Accelerating Newton Optimization for Log-Linear Models through Feature Redundancy

Arpit Mathur
IIT Bombay
arpit@cse.iitb.ac.in

Soumen Chakrabarti
IIT Bombay
soumen@cse.iitb.ac.in

***Abstract*— Log-linear models are widely used for labeling feature vectors and graphical models, typically to estimate robust conditional distributions in presence of a large number of potentially redundant features. Limited-memory quasi-Newton methods like LBFGS or BLMVM are optimization workhorses for such applications, and most of the training time is spent computing the objective and gradient for the optimizer. We propose a simple technique to speed up the training optimization by clustering features dynamically, and interleaving the standard optimizer with another, coarse-grained, faster optimizer that uses far fewer variables. Experiments with logistic regression training for text classification and conditional random field (CRF) training for information extraction show promising speed-ups between $2\times$ and $9\times$ without any systematic or significant degradation in the quality of the estimated models.**

I. Introduction

Log-linear models are in widespread use for feature vector classification [1], [2] and in graphical models [3], [4]. Given training observations x and labels y, the goal is to learn a weight vector $\beta \in \ ^d$ to fit a conditional distribution

$$\Pr(Y = y|x) = \frac{\exp(\beta' f(x,y))}{Z_\beta(x)} = \frac{\exp\left(\sum_j \beta_j f_j(x,y)\right)}{Z_\beta(x)}, \tag{1}$$

where $f(x,y) \in \ ^d$ is a feature vector and $Z(x)$ is a normalizing constant. Throughout we will assume $f_j(x,y) > 0$. Given instances $\{(x_i, y_i)\}$ we seek β to maximize $\sum_i \log \Pr(Y = y_i|x_i)$, i.e., to minimize

$$\ell_0(\beta) = -\sum_i \left(\beta' f(x_i, y_i) - \log Z_\beta(x_i)\right). \tag{2}$$

In practice, one often asserts a Gaussian prior on each β_j with zero mean and variance σ^2 (i.e., $\Pr(\beta) \propto \exp(\beta'\beta/(2\sigma^2))$, and minimizes wrt β the objective

$$\begin{aligned}\ell(\beta) &= \frac{\beta'\beta}{2\sigma^2} - \sum_i \left(\beta' f(x_i, y_i) - \log Z_\beta(x_i)\right) \\ &= \frac{1}{2\sigma^2}\sum_j \beta_j^2 - \sum_i \left(\beta' f(x_i, y_i) - \log Z_\beta(x_i)\right).\end{aligned} \tag{3}$$

The term $\beta'\beta/(2\sigma^2)$ is also called the *ridge penalty*.

Part of the popularity of conditional log-linear models arises from the flexibility of adding enormous numbers of diverse, noisy, possibly correlated and redundant features without the fear of corrupting a naively-estimated joint density between x and y, as in naive Bayesian classifiers. Logistic regression (LR, an instance of log-linear models) gives much more accurate text classifiers than naive Bayes models, and conditional random fields (CRFs, another instance) gives more accurate information extractors than hidden Markov models (HMMs).

In traditional machine learning, training is considered a one-time computational effort, after which the trained model β is applied to test cases x to compute $\arg\max_y \beta' f(x,y)$, which is computationally much cheaper. However, in deployed classification or extraction systems, training is a continual or "life-long" activity, with class labels changing, instances being added or removed, labeling actively modified, and error cases interactively analyzed [5]. Consequently, fast training is important, and much effort has been invested on accelerating the estimation of β [6], [1], [7], [2].

Leading the field, and used in the vast majority of applications, is the limited-memory quasi-Newton method called "L-BFGS" [6], [7]. To use a Newton method, one must implement routines to calculate, given a specific vector β^*, the objective $\ell(\beta^*)$ and the gradient $\nabla_\beta \ell(\beta)$ of ℓ wrt β evaluated at β^*. The optimizer starts with an initial guess $\beta^{(0)}$ and iteratively updates it to $\beta^{(1)}$, $\beta^{(2)}$, etc., until convergence, calling the user's objective and gradient method in each iteration. If the best-of-breed optimizers are used, most of the training time is spent in these user-defined methods and relatively little time is spent inside the optimizer itself.

Our contribution: We start with the most competitive optimizer L-BFGS, widely used in log-linear models, and propose a simple and practical approach to further speed it up substantially without any deep understanding of or modification to the optimizer. Our approach achieves convergence 2–9 times faster than the baseline. In fact, we achieve saturation of test-set accuracy (F1 scores typically within 2–5% of best possible, and sometimes better than the baseline) even faster than that. While other approaches are generic to the log-linear family [2], [8], the crux of our approach is to expose explicitly to the optimizer the redundancy and correlations among the βs evolving in time. We achieve this by periodically clustering and projecting β down to an optimization over a lower dimensional parameter vector γ, optimize γ for a while, then "lift" γ back to β and "patch up" the estimates, repeating this process if necessary. We have applied this idea to text classification using logistic regression (LR) and to named-entity tagging using CRFs, demonstrating substantial speedups

beyond state of the art.

