Structured Penalties for Log-linear Language Models

Anil Nelakanti, Cédric Archambeau, Julien Mairal, Francis Bach, Guillaume Bouchard

Structured Penalties for Log-linear Language Models

Anil Nelakanti,[*‡] Cédric Archambeau,[*] Julien Mairal,[†] Francis Bach,[‡] Guillaume Bouchard[*]
[*]Xerox Research Centre Europe, Grenoble, France
[†]INRIA-LEAR Project-Team, Grenoble, France
[‡]INRIA-SIERRA Project-Team, Paris, France
firstname.lastname@xrce.xerox.com firstname.lastname@inria.fr

Abstract

Language models can be formalized as log-linear regression models where the input features represent previously observed contexts up to a certain length m. The complexity of existing algorithms to learn the parameters by maximum likelihood scale linearly in nd, where n is the length of the training corpus and d is the number of observed features. We present a model that grows logarithmically in d, making it possible to efficiently leverage longer contexts. We account for the sequential structure of natural language using tree-structured penalized objectives to avoid overfitting and achieve better generalization.

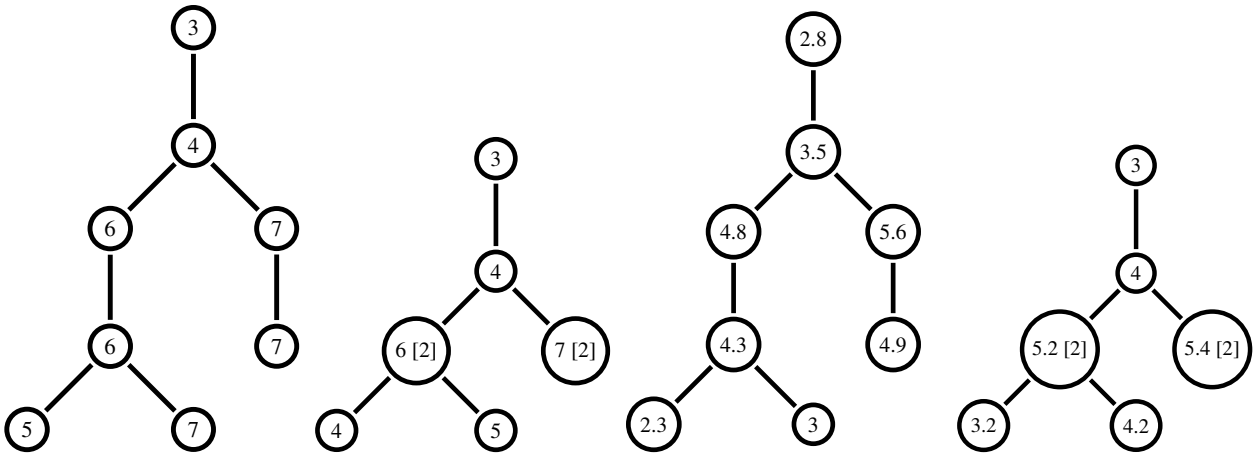
1 Introduction

Language models are crucial parts of advanced natural language processing pipelines, such as speech recognition (Burget et al., 2007), machine translation (Chang and Collins, 2011), or information retrieval (Vargas et al., 2012). When a sequence of symbols is observed, a language model predicts the probability of occurrence of the next symbol in the sequence. Models based on so-called back-off smoothing have shown good predictive power (Goodman, 2001). In particular, Kneser-Ney (KN) and its variants (Kneser and Ney, 1995) are still achieving state-of-the-art results for more than a decade after they were originally proposed. Smoothing methods are in fact clever heuristics that require tuning parameters in an ad-hoc fashion. Hence, more principled ways of learning language models have been proposed based on maximum entropy (Chen and Rosenfeld, 2000) or conditional random fields (Roark et al., 2004), or by adopting a Bayesian approach (Wood et al., 2009).

In this paper, we focus on penalized maximum likelihood estimation in log-linear models. In contrast to language models based on *unstructured* norms such as ℓ_2 (quadratic penalties) or ℓ_1 (absolute discounting), we use *tree-structured* norms (Zhao et al., 2009; Jenatton et al., 2011). Structured penalties have been successfully applied to various NLP tasks, including chunking and named entity recognition (Martins et al., 2011), but not language modelling. Such penalties are particularly well-suited to this problem as they mimic the nested nature of word contexts. However, existing optimizing techniques are not scalable for large contexts m.

In this work, we show that structured tree norms provide an efficient framework for language modelling. For a special case of these tree norms, we obtain an memory-efficient learning algorithm for log-linear language models. Furthermore, we aslo give the first efficient learning algorithm for structured ℓ_∞ tree norms with a complexity nearly linear in the number of training samples. This leads to a memory-efficient *and* time-efficient learning algorithm for generalized linear language models.

The paper is organized as follows. The model and other preliminary material is introduced in Section 2. In Section 3, we review unstructured penalties that were proposed earlier. Next, we propose structured penalties and compare their memory and time requirements. We summarize the characteristics of the proposed algorithms in Section 5 and experimentally validate our findings in Section 6.



(a) Trie-structured vector. $w = [\,3\,4\,6\,6\,4\,5\,7\,7\,]^\top$.

(b) Tree-structured vector. $w = [\,3\,4\,6\,6\,4\,5\,7\,7\,]^\top$.

(c) ℓ_2^T-proximal $\Pi_{\ell_2^T}(w, 0.8) =$ $[\,2.8\,3.5\,4.8\,4.3\,2.3\,3\,5.6\,4.9\,]^\top$.

(d) ℓ_∞^T-proximal $\Pi_{\ell_\infty^T}(w, 0.8) =$ $[\,3\,4\,5.2\,5.2\,3.2\,4.2\,5.4\,5.4\,]^\top$.

