$, and so would be read by the c arc from $\texttt{ab}$ rather than the c arc from $\texttt{b}$. However, the approximation counts them in both contexts.

It is straightforward to generalize this approximation to any $\mathcal{A}$ topology with ϕ arcs (provided that there are no cycles consisting solely of ϕ 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 ϕ arcs are treated as if they were ϵ 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 ϕ 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 $h \overset{\phi/k}{\to} h'$, 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 ϕ-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 $\mathcal{W} \supseteq \{\epsilon, \textsc{bos}\}$, 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 $w \in \textsc{bos}^?\Sigma^*\textsc{eos}^?$ 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 λ 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 $\textsc{bos} \in \mathcal{W}$ only so that we can extend it rightward into $\textsc{bos}\ \texttt{a}$, $\textsc{bos}\ \texttt{ab}$, etc. When constructing a WFSA, Appendix B.3 will remove $\textsc{bos}$ 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 $e(w) \overset{\text{def}}{=} \mathbb{E}_{v\sim p}[f_w(v)]$, the expected count of the substring w.

For simplicity, let us assume that p is given by an ϵ-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 $s \overset{c/k}{\rightsquigarrow} s'$). The base case m_ϵ is given by $m_\epsilon[s] = \alpha(s)$. The base case m_{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 $\textsc{Encode}(\boldsymbol{\theta})$ 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 using only factors of degree 2. This follows a standard transformation that reformulates any problem in constraint satisfaction programming (CSP) using 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 ith 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`

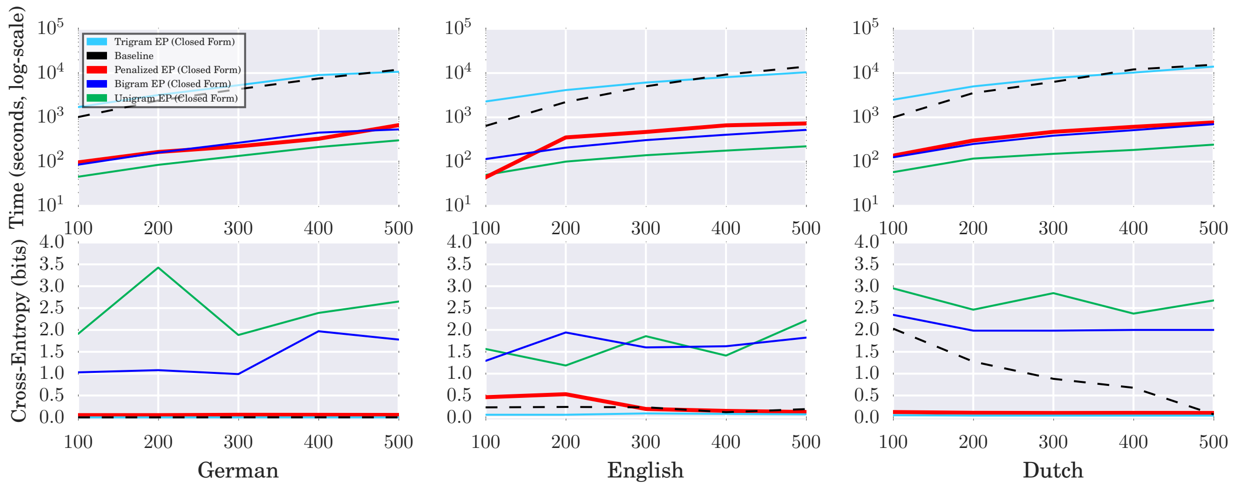


Figure 6: A version of Figure 3 that uses the closed-form methods to estimate $\boldsymbol{\theta}$, rather than the gradient-based methods. The same general pattern is observed.

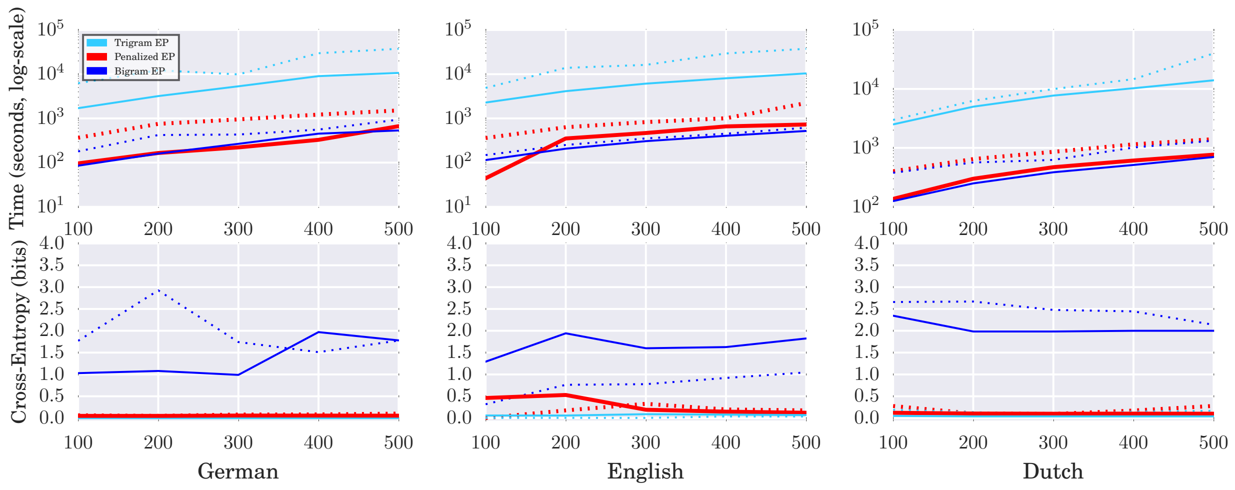


Figure 7: A comparison of the key curves from Figures 3 and 6. The dotted lines show the gradient-based methods, while the solid lines show the closed-form methods. The closed-form method is generally a bit faster, and has comparable accuracy.