Thus, our idea helps retain a key advantage of conditional models—the ability to exploit large numbers of potentially useful features—while substantially reducing the computational burden of doing so. Our proposal may also be regarded as a means to model parsimony that is quite distinct from feature selection: even as application writers introduce millions of features f_j, there may be no evidence that millions of difference weights β_j are justified by the data.

The reduction in training time is almost entirely due to simplifying the original high-dimensional optimization problem into a lower dimensional one. By removing redundancies in the optimization surface, we make it easier for the Newton optimizer to find the best hill-climbing directions. In log-linear models, we can take advantage of feature clustering in another way: by pre-aggregating high-dimensional feature vectors into lower dimensional ones, thus saving repeated access to the feature generators in the original space (which can be quite expensive in text-processing applications). We believe we can improve training performance even further by clever engineering of feature aggregation.

Paper outline: We review background material on optimization for log-linear models in Section II, together with their adaptation to LR and CRFs. We present the feature-clustering approach for LR in Section III, and its performance on text classification tasks. In Section IV we extend the approach to CRFs and present experimental results on information extraction or named-entity tagging tasks. We conclude in Section V.

II. Related work

A. Optimization review

1) Iterative scaling variants: Suppose weight vector β is to be updated to $\beta + \delta$. We would like to maximize the reduction in the objective (2). The trick is to lower-bound the reduction with a "nice" function $Q(\delta|\beta) \leq \ell(\beta) - \ell(\beta + \delta)$, maximize Q wrt δ, then apply an additive update $\beta^{(t)} \leftarrow \beta^{(t-1)} + \delta$. The tighter the bound $Q(\delta|\beta)$ the better, provided finding $\arg\max_\delta Q(\delta|\beta)$ does not become too difficult. Several bounding functions and δ optimization strategies have been devised, leading to Improved Iterative Scaling (IIS) [9] and Faster Iterative Scaling (FIS) [2]. We will not discuss these methods in detail because, in text-processing and graphical model applications, they have been supplanted by direct Newton optimization, discussed next. In experiments we describe shortly, FIS (which was already shown to be faster than IIS) was decisively beaten by a direct Newton method.

2) Direct Newton optimization: Let $\ell(\beta)$ be the (scalar) objective at β and $g(\beta) = \nabla_\beta \ell \in \ ^d$ be the gradient vector. A Newton method seeks to minimize ℓ by walking against the gradient, i.e., by setting

$$\beta^{(t+1)} \leftarrow \beta^{(t)} - \eta_t H_t^{-1} g^{(t)}, \tag{4}$$

where $\eta_t \in$ is a step size, $g^{(t)} = g(\beta^{(t)})$ is the gradient evaluated at $\beta^{(t)}$, and H_t is the Hessian matrix [10]. (In one dimension, H is the familiar second derivative $\ell''(\beta)$ and H^{-1} is just $1/\ell''(\beta)$, and $g(\beta) = \ell'(\beta)$, leading to the usual update $\beta^{(t+1)} \leftarrow \beta^{(t)} - \eta \ell'(\beta^{(t)})/\ell''(\beta^{(t)})$.)

For large d, maintaining the Hessian matrix from iteration to iteration is a computational challenge, and that is where the BFGS family of update methods [11], [6] comes into play.

3) Limited Memory BFGS (L-BFGS): L-BFGS is a limited memory version of BFGS which saves the time and space required to compute the Hessian matrix. It maintains the last m (typically 3–7 in applications) corrections $s^{(t)} = \beta^{(t)} - \beta^{(t-1)}$ and $y^{(t)} = g^{(t)} - g^{(k-1)}$, to the solution vector and the gradient respectively, and stores an initial approximation B_0 of the inverse of Hessian (identity matrix by default). It then calculates the product $B_t g(t)$ using efficient sparse matrix multiplications during each iteration. For the first m iterations, L-BFGS is exactly same as BFGS. The specifics of L-BFGS are listed in Figure 1. A complete analysis of L-BFGS has been given by Liu et al. [6].