Figure 1: Example of uncollapsed (trie) and corresponding collapsed (tree) structured vectors and proximal operators applied to them. Weight values are written inside the node. Subfigure (a) shows the complete trie S and Subfigure (b) shows the corresponding collapsed tree T. The number in the brackets shows the number of nodes collapsed. Subfigure (c) shows vector after proximal projection for ℓ_2^T-norm (which cannot be collapsed), and Subfigure (d) that of ℓ_∞^T-norm proximal projection which can be collapsed.

2 Log-linear language models

Multinomial logistic regression and Poisson regression are examples of log-linear models (McCullagh and Nelder, 1989), where the likelihood belongs to an exponential family and the predictor is linear. The application of log-linear models to language modelling was proposed more than a decade ago (Della Pietra et al., 1997) and it was shown to be competitive with state-of-the-art language modelling such as Knesser-Ney smoothing (Chen and Rosenfeld, 2000).

2.1 Model definition

Let V be a set of words or more generally a set of symbols, which we call vocabulary. Further, let xy be a sequence of $n+1$ symbols of V, where $x \in V^n$ and $y \in V$. We model the probability that symbol y succeeds x as

$$P(y = v|x) = \frac{e^{w_v^\top \phi_m(x)}}{\sum_{u \in V} e^{w_u^\top \phi_m(x)}}, \qquad (1)$$

where $W = \{w_v\}_{v \in V}$ is the set of parameters, and $\phi_m(x)$ is the vector of features extracted from x, the sequence preceding y. We will describe the features shortly.

Let $x_{1:i}$ denote the subsequence of x starting at the first position up to the i^{th} position and y_i the next symbol in the sequence. Parameters are estimated by minimizing the penalized log-loss:

$$W^* \in \operatorname*{argmin}_{W \in \mathcal{K}} f(W) + \lambda \Omega(W), \qquad (2)$$

where $f(W) := -\sum_{i=1}^n \ln p(y_i|x_{1:i}; W)$ and $\mathcal{K}$ is a convex set representing the constraints applied on the parameters. Overfitting is avoided by adjusting the regularization parameter λ, e.g., by cross-validation.

2.2 Suffix tree encoding

Suffix trees provide an efficient way to store and manipulate discrete sequences and can be constructed in linear time when the vocabulary is fixed (Giegerich and Kurtz, 1997). Recent examples include language models based on a variable-length Markovian assumption (Kennington et al., 2012) and the sequence memoizer (Wood et al., 2011). The suffix tree data structure encodes all the unique suffixes observed in a sequence up to a maximum given length. It exploits the fact that the set of observed contexts is a small subset of all possible contexts. When a series of suffixes of increasing lengths are

Algorithm 1 $W^* := \operatorname{argmin} \{f(X, Y; W) + \lambda\Omega(W)\}$ Stochastic optimization algorithm (Hu et al., 2009)

Input: λ regularization parameter , L Lipschitz constant of ∇f , μ coefficient of strong-convexity of $f + \lambda\Omega$, X design matrix, Y label set
Initialize: $\overline{W} = Z = 0, \tau = \delta = 1, \rho = L + \mu$
repeat until maximum iterations
  #estimate point for gradient update
  $\overline{W} = (1 - \tau)W + \tau Z$
  #use mini-batch $\{X_\vartheta, Y_\vartheta\}$ for update
  W = ParamUpdate$(X_\vartheta, Y_\vartheta, \overline{W}, \lambda, \rho)$
  #weighted combination of estimates
  $Z = \frac{1}{\rho\tau+\mu}\big((1 - \mu)Z + (\mu - \rho)\overline{W} + \rho W\big)$
  #update constants
  $\rho = L + \mu/\delta, \tau = \frac{\sqrt{4\delta+\delta^2}-\delta}{2}, \delta = (1 - \tau)\delta$

Procedure: W := ParamUpdate$(X_\vartheta, Y_\vartheta, \overline{W}, \lambda, \rho)$
  $W' = \overline{W} - \frac{1}{\rho}\nabla f(X_\vartheta, Y_\vartheta, \overline{W})$ #gradient step
  $W = [W]_+$ #projection to non-negative orthant
  $W = \Pi_\Omega(w, \kappa)$ #proximal step

always observed in the same context, the successive suffixes are collapsed into a single node. The uncollapsed version of the suffix tree T is called a suffix *trie*, which we denote S. A suffix trie also has a tree structure, but it potentially has much larger number of nodes. An example of a suffix trie S and the associated suffix tree T are shown in Figures 1(a) and 1(b) respectively. We use $|S|$ to denote the number of nodes in the trie S and $|T|$ for the number of nodes in the tree T.

Suffix tree encoding is particularly helpful in applications where the resulting hierarchical structures are thin and tall with numerous non-branching paths. In the case of text, it has been observed that the number of nodes in the tree grows slower than that of the trie with the length of the sequence (Wood et al., 2011; Kennington et al., 2012). This is a significant gain in the memory requirements and, as we will show in Section 4, can also lead to important computational gains when this structure is exploited.

The feature vector $\phi_m(x)$ encodes suffixes (or contexts) of increasing length up to a maximum length m. Hence, the model defined in (1) is similar to m-gram language models. Naively, the feature vector $\phi_m(x)$ corresponds to one path of length m starting at the root of the suffix trie S. The entries in W correspond to weights for each suffix. We thus have a trie structure S on W (see Figure 1(a)) constraining the number of free parameters. In other words, there is one weight parameter per node in the trie S and the matrix of parameters W is of size $|S|$.

In this work, however, we consider models where the number of parameters is equal to the size of the suffix tree T, which has much fewer nodes than S. This is achieved by ensuring that all parameters corresponding to suffixes at a node share the same parameter value (see Figure 1(b)). These parameters correspond to paths in the suffix trie that do not branch *i.e.* sequence of words that always appear together in the same order.

2.3 Proximal gradient algorithm

The objective function (2) involves a smooth convex loss f and a possibly non-smooth penalty Ω. Subgradient descent methods for non-smooth Ω could be used, but they are unfortunately very slow to converge. Instead, we choose proximal methods (Nesterov, 2007), which have fast convergence rates and can deal with a large number of penalties Ω, see (Bach et al., 2012).

Proximal methods iteratively update the current estimate by making a generalized gradient update at each iteration. Formally, they are based on a linearization of the smooth function f around a parameter estimate $\overline{W}$, adding a quadratic penalty term to keep the updated estimate in the neighborhood of $\overline{W}$. At iteration t, the update of the parameter W is given by

$$W^{t+1} = \operatorname*{argmin}_{W \in \mathcal{K}} \Big\{ f(\overline{W}) + (W - \overline{W})^\top \nabla f(\overline{W}) + \Omega(W) + \frac{L}{2}\|W - \overline{W}\|_2^2 \Big\}, \quad (3)$$

where $L > 0$ is an upper-bound on the Lipschitz constant of the gradient ∇f. The matrix $\overline{W}$ could either be the current estimate W^t or its weighted combination with the previous estimate for accelerated convergence depending on the specific algorithm used (Beck and Teboulle, 2009). Equation (3) can be rewritten to be solved in two independent steps: a gradient update from the smooth part followed by a projection depending only on the non-smooth penalty:

$$W' = \overline{W} - \frac{1}{L}\nabla f(\overline{W}), \quad (4)$$

$$W^{t+1} = \operatorname*{argmin}_{W \in \mathcal{K}} \frac{1}{2} \left\| W - W' \right\|_2^2 + \frac{\lambda \Omega(W)}{L}. \quad (5)$$

Update (5) is called the proximal operator of W' with parameter $\frac{\lambda}{L}$ that we denote $\Pi_\Omega \left(W', \frac{\lambda}{L} \right)$. Efficiently computing the proximal step is crucial to maintain the fast convergence rate of these methods.

2.4 Stochastic proximal gradient algorithm

In language modelling applications, the number of training samples n is typically in the range of 10^5 or larger. Stochastic version of the proximal methods (Hu et al., 2009) have been known to be well adapted when n is large. At every update, the stochastic algorithm estimates the gradient on a mini-batch, that is, a subset of the samples. The size of the mini-batches controls the trade-off between the variance in the estimate of gradient and the time required for compute it. In our experiments we use mini-batches of size 400. The training algorithm is summarized in Algorithm 1. The acceleration is obtained by making the gradient update at a specific weighted combination of the current and the previous estimates of the parameters. The weighting is shown in step 6 of the Algorithm 1.

2.5 Positivity constraints

Without constraining the parameters, the memory required by a model scales linearly with the vocabulary size $|V|$. Any symbol in V observed in a given context is a positive example, while any symbols in V that does not appear in this context is a negative example. When adopting a log-linear language model, the negative examples are associated with a small negative gradient step in (4), so that the solution is not sparse accross multiple categories in general. By constraining the parameters to be positive (i.e., the set of feasible solutions $\mathcal{K}$ is the positive orthant), the projection step 2 in Algorithm 1 can be done with the same complexity, while maintaining sparse parameters accross multiple categories. More precisely, the weights for the category k associated to a given context x, is always zeros if the category k never occured after context x. A significant gain in memory (nearly $|V|$-fold for large context lengths) was obtained without loss of accuracy in our experiments.

3 Unstructured penalties

Standard choices for the penalty function $\Omega(W)$ include the ℓ_1-norm and the squared ℓ_2-norm. The former typically leads to a solution that is sparse and easily interpretable, while the latter leads to a non-sparse, generally more stable one. In particular, the squared ℓ_2 and ℓ_1 penalties were used in the context of log-linear language models (Chen and Rosenfeld, 2000; Goodman, 2004), reporting performances competitive with bi-gram and tri-gram interpolated Kneser-Ney smoothing.

3.1 Proximal step on the suffix trie

For squared ℓ_2 penalties, the proximal step $\Pi_{\ell_2^2}(w^t, \frac{\kappa}{2})$ is the element-wise rescaling operation:

$$w_i^{(t+1)} \leftarrow w_i^{(t)} (1 + \kappa)^{-1} \quad (6)$$

For ℓ_1 penalties, the proximal step $\Pi_{\ell_1}(w^t, \kappa)]$ is the soft-thresholding operator:

$$w_i^{(t+1)} \leftarrow \max(0, w_i^{(t)} - \kappa). \quad (7)$$

These projections have linear complexity in the number of features.

3.2 Proximal step on the suffix tree

When feature values are identical, the corresponding proximal (and gradient) steps are identical. This can be seen from the proximal steps (7) and (6), which apply to single weight entries. This property can be used to group together parameters for which the feature values are equal. Hence, we can collapse successive nodes that always have the same values in a suffix trie (as in Figure 1(b)), that is to say we can directly work on the suffix tree. This leads to a proximal step with complexity that scales linearly with the number of symbols seen in the corpus (Ukkonen, 1995) and logarithmically with context length.

4 Structured penalties

The ℓ_1 and squared ℓ_2 penalties do not account for the sequential dependencies in the data, treating suffixes of different lengths equally. This is inappropriate considering that longer suffixes are typically observed less frequently than shorter ones. Moreover, the fact that suffixes might be nested is disregarded. Hence, we propose to use the tree-structured

Algorithm 2 $w := \Pi_{\ell_2^T}(w, \kappa)$ Proximal projection step for ℓ_2^T on grouping $\mathcal{G}$.