1: Initialize β_0, m, $t = 0$, and symmetric positive definite matrix B_0, the initial approximation to the inverse of Hessian.
2: Compute

$$d_t = -B_t \cdot g^{(t)} \quad \text{and}$$
$$\beta^{(t+1)} = \beta^{(t)} + \alpha_t d_t$$

Step length α_t is found by performing line search and choosing the one which minimizes the function in the direction of descent. That is,

$$\alpha_t = \arg\min_{\alpha>0} f(\beta^{(t)} - \alpha d_k)$$

3: Let $l = \min\{t, m-1\}$. Calculate B_{t+1} by updating B_0 $(l+1)$ times using the pairs $\{y_t, s_t\}_{\{j=t-m\}}^{t}$ as follows:

$$\begin{aligned} B_{t+1} = {} & (V_t^T \dots V_{t-l}^T) B_0 (V_{t-l} \dots V_t) + \\ & \rho_{t-l} (V_t^T \dots V_{t-l+1}^T) s_{t-l} s_{t-l}^T (V_{t-l+1} \dots V_t) + \\ & \rho_{t-l+1} (V_t^T \dots V_{t-l+2}^T) s_{t-l+1} s_{t-l+1}^T (V_{t-l+2} \dots V_t) \\ & + \dots + \rho_t s_t s_t^T \end{aligned}$$

4: If not converged, set $t = t + 1$ and goto step 2.

Fig. 1. The L-BFGS algorithm.

4) Memory Requirements of L-BFGS: L-BFGS methods are particularly helpful because of their low memory requirements as compared to actual BFGS. Since only m pairs of updates $\{s_k, y_k\}$ are stored, with each of size d, the memory required to store the updates is $2md + O(m)$. Additional memory is required to store the initial approximation H_0. L-BFGS, by default starts with H_0 being the identity matrix (or some user-defined matrix), which requires and additional d memory cells. So the total memory requirement for L-BFGS is $d(2m+1) + O(m)$. Each update operation $H_k g_k$, where H_k is obtained by updating H_0 m times, is done in $4md + O(m)$ floating point operations.

5) FIS vs. BFGS experimental comparison: We are not aware that FIS [2] has been compared directly with any BFGS algorithm, so we coded up FIS and BFGS in Matlab and used it to train on the Reuters [12] text classification task, which has about 7000 training documents and over $d = 26000$ raw

word features. To keep memory requirements of H tractable, we chose 500 word features that had the largest mutual information wrt to the classes. (See Section II-B for details of LR-based text classification.)

Class-Name	Time Taken by FIS (in secs.)	Time taken by BFGS (in secs.)
grain	46.8	25.44
money-fx	46.06	29.32
interest	42.53	23.09

TABLE I

BFGS TRAINS SIGNIFICANTLY FASTER THAN FIS.

Not only does BFGS take significantly less time than FIS for minimization, but it also attains better objective values (not shown). There can be quite a few reasons for this. One, the BFGS approximates the Hessian of the function, hence goes up to second order of approximations whereas, FIS is just a first order approximation. Moreover, in FIS, calculating the difference vector δ at each iteration is not feasible since it takes a lot of time. So, as proposed by Jin et al. [2], we calculate this vector just once at the start and use it for all subsequent iterations to update the weight vector. This approximation may have caused the quality of FIS solution to suffer.

While BFGS is faster than iterative scaling, Liu et al. [6] showed that L-BFGS is even faster, so we did not do an in-house comparison between BFGS and L-BFGS. Text mining tools overwhelmingly often use L-BFGS.

B. Logistic regression for text classification

LR is a canonical member of the log-linear family. In the simplest case labels can take two values $Y \in \{-1, +1\}$. For text classification, it is common to create a feature $f_j(x, y)$ for each word j in the document x and each y. One common definition is $f_j(x, -1) = 0$ for all j and x, while $f_j(x, +1) = 1$ if word j appears in document x and 0 otherwise. (Some form of term weighting, like counting the number of occurrences of j in x, can also be used.) This leads to

$$\Pr(Y = +1|x) = \frac{\exp(\beta' f(x, +1))}{1 + \exp(\beta' f(x, +1))}, \tag{5}$$

because $\exp(\beta' f(x, -1)) = \exp(0) = 1$. The two class LR is commonly used for "one-vs-rest" text classification, e.g., to determine if x is or is not about cricket. Each model parameter β_j corresponds to a word. If β_j is strongly positive (respectively, negative), the existence of word j in x hints that Y is likely to be $+1$ (respectively, -1). E.g., for the cricket vs. not-cricket document classifier, we would expect $\beta_{\textit{wicket}}$ to be positive, and $\beta_{\textit{parachute}}$ to be negative. β for function words like *the*, *an* will tend to have much smaller magnitude, because they appear at similar rates in positive documents (with $Y = +1$) and negative documents (with $Y = -1$). LR does implicit feature selection by driving these β_js toward zero.

Suppose we take a word j and, in every document where j occurs, add a new unique word j'. Features f_j and $f_{j'}$ will be perfectly correlated, and j' would add absolutely no predictive power to any classifier. Clearly, the LR optimization would be at liberty to keep β_j unchanged and set $\beta_{j'} = 0$, and achieve the same accuracy as before on any test data.