Input: T suffix tree, w trie-structured vector, κ threshold
Initialize: $\{\gamma_i\} = 0$, $\{\eta_i\} = 1$
η = UpwardPass(η, γ, κ, w)
w = DownwardPass(η, w)

Procedure: η := UpwardPass(η, γ, κ, w)
for $x \in$ DepthFirstSuffixTraversal(T, *PostOrder*)
  $\gamma_x = w_x^2 + \sum_{h \in children(x)} \gamma_h$
  $\eta_x = [1 - \kappa/\sqrt{\gamma_x}]_+$
  $\gamma_x = \eta_x^2 \gamma_x$

Procedure: w := DownwardPass(η, w)
for $x \in$ DepthFirstSuffixTraversal(T, *PreOrder*)
  $w_x = \eta_x w_x$
  for $h \in children(x)$
    $\eta_h = \eta_x \eta_h$

a DepthFirstSuffixTraversal(T,*Order*) returns observed suffixes from the suffix tree T by depth-first traversal in the order prescribed by *Order*.
b w_x is the weights corresponding to the suffix x from the weight vector w and $children(x)$ returns all the immediate children to suffix x in the tree.

Algorithm 3 $w := \Pi_{\ell_\infty^T}(w, \kappa)$ Proximal projection step for ℓ_∞^T on grouping $\mathcal{G}$.

Input: T suffix tree, w=[v c] tree-structured vector v with corresponding number of suffixes collapsed at each node in c, κ threshold
for $x \in$ DepthFirstNodeTraversal(T, *PostOrder*)
  $g(v, x) := \pi_{\ell_\infty^T}(\, g(v, x), c_x \kappa\,)$

Procedure: $q := \pi_{\ell_\infty}(q, \kappa)$
Input: $q = [v\ c],\ q_i = [v_i\ c_i],\ i = 1, \cdots, |q|$
Initialize: $U = \{\}, L = \{\}, I = \{1, \cdots, |q|\}$
while $I \neq \varnothing$
  pick random $\rho \in I$ #choose pivot
  $U = \{j | v_j \geq v_\rho\}$ #larger than v_ρ
  $L = \{j | v_j < v_\rho\}$ #smaller than v_ρ
  $\delta S = \sum_{i \in U} v_i \cdot c_i,\ \delta C = \sum_{i \in U} c_i$
  if $(S + \delta S) - (C + \delta C)\rho < \kappa$
    $S := (S + \delta S),\ C := (C + \delta C),\ I := L$
  else $I := U \backslash \{\rho\}$
$r = \frac{S - \kappa}{C}$, $v_i := v_i - \max(0, v_i - r)$ #take residuals

a DepthFirstNodeTraversal(T,*Order*) returns nodes x from the suffix tree T by depth-first traversal in the order prescribed by *Order*.

norms (Zhao et al., 2009; Jenatton et al., 2011), which are based on the suffix trie or tree, where subtrees correspond to contexts of increasing lengths. As will be shown in the experiments, this prevents the model to overfit unlike the ℓ_1- or squared ℓ_2-norm.

4.1 Definition of tree-structured ℓ_p^T norms

Definition 1. *Let x be a training sequence. Group $g(w, j)$ is the subvector of w associated with the subtree rooted at the node j of the suffix trie $S(x)$.*

Definition 2. *Let $\mathcal{G}$ denote the ordered set of nodes of the tree $T(x)$ such that for $r < s$, $g(w, r) \cap g(w, s) = \emptyset$ or $g(w, r) \subset g(w, s)$. The tree-structured ℓ_p-norm is defined as follows:*

$$\ell_p^T(w) = \sum_{j \in \mathcal{G}} \|g(w, j)\|_p . \tag{8}$$

We specifically consider the cases $p = 2, \infty$ for which efficient optimization algorithms are available. The ℓ_p^T-norms can be viewed as a group sparsity-inducing norms, where the groups are organized in a tree. This means that when the weight associated with a parent in the tree is driven to zero, the weights associated to all its descendants should also be driven to zero.

For structured ℓ_p^T-norm, the proximal step amounts to residuals of recursive projections on the ℓ_q-ball in the order defined by $\mathcal{G}$ (Jenatton et al., 2011), where ℓ_q-norm is the dual norm of ℓ_p-norm[1]. In the case ℓ_2^T-norm this comes to a series of projections on the ℓ_2-ball. For ℓ_∞^T-norm it is instead projections on the ℓ_1-ball. The order of projections defined by $\mathcal{G}$ is generated by an upward pass of the suffix trie. At each node through the upward pass, the subtree below is projected on the dual norm ball of size κ, the parameter of proximal step. We detail the projections on the norm ball below.

4.2 Projections on ℓ_q-ball for $q = 1, 2$

Each of the above projections on the dual norm ball takes one of the following forms depending on the choice of the norm. Projection of vector w on the ℓ_2-ball is equivalent to thresholding the magnitude of w by κ units while retaining its direction:

$$w \leftarrow [||w||_2 - \kappa]_+ \frac{w}{||w||_2}. \tag{9}$$

This can be performed in time linear in size of w, $O(|w|)$. Projection of a non-negative vector w on the ℓ_1-ball is more involved and requires thresholding

[1] ℓ_p-norm and ℓ_q-norm are dual to each other if $\frac{1}{p} + \frac{1}{q} = 1$. ℓ_2-norm is self-dual while the dual of ℓ_∞-norm is the ℓ_1-norm.

by a value such that the entries in the resulting vector add up to κ, otherwise w remains the same:

$$w \leftarrow [w - \tau]_+ \text{ s.t. } ||w||_1 = \kappa \text{ or } \tau = 0. \quad (10)$$

$\tau = 0$ is the case where w lies inside the ℓ_1-ball of size κ with $||w||_1 < \kappa$, leaving w intact. In the other case, the threshold τ is to be computed such that after thresholding, the resulting vector has an ℓ_1-norm of κ. The simplest way to achieve this is to sort by descending order the entries $\overline{w} = \text{sort}(w)$ and pick the k largest values such that the $(k+1)^{th}$ largest entry is smaller than τ:

$$\sum_{i=1}^{k} \overline{w}_i - \tau = \kappa \text{ and } \tau > \overline{w}_{k+1}. \quad (11)$$

We refer to $\overline{w}_k$ as the pivot and are only interested in entries larger than the pivot. Given a sorted vector, it requires looking up to exactly k entries, however, sorting itself take $O(|w| \log |w|)$.

4.3 Proximal step

Naively employing the projection on the ℓ_2-ball described above leads to an $O(d^2)$ algorithm for ℓ_2^T proximal step. This could be improved to a linear algorithm by aggregating all necessary scaling factors while making an upward pass of the trie S and applying them in a single downward pass as described in (Jenatton et al., 2011). In Algorithm 2, we detail this procedure for trie-structured vectors.

The complexity of ℓ_∞^T-norm proximal step depends directly on that of the pivot finding algorithm used within its ℓ_1-projection method. Naively sorting vectors to find the pivot leads to an $O(d^2 \log d)$ algorithm. Pivot finding can be improved by randomly choosing candidates for the pivot and the best known algorithm due to (Bruckner, 1984) has amortized linear time complexity in the size of the vector. This leaves us with $O(d^2)$ complexity for ℓ_∞^T-norm proximal step. (Duchi et al., 2008) proposes a method that scales linearly with the number of non-zero entries in the gradient update (s) but logarithmically in d. But recursive calls to ℓ_1-projection over subtrees will fail the sparsity assumption (with $s \approx d$) making proximal step quadratic. Procedure for $\Pi_{\ell_\infty^T}$ on trie-structured vectors using randomized pivoting method is described in Algorithm 3.

We next explain how the number of ℓ_1-projections can be reduced by switching to the tree T instead of trie S which is possible due to the good properties of ℓ_∞^T-norm. Then we present a pivot finding method that is logarithmic in the feature size for our application.

4.4 ℓ_∞^T-norm with suffix trees

We consider the case where all parameters are initialized with the same value for the optimization procedure, typically with zeros. The condition that the parameters at any given node continue to share the same value requires that both the gradient update (4) and proximal step (5) have this property. We modify the tree structure to ensure that after gradient updates parameters at a given node continue to share a single value. Nodes that do not share a value after gradient update are split into multiple nodes where each node has a single value. We formally define this property as follows:

Definition 3. *A **constant value non-branching path** is a set of nodes $P \in \mathcal{P}(T, w)$ of a tree structure T w.r.t. vector w if P has $|P|$ nodes with $|P| - 1$ edges between them and each node has at most one child and all nodes $i, j \in P$ have the same value in vector w as $w_i = w_j$.*

The nodes of Figure 1(b) correspond to constant value non-branching paths when the values for all parameters at each of the nodes are the same. Next we show that this tree structure is retained after proximal steps of ℓ_∞^T-norm.

Proposition 1. *Constant value non-branching paths $\mathcal{P}(T, w)$ of T structured vector w are preserved under the proximal projection step $\Pi_{\ell_\infty^T}(w, \kappa)$.*

Figure 1(d) illustrates this idea showing ℓ_∞^T projection applied on the collapsed tree. This makes it memory efficient but the time required for the proximal step remains the same since we must project each subtree of S on the ℓ_1-ball. The sequence of projections at nodes of S in a non-branching path can be rewritten into a single projection step using the following technique bringing the number of projections from $|S|$ to $|T|$.

Proposition 2. *Successive projection steps for subtrees with root in a constant value non-branching path $P = \{g_1, \cdots, g_{|P|}\} \in \mathcal{P}(T, w)$ for $\Pi_{\ell_\infty^T}(w, \kappa)$*

is $\pi_{g_{|P|}} \circ \cdots \circ \pi_{g_1}(w, \kappa)$ applied in bottom-up order defined by $\mathcal{G}$. The composition of projections can be rewritten into a single projection step with κ scaled by the number of projections $|P|$ as,

$$\pi_{g_{|P|}}(w, \kappa|P|) \equiv \pi_{g_{|P|}} \circ \cdots \circ \pi_{g_1}(w, \kappa).$$

The above propositions show that ℓ_∞^T-norm can be used with the suffix tree with fewer projection steps. We now propose a method to further improve each of these projection steps.

4.5 Fast proximal step for ℓ_∞^T-norm

Let k be the cardinality of the set of values larger than the pivot in a vector to compute the threshold for ℓ_1-projection as referred in (11). This value varies from one application to another, but for language applications, our experiments on 100K english words (APNews dataset) showed that k is generally small: its value is on average 2.5, and its maximum is around 10 and 20, depending on the regularization level. We propose using a max-heap data structure (Cormen et al., 1990) to fetch the k-largest values necessary to compute the threshold. Given the heap of the entries the cost of finding the pivot is $O(k \log(d))$ if the pivot is the k^{th} largest entry and there are d features. This operation is performed d times for ℓ_∞^T-norm as we traverse the tree bottom-up. The heap itself is built on the fly during this upward pass. At each subtree, the heap is built by merging those of their children in constant time by using Fibonacci heaps. This leaves us with a $O(dk \log(d))$ complexity for the proximal step. This procedure is detailed in Algorithm 4.

Algorithm 4 $w := \Pi_{\ell_\infty^T}(w, \kappa)$ Proximal projection step for ℓ_∞^T on grouping $\mathcal{G}$ using heap data structure.