However, thanks to the ridge penalty, this will not happen, because, upon including the ridge penalty into the objective, a better strategy for the optimizer would be to "split the evidence" and set $\beta_j^{\text{new}} = \beta_{j'}^{\text{new}} = \beta_j^{\text{old}}/2$, because $2\,(\beta_j^{\text{old}}/2)^2 = (\beta_j^{\text{old}})^2/2 < (\beta_j^{\text{old}})^2 + 0^2$, and the data likelihood part (2) remains unchanged whenever $\beta_j^{\text{new}} + \beta_{j'}^{\text{new}} = \beta_j^{\text{old}}$.

Summarizing, *the ridge penalty shrinks weights corresponding to redundant features toward similar values*—a vital property that we exploit. To be sure, this is not always a desirable property, in particular, the ridge penalty destroys sparseness of β even if training xs are sparse. Alternatives like the Lasso assert an L1 penalty $\|\beta\|_1 = \sum_j |\beta_j|$, which is better at keeping β sparse [13, Figure 3.9]. However, the Lasso penalty leads to a quadratic programming problem that is computationally more complex, and therefore the L2 ridge penalty is still overwhelmingly popular.

We conclude this section by noting that the feature set, with one feature for each word, is very large and highly redundant in this application. Reuters-21578 [12], a standard text classification benchmark, has about 10000 labeled documents and more than this number of distinct words.

C. CRFs for information extraction

Conditional Random Fields (CRFs) are a graphical representation of a conditional distribution $\Pr(Y|x)$ where both x and Y can have non-trivial structure (e.g., Y may be a random sequence of labels), x is observed, and properties of the distribution over Y are sought. In a linear-chain CRF commonly used for sequential tagging, training instances (x^i, y^i) are provided. Each instance i is a sequence x^i of W tokens and a sequence y^i of W labels. Positions in the sequence are indexed by $p \in \{1, \ldots, W\}$, thus we will write y_p^i as the pth state of instance i. A label could be, e.g., a part of speech, or one of person name, place name, time or date, or "none of the above". Let there be S possible labels, also called *states*. The feature vector corresponding to an instance (x, y) will be written as $F(x, y) \in \ ^d$, and is the result of computing a vector sum over feature vectors defined at each position p:

$$F(x, y) = \sum_{1 \le p \le W} f(y_{p-1}, y_p, x, p) \tag{6}$$

The full set of d scalar features in f is a concatenation of S^2 scalar *state transition* features $f(y_{p-1}, y_p, x, p) = t(y_{p-1}, y_p)$ that only depend on and expose dependencies between y_{p-1} and y_p, and something like VS scalar so-called *symbol emission* features $f(y_{p-1}, y_p, x, p) = s(y_p, x_p)$ that depend only on x_p and y_p and expose dependencies between them. Here V is the number of predicates defined on each token x_p, e.g., does the word have digits, does it start with an uppercase letter, etc. (We are simplifying a bit for exposition; in applications, V may include features defined over multiple positions of x.)

For many NLP tasks, there are also *lexicalized* predicates, one for each word in a large vocabulary. Lexicalization assists

rote learning, i.e., recognizing in test documents tokens that appeared in training documents and labeling them accordingly, and more important, in conjunction with other features, induct to the immediate neighborhood, e.g. from *York* and *New York* labeled as places, induct that *New Hampshire* is a place. To further assist this, lexicalization is sometimes extended to neighboring tokens at $p-1$ and $p+1$, leading to a threefold (or worse, if features combine x_{p-1} and x_p in any way) increase in the number of features from a set that already ranges into tens of thousands. A million features is not uncommon in NLP tasks.

Given a trained weight vector $\beta \in \ ^d$ and an instance (x, y), we can write $\Pr(y|x) = \Pr(y_1, \ldots, y_W|x) =$

$$\frac{\exp(\beta' F(x,y))}{Z(x)} = \frac{\exp\left(\beta' \sum_p f(y_{p-1}, y_p, x, p)\right)}{Z(x)} \quad (7)$$

Finding the most likely label sequence means finding $\arg\max_y \beta' F(x, y)$. We do not wish to enumerate all S^W possible label sequences, so dynamic programming is used instead. Let $A(y, p)$ be the unnormalized probability of a labeling of positions 1 through p ending in state y. $A(y, p)$ can be defined inductively as

$$A(y, 0) = \ y = \textit{beginState} \ , \quad \text{and} \quad (8)$$

$$A(y,p) = \sum_{y'} A(y', p-1) \exp(\beta' f(y', y, x, p)) \quad (9)$$

Similarly we can define the unnormalized probability of a labeling of positions p through W, starting with state y:

$$B(y, W+1) = \ y = \textit{endState} \ , \quad \text{and} \quad (10)$$

$$B(y,p) = \sum_{y'} \exp(\beta' f(y, y', x, p)) B(y', p+1). \quad (11)$$

Note that $\sum_y A(y, W) = Z(x) = \sum_y B(y, 1)$.