Input: T suffix tree, w=[v c] tree-structured vector v with corresponding number of suffixes collapsed at each node in c, κ threshold
Initialize $\mathcal{H}$ = {} # empty set of heaps
for $x \in$ DepthFirstNodeTraversal(T, *PostOrder*)
  $g(v, x) := \pi_{\ell_\infty^T}(w, x, c_x\kappa, \mathcal{H})$

Procedure: $q := \pi_{\ell_\infty}(w, x, \kappa, \mathcal{H})$
$\mathcal{H}_x = \text{NewHeap}(v_x, c_x, v_x)$
for $j \in$ children(x) # merge with child heaps
  $\tau_x = \tau_x + \tau_j$ # update ℓ_1-norm
  $\mathcal{H}_x = \text{Merge}(\mathcal{H}_x, \mathcal{H}_j), \mathcal{H} = \mathcal{H} \backslash \mathcal{H}_j$
$\mathcal{H} = \mathcal{H} \cup H_x, S = 0, C = 0, J = \{\}$
if $\mathcal{H}_x(\tau) < \kappa$, **set** $\mathcal{H}_x = 0$ **return**
for $j \in$ OrderedIterator($\mathcal{H}_x$) # get max values
  if $v_j > \frac{S + (v_j c_j) - \kappa}{C + c_j}$
    $S = S + (v_j \cdot c_j), C = C + c_j, J = J \cup \{j\}$
  else break
$r = \frac{S-\kappa}{C}, \delta = 0$ # compute threshold
for $j \in J$ # apply threshold
  $\nu = \min(v_j, r), \delta = \delta + (v_j - \nu)$
  $\mathcal{H}_j(v) = \nu$
$\mathcal{H}_x(\tau) = \mathcal{H}_j(\tau) - \delta$ # update ℓ_1-norm

a. Heap structure on vector w holds three values (v, c, τ) at each node. v, c being value and its count, τ is the ℓ_1-norm of the sub-vector below. Tuples are ordered by decreasing value of v and $\mathcal{H}_j$ refers to heap with values in sub-tree rooted at j. Merge operation merges the heaps passed. OrderedIterator returns values from the heap in decreasing order of v.

5 Summary of the algorithms

Table 1 summarizes the characteristics of the algorithms associated to the different penalties:

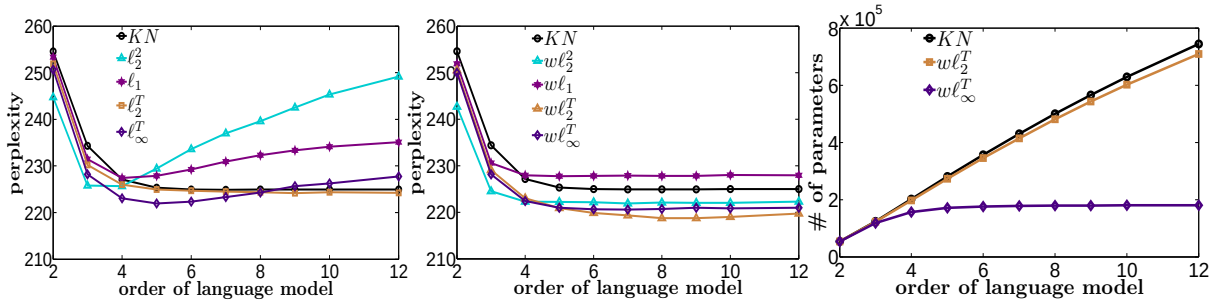
1. The unstructured norms ℓ_p do not take into account the varying sparsity level with context length. For p=1, this leads to a sparse solution and for p=2, we obtain the classical quadratic penalty. The suffix tree representation leads to an efficient memory usage. Furthermore, to make the training algorithm time efficient, the parameters corresponding to contexts which always occur in the same larger context are grouped. We will illustrate in the experiments that these penalties do not lead to good predictive performances.

2. The ℓ_2^T-norm nicely groups features by subtrees which concurs with the sequential structure of sequences. This leads to a powerful algorithm in terms of generalization. But it can only be applied on the uncollapsed tree since there is no closure property of the *constant value non-branching path* for its proximal step making it less amenable for larger tree depths.

3. The ℓ_∞^T-norm groups features like the ℓ_2^T-norm while additionally encouraging numerous feature groups to share a single value, leading to a substantial reduction in memory usage. The generalization properties of this algorithm is as good as the generalization obtained with the ℓ_2^T penalty, if not better. However, it has the *constant value non-branching path* property, which

Penalty			good generalization	memory efficient	time efficient
unstructured ℓ_1 and ℓ_2^2			no	yes $_{O(\|T\|)}$	yes $_{O(\|T\|)}$
struct.	ℓ_2^T		yes	no $_{O(\|S\|)}$	no $_{O(\|S\|)}$
	ℓ_∞^T	rand. pivot	yes	yes $_{O(\|T\|)}$	no $_{O(\|T\|^2)}$
	ℓ_∞^T	heap	yes	yes $_{O(\|T\| \log \|T\|)}$	yes $_{O(\|T\| \log \|T\|)}$

Table 1: Properties of the algorithms proposed in this paper. Generalization properties are as compared by their performance with increasing context length. Memory efficiency is measured by the number of free parameters of W in the optimization. Note that the suffix tree is much smaller than the trie (uncollapsed tree): $|T| << |S|$. Time complexities reported are that of one proximal projection step.



(a) Unweighted penalties. (b) Weighted penalties. (c) Model complexity for structured penalties.

Figure 2: (a) compares average perplexity (lower is better) of different methods from 2-gram through 12-gram on four different 100K-20K train-test splits. (b) plot compares the same with appropriate feature weighting. (c) compares model complexity for weighted structured penalties $w\ell_2^T$ and $w\ell_\infty^T$ measure by then number of parameters.

means that the proximal step can be applied directly to the suffix tree. There is thus also a significant gain of performances.