For training the CRF using L-BFGS, as in LR, we must estimate the gradient wrt each β_j, corresponding to the jth element $f_j(y, y', x, p)$ of feature vector $f(y, y', x, p)$. To within a constant, the gradient of ℓ_0 in (2) is

$$\frac{\partial \ell(\beta)}{\partial \beta_j} = \sum_i \left(F_j(x^i, y^i) - E_{Y|x^i} F_j(x^i, Y)\right), \quad (12)$$

and we generally add on $-\beta_j/\sigma^2$ corresponding to the ridge penalty $\beta_j^2/(2\sigma^2)$. The crux is to compute $E_{Y|x^i} F_j(x^i, Y)$, which, by linearity of expectation, can be written as

$$\sum_p E_{Y|x^i} f_j(Y_{p-1}, Y_p, x^i, p). \quad (13)$$

Note that Y, not y^i is involved in the above expression. I.e., we must sum over all possible Y_{p-1} and Y_p. Again, through dynamic programming this can be computed via A and B:

$$\sum_p E_{Y|x^i} f_j(Y_{p-1}, Y_p, x^i, p)$$
$$= \sum_p \sum_{y,y'} A(y, p-1) \underline{f_j(y, y', x^i, p) e^{\beta' f(y,y',x^i,p)}} B(y', p)$$

III. Speeding up feature-vector classification

In the previous Section we saw the usefulness of BFGS update and its limited memory version L-BFGS in solving large-scale optimization problems. When L-BFGS is applied to text classification and information extraction, it is used as a black-box, and no specific advantage is taken of the peculiarities of those problem domains. In this Section we seek to remedy this limitation.

As mentioned in Section II-B, log-linear text classification models treat each word of the document as a potential attribute, leading easily to instances with d in the tens of thousands. In multi-label (as against one-vs-rest) classification, this would generally be multiplied by the number of class labels, potentially leading to millions of features of the form $f_j(x, y)$ where j ranges over words and y over class labels. If the labels are arranged in a hierarchy, the feature space may be even larger.

An important property of text classification problems is that, although there are potentially many features that can determine the label of a document, very few features actually end up getting significant weights β_j that decide the score of the document. The remaining words are regarded as noise. It is also common to find two informative words strongly correlated with each other, e.g. *Opteron* and *HyperTransport*. As explained in Section II-B, their corresponding weights may end up being very similar in value.

A. Motivating examples

To see if this is indeed the case, we trace the evolution of the β vector, iteration by iteration, in a text classification application. For these preliminary experiments, we run BFGS provided by MATLAB on the Reuters data. To keep memory requirements of H tractable, we chose 500 word features that had the largest mutual information wrt to the nine most frequent classes. Figure 2 shows how the weights of these 500 "important" terms evolve.

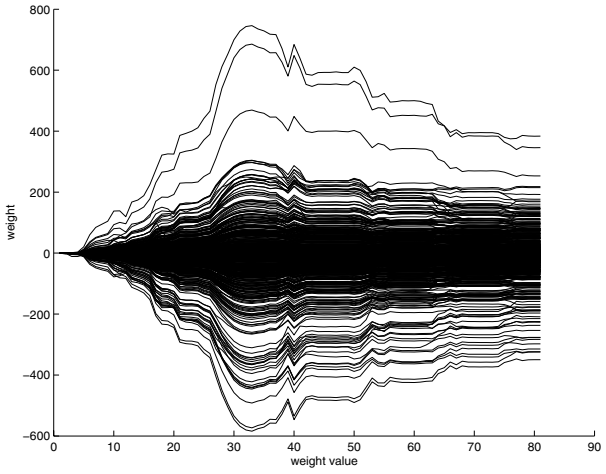

Fig. 2. Evolution of β in BFGS.

A close-up is shown in Figure 3. We point out two important features. First, many weights co-evolve in largely parallel

trajectories, and, after the "big bang" moment has passed, β_js clustered close together usually tend to remain clustered close together. Tracing back from the features to class labels and words gives some interesting insight. E.g., one tight cluster of contiguous β_js corresponded to class *grain* in the Reuters data, and these words: marketing, price, imports, annual, average, traders, credit, exporters. Meanwhile, another tight cluster of β_js corresponded to class *grain* and these words: was, these, been, among, like, can, what, say, given, and the meta-word "digits"—for topic *grain*, these were all low-signal words.

However, it is important to recognize that there *are* exceptions to the "persistent cluster" hypothesis, and *crossovers* do happen, and therefore, any algorithm that seeks to exploit redundancy between clusters must occasionally regroup features.