6 Experiments

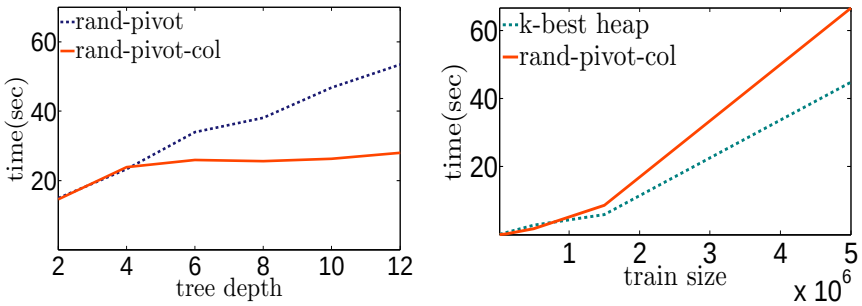
In this section, we demonstrate empirically the properties of the algorithms summarized in Table 1. We consider four distinct subsets of the Associated Press News (AP-news) text corpus with train-test sizes of 100K-20K for our experiments. The corpus was preprocessed as described in (Bengio et al., 2003) by replacing proper nouns, numbers and rare words with special symbols "⟨proper_noun⟩", "#n" and "⟨unknown⟩" respectively. Punctuation marks are retained which are treated like other normal words. Vocabulary size for each of the training subsets was around 8,500 words. The model was reset at the start of each sentence, meaning that a word in any given sentence does not depend on any word in the previous sentence. The regularization parameter λ is chosen for each model by cross-validation on a smaller subset of data. Models are fitted to training sequence of 30K words for different values of λ and validated against a sequence of 10K words to choose λ.

We quantitatively evaluate the proposed model using perplexity, which is computed as follows:

$$P(\{x_i, y_i\}, W) = 10^{\left\{\frac{-1}{n_V}\sum_{i=1}^{n}\mathbb{I}(y_i \in V)\log p(y_i|x_{1:i};W)\right\}},$$

where $n_V = \sum_i \mathbb{I}(y_i \in V)$. Performance is measured for varying depth of the suffix trie with different penalties. Interpolated Kneser-Ney results were computed using the openly available SRILM toolkit (Stolcke, 2002).

Figure 2(a) shows perplexity values averaged over four data subsets as a function of the language model order. It can be observed that performance of unstructured ℓ_1 and squared ℓ_2 penalties improve until a relatively low order and then degrade, while ℓ_2^T penalty does not show such degradation, indicating



(a) Iteration time of random-pivoting on the collapsed and uncollapsed trees.

(b) Iteration time of random-pivoting and k-best heap on the collapsed tree.

Figure 3: Comparison of different methods for performing ℓ_∞^T proximal projection. The `rand-pivot` is the random pivoting method of (Bruckner, 1984) and `rand-pivot-col` is the same applied with the nodes collapsed. The `k-best heap` is the method described in Algorithm 4.

that taking the tree-structure into account is beneficial. Moreover, the log-linear language model with ℓ_2^T penalty performs similar to interpolated Kneser-Ney. The ℓ_∞^T-norm outperforms all other models at order 5, but taking the structure into account does not prevent a degradation of the performance at higher orders, unlike ℓ_2^T. This means that a single regularization for all model orders is still inappropriate.

To investigate this further, we adjust the penalties by choosing an exponential decrease of weights varying as α^m for a feature at depth m in the suffix tree. Parameter α was tuned on a smaller validation set. The best performing values for these weighted models $w\ell_2^2$, $w\ell_1$, $w\ell_2^T$ and $w\ell_\infty^T$ are 0.5, 0.7, 1.1 and 0.85 respectively. The weighting scheme further appropriates the regularization at various levels to suit the problem's structure. Perplexity plots for weighted models are shown in Figure 2(b). While $w\ell_1$ improves at larger depths, it fails to compare to others showing that the problem does not admit sparse solutions. Weighted ℓ_2^2 improves considerably and performs comparably to the unweighted tree-structured norms. However, the introduction of weighted features prevents us from using the suffix tree representation, making these models inefficient in terms of memory. Weighted ℓ_∞^T is corrected for overfitting at larger depths and $w\ell_2^T$ gains more than others. Optimal values for α are fractional for all norms except $w\ell_2^T$-norm showing that the unweighted model ℓ_2^T-norm was over-penalizing features at larger depths, while that of others were under-penalizing them. Interestingly, perplexity improves up to about 9-grams with $w\ell_2^T$ penalty for the data set we considered, indicating that there is more to gain from longer dependencies in natural language sentences than what is currently believed.

Figure 2(c) compares model complexity measured by the number of parameters for weighted models using structured penalties. The ℓ_2^T penalty is applied on trie-structured vectors, which grows roughly at a linear rate with increasing model order. This is similar to Kneser-Ney. However, the number of parameters for the $w\ell_\infty^T$ penalty grows logarithmically with the model order. This is due to the fact that it operates on the suffix tree-structured vectors instead of the suffix trie-structured vectors. These results are valid for, both, weighted and unweighted penalties.

Next, we compare the average time taken per iteration for different implementations of the ℓ_∞^T proximal step. Figure 3(a) shows this time against increasing depth of the language model order for random pivoting method with and without the collapsing of parameters at different constant value non-branching paths. The trend in this plot resembles that of the number of parameters in Figure 2(c). This shows that the complexity of the full proximal step is sublinear when accounting for the suffix tree data structure. Figure 3(b) plots time per iteration random pivoting and k-best heap against the varying size of training sequence. The two algorithms are operating directly on the suffix tree. It can be observed that the heap-based method are superior with

increasing size of training data.