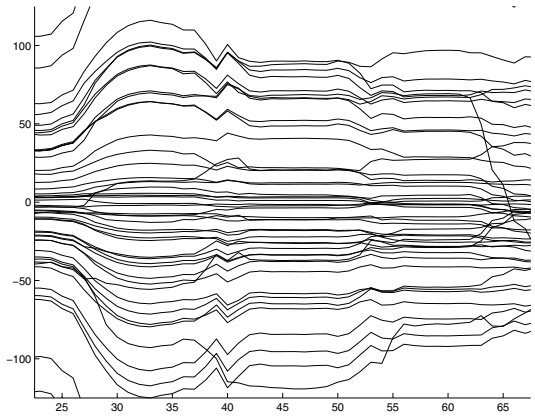

Fig. 3. β evolution, close-up. Note the occasional crossover β_js slashing vertically across.

B. The feature clustering technique

From the above discussion, we come up with the pseudocode shown in Figure 4. The basic idea is to run `fineBFGS` for a while, cluster the d features into $d' < d$ groups, project down the weight vector $\beta \in \ ^{d}$ to a coarse approximation $\gamma \in \ ^{d'}$, run a faster `coarseBFGS` procedure over γ, project back up from γ to β, and run a "patch-up" optimization over β. If needed we can repeat several alternating rounds between `fineBFGS` and `coarseBFGS` before the final patch-up.

cmap is an index map from feature IDs in $[1, d]$ to cluster IDs in $[1, d']$. Given *cmap*, β can be reduced to γ by averaging values in each cluster. Conversely, we can copy back $\beta_j = \gamma_{cmap(j)}$. The only change required for specific applications is to generalize the code for calculating $\ell(\beta)$ and $\nabla_\beta \ell$ to also calculate $\ell(\gamma)$ and $\nabla_\gamma \ell$ (see this and the next Section).

The intuition is that d' being much smaller than d, `coarseBFGS` will run much faster than `fineBFGS`, but will have the benefit of reducing the objective and providing the next copy of `fineBFGS` with a better starting point, which will also ease convergence.

The important policy choices are embedded in these variables:

Clustering: How we represent features and how we cluster them are important considerations.

```
Let initial approximation to the solution be β
for numRounds do
    for fineIters iterations do
        [f, g] ← computeFunctionGradient(β)
        β ← fineBFGS(β, f, g)
    end for
    Set numClusters = d/clusterFactor
    Prepare the feature cluster map
    cmap ← CLUSTER(β, numClusters)
    γ ← projectDown(β, cmap)
    for coarseIters iterations do
        [φ, δ] ← computeFunctionGradient(γ)
        γ ← coarseBFGS(γ, φ, δ)
    end for
    β ← projectUp(γ, cmap)
end for
while not converged do
    [f, g] ← computeFunctionGradient(β)
    β ← fineBFGS(β, f, g)
end while
```

Fig. 4. Pseudocode of the proposed feature clustering approach.

***numRounds*:** The number of fine-coarse alternations. 1–2 always suffice.

***fineIters*:** The number of iterations allowed to a `fineBFGS` invocation.

***coarseIters*:** Likewise for `coarseBFGS`. In fact, we may need a device other than number of iterations to terminate `coarseBFGS`, and not run to convergence.

***clusterFactor*:** A target for the reduction factor d/d'. A criteria different from reduction factor (e.g. square error) may also be used.

The above policies should try to ensure that very little time is spent in `fineBFGS`, and that when `coarseBFGS` terminates, its output can be used to land `fineBFGS` close to the optimum.

We do not have perfect recommendations for all these policies for all learning problems. However, for a specific domain and application, only a few trial runs suffice to tune the above policies quite effectively. Unlike in a one-shot learning exercise, this small effort will be paid back in a continual or lifelong learning scenario, e.g. in operational text classification.

Clustering algorithm: We experimented with two clustering approaches. L-BFGS is very fast and we need a very lightweight clustering algorithm that consumes negligible CPU compared to the optimization itself. For L-BFGS, we simply sorted the current β values and performed a one-dimensional clustering with a fixed cap of *clusterFactor* on the cluster size. This *contiguous chunking* approach already performed reasonably well.

BFGS, being somewhat slower than L-BFGS, gives us a little more time for clustering. Here we tried to bring in a second attribute of each feature: the temporal behavior of its weight in recent iterations. Each feature f_j is characterized

by two numbers: its weight $\beta_j^{(t)}$ in the current iteration, and the latest change $\beta_j^{(t)} - \beta_j^{(t-1)}$. The intention was to prevent clustering together feature pairs where one is momentarily crossing over the other, e.g. the sample in Figure 3. However, this did not make a significant difference, probably because crossovers are rare overall.

In our experiments so far, we have found that *clusterFactor* matters more than the clustering algorithm itself.

Final patch-up: Note that we always finish by running a fine L-BFGS over β. This ensures that we end up as close to the original optimum as possible. This process actually brings our algorithm back near the actual optimum, correcting if `coarseBFGS` has taken our solution far from it. In our experiments, we sometimes found that `coarseBFGS` actually helped the patch-up stage to find a β *better* than the baseline.

A bad clustering can take our solution far from original optimum, implying that we would have to spend more time in the final patch-up to get back near the original optimum. This implies that we need a good clustering algorithm to ensure that we spend as little time in patching up as possible.

We will discuss the remaining issues while narrating our experiments in Section III-D.

C. Computing $\ell(\gamma)$ and $\nabla_\gamma \ell$

Our recipe requires that, starting with code to compute $\ell(\beta)$ and $\nabla_\beta \ell$, we write code to compute $\ell(\gamma)$ and $\nabla_\gamma \ell$, for a given clustering expressed through *cmap*. Computing $\ell(\gamma)$ is trivial, and so is $\nabla_\gamma \ell$:

$$\frac{\partial \ell}{\partial \gamma_k} = \sum_j \frac{\partial \ell}{\partial \beta_j} \frac{\partial \beta_j}{\partial \gamma_k} = \sum_j \frac{\partial \ell}{\partial \beta_j} \begin{cases} 1 & \textit{cmap}(j) = k \\ 0 & \text{otherwise} \end{cases}$$
$$= \sum_{j:\textit{cmap}(j)=k} \frac{\partial \ell}{\partial \beta_j} \tag{14}$$

Most log-linear training routines initialize an array of gradients wrt β, of size d, and update it additively:

```
1: gradWrtBeta[1...d] ← 0
2: for each instance i do
3:    for each feature index j do
4:       gradWrtBeta[j] += ...
5:    end for
6: end for
```

The above code is simply replaced with the many-to-one accumulation:

```
1: gradWrtGamma[1...d'] ← 0
2: for each instance i do
3:    for each feature index j do
4:       gradWrtGamma[cmap(j)] += ...
5:    end for
6: end for
```

It is thus trivial to transform `fineBFGS` into `coarseBFGS` completely mechanically.

D. Experiments with BFGS

To further motivate the point, we show the results of the experiments done in MATLAB using its in-built minimization subroutine which uses BFGS updates. We use the same 500-attribute version of the Reuters Corpus. We have each document represented in a 500 dimensional feature space and we choose just one of the 9 classes for the one-vs-rest problem. We call the basic optimization process as "basic optimizer" and our algorithm as the "cluster optimizer." The policy decisions were:

- Fix the value of *numRounds* to one
- Run `coarseBFGS` to convergence
- Fix the initial *fineIters* to 5
- Fix dimensionality reduction factor *clusterFactor* $= 15$
- Use the `KMEANS` subroutine in MATLAB for clustering. Represent each feature f_j as a point in two dimensional feature space as explained before.

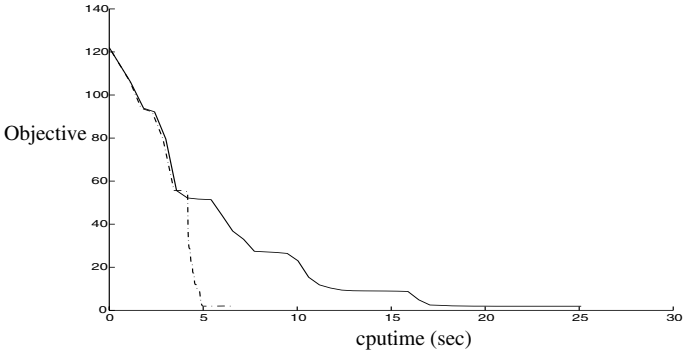

Fig. 5. Objective $\ell_0(\beta)$ vs. CPU time of basic and cluster optimizer.

Figure 5 compares the value of $\ell_0(\beta)$ vs. CPU time in seconds (no ridge penalty was used here, but it will be in later experiments). The bold line denotes standard BFGS in original feature space. The dotted line denotes our algorithm working (part of the time) in the projected feature space with parameters γ. When `coarseBFGS` completes, final patch-up iterations are never more than one or two. We see that our clustering collapses the feature space nicely, and the new features have captured all the key variations of the original feature space to drive the function faster toward the optimum.

E. Experiments with L-BFGS

Encouraged with our MATLAB experiments, we implemented the scheme in Java on top of the publicly-available and widely-used L-BFGS package from `http://riso.sourceforge.net/LBFGS-20020202.java.jar`.

1) Setting the ridge parameter: The "regularizer" or the ridge penalty $1/\sigma^2$ in (3) is an important parameter, not only to the accuracy of the learnt model, but also to the training time. Since this parameter provides a check on $\|\beta\|_2$, it determines the number of cross-overs in the β vector evolution. There is no algorithmic solution known to find a good ridge penalty for a given problem and a given dataset, so we find the best ridge parameter using cross validation. For the Reuters data, a plot of F_1-score vs. ridge parameter (by which we generally mean $1/\sigma^2$) is shown in Figure 6.

F_1 score is a standard accuracy measure defined in terms of recall and precision. Let TP be the number of true positives, that is, the number of documents on which were marked and predicted to be positives. FN be the false negatives, that is, the number of documents that were marked positive but predicted negative; FP be the false positives marked negative but predicted positives. Then recall is defined as $R = \text{TP}/(\text{TP}+\text{FN})$ and precision is defined as $P = \text{TP}/(\text{TP}+\text{FP})$. F_1 is the harmonic mean of recall and precision: $F_1 = 2RP/(R+P)$.

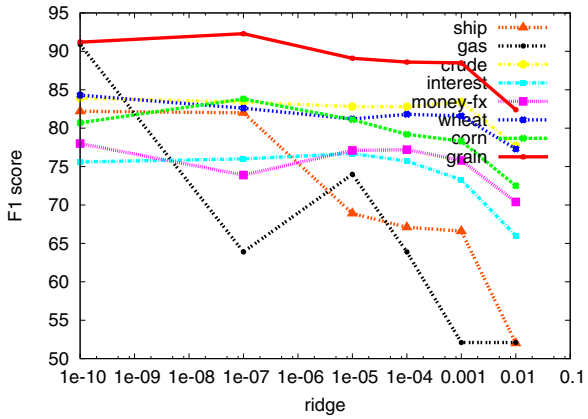

Fig. 6. Test F1 score vs. ridge parameter for some topics in Reuters.

From Figure 6, it is evident that lower values of ridge penalty are preferable for Reuters dataset. 10^{-7} and 10^{-10} win on test F1-score for most of the topics. We choose both these values for comparisons in the sections that follow.

Figure 7 shows the performance implications of the ridge parameter on the training time of the basic LR algorithm. It is curious to note that training becomes most time-consuming for a middle range of ridge parameters, where the test accuracy is not the best possible! Although LR is so widely used, we have seen no detailed report of the accuracy and performance implications of the ridge parameter. It turns out that our idea accelerates L-BFGS *at the best ridge value*, and is therefore *even faster* compared to the worst-case training time.

2) `coarseBFGS` termination policy: For every call to `coarseBFGS`, since it is faster than `fineBFGS`, our goal is to let it reach as close to its optimum as possible. Hence, we do not put any upper bound on the number of coarse iterations for any call to `coarseBFGS`. However, the last few coarse iterations, without any significant change in the objective, are of little use to us. We do not want `coarseBFGS` to drag slowly towards it optimum in the end. We leave full termination for fine patch-up and decide to terminate `coarseBFGS` when the relative change it brings in the objective in last w iterations, is less than some δ. That is, when $(\max \ell(\gamma) - \min \ell(\gamma)) < \delta \min \ell(\gamma)$, where $\max \ell(\gamma)$ and $\min \ell(\gamma)$ are the maximum and minimum objective values in the last w iterations.

Through trial-and-error we arrived at choices of history size $w = 5$ and $\delta = 10^{-4}$, which worked well for all our experiments. However these were not very sensitive values.

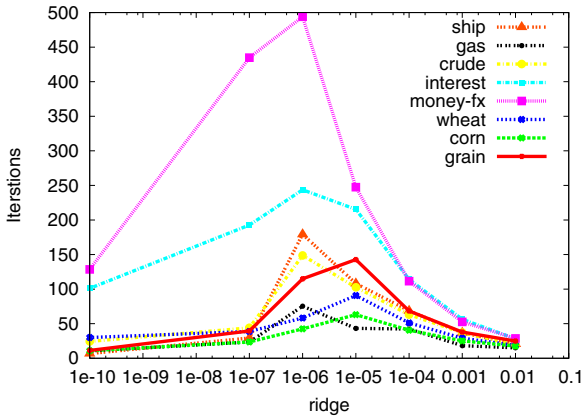

Fig. 7. Training time (seconds) against the ridge parameter for the basic optimizer.

3) `projectUp` and `projectDown`: While going from the original space to the reduced space (`projectDown`), we use the cluster centroids (obtained by averaging of weights of features mapped to same cluster) as the γ vector in the reduce feature space.

We compute the function value and gradient by *aggregating* the features using *cmap*. Note that there is a call to `computeFunctionGradient` before each iteration of L-BFGS. We pass *cmap* to it, to calculate the function value and gradient in reduced feature space on the fly.

While going back from the reduced space to the original space via `projectUp`, we just replicate the values as $\beta_j = \gamma_{cmap(j)}$. `computeFunctionGradient` then calculates the objective value and gradient vector in the original space because calls to `coarseBFGS` are implemented using the identity map from $[1, d]$ to $[1, d]$ passed as *cmap*.

4) numRounds*, the number of alternations:* The parameter *numRounds* determines how many calls to `coarseBFGS` we would make. *numRounds* should not be too small, or the final `fineBFGS` patch-up will take a lot of time. It should not be too large, or we would be doing unnecessary work in `coarseBFGS`, while `fineBFGS` was already