7 Conclusion

In this paper, we proposed several log-linear language models. We showed that with an efficient data structure and structurally appropriate convex regularization schemes, they were able to outperform standard Kneser-Ney smoothing. We also developed a proximal projection algorithm for the tree-structured ℓ_∞^T-norm suitable for large trees.

Further, we showed that these models can be trained online, that they accurately learn the m-gram weights and that they are able to better take advantage of long contexts. The time required to run the optimization is still a concern. It takes 7583 minutes on a standard desktop computer for one pass of the of the complete AP-news dataset with 13 million words which is little more than time reported for (Mnih and Hinton, 2007). The most time consuming part is computing the normalization factor for the log-loss. A hierarchical model in the flavour of (Mnih and Hinton, 2008) should lead to significant improvements to this end. Currently, the computational bottleneck is due to the normalization factor in (1) as it appears in every gradient step computation. Significant savings would be obtained by computing it as described in (Wu and Khundanpur, 2000).

Acknowledgements

The authors would like to thank anonymous reviewers for their comments. This work was partially supported by the CIFRE grant 1178/2010 from the French ANRT.

References

F. Bach, R. Jenatton, J. Mairal, and G. Obozinski. 2012. Optimization with sparsity-inducing penalties. *Foundations and Trends in Machine Learning*, pages 1–106.

A. Beck and M. Teboulle. 2009. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM Journal of Imaging Sciences*, 2(1):183–202.

Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. 2003. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155.

P. Bruckner. 1984. An $o(n)$ algorithm for quadratic knapsack problems. *Operations Research Letters*, 3:163–166.

L. Burget, P. Matejka, P. Schwarz, O. Glembek, and J.H. Cernocky. 2007. Analysis of feature extraction and channel compensation in a GMM speaker recognition system. *IEEE Transactions on Audio, Speech and Language Processing*, 15(7):1979–1986, September.

Y-W. Chang and M. Collins. 2011. Exact decoding of phrase-based translation models through lagrangian relaxation. In *Proc. Conf. Empirical Methods for Natural Language Processing*, pages 26–37.

S. F. Chen and R. Rosenfeld. 2000. A survey of smoothing techniques for maximum entropy models. *IEEE Transactions on Speech and Audio Processing*, 8(1):37–50.

T. H. Cormen, C. E. Leiserson, and R. L. Rivest. 1990. *An Introduction to Algorithms*. MIT Press.

S. Della Pietra, V. Della Pietra, and J. Lafferty. 1997. Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(4):380–393.

J. Duchi, S. Shalev-Shwartz, Y. Singer, and T. Chandra. 2008. Efficient projections onto the ℓ_1-ball for learning in high dimensions. *Proc. 25th Int. Conf. Machine Learning*.

R. Giegerich and S. Kurtz. 1997. From ukkonen to McCreight and weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*.

J. Goodman. 2001. A bit of progress in language modelling. *Computer Speech and Language*, pages 403–434, October.

J. Goodman. 2004. Exponential priors for maximum entropy models. In *Proc. North American Chapter of the Association of Computational Linguistics*.

C. Hu, J.T. Kwok, and W. Pan. 2009. Accelerated gradient methods for stochastic optimization and online learning. *Advances in Neural Information Processing Systems*.

R. Jenatton, J. Mairal, G. Obozinski, and F. Bach. 2011. Proximal methods for hierarchical sparse coding. *Journal of Machine Learning Research*, 12:2297–2334.

C. R. Kennington, M. Kay, and A. Friedrich. 2012. Sufx trees as language models. *Language Resources and Evaluation Conference*.

R. Kneser and H. Ney. 1995. Improved backing-off for m-gram language modeling. In *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing*, volume 1.

A. F. T. Martins, N. A. Smith, P. M. Q. Aguiar, and M. A. T. Figueiredo. 2011. Structured sparsity in structured prediction. In *Proc. Conf. Empirical Methods for Natural Language Processing*, pages 1500–1511.

P. McCullagh and J. Nelder. 1989. *Generalized linear models*. Chapman and Hall. 2nd edition.

A. Mnih and G. Hinton. 2007. Three new graphical models for statistical language modelling. *Proc. 24th Int. Conference on Machine Learning*.

A. Mnih and G. Hinton. 2008. A scalable hierarchical distributed language model. *Advances in Neural Information Processing Systems*.

Y. Nesterov. 2007. Gradient methods for minimizing composite objective function. *CORE Discussion Paper*.

B. Roark, M. Saraclar, M. Collins, and M. Johnson. 2004. Discriminative language modeling with conditional random fields and the perceptron algorithm. *Proc. Association for Computation Linguistics*.

A. Stolcke. 2002. Srilm- an extensible language modeling toolkit. *Proc. Int. Conf. Spoken Language Processing*, 2:901–904.

E. Ukkonen. 1995. Online construction of suffix trees. *Algorithmica*.

S. Vargas, P. Castells, and D. Vallet. 2012. Explicit relevance models in intent-oriented information retrieval diversification. In *Proc. 35th Int. ACM SIGIR Conf. Research and development in information retrieval*, SIGIR '12, pages 75–84. ACM.

F. Wood, C. Archambeau, J. Gasthaus, J. Lancelot, and Y.-W. Teh. 2009. A stochastic memoizer for sequence data. In *Proc. 26th Intl. Conf. on Machine Learning*.

F. Wood, J. Gasthaus, C. Archambeau, L. James, and Y. W. Teh. 2011. The sequence memoizer. In *Communications of the ACM*, volume 54, pages 91–98.

J. Wu and S. Khundanpur. 2000. Efficient training methods for maximum entropy language modeling. *Proc. 6th Inter. Conf. Spoken Language Technologies*, pages 114–117.

P. Zhao, G. Rocha, and B. Yu. 2009. The composite absolute penalties family for grouped and hierarchical variable selection. *The Annals of Statistics*, 37(6A):3468–3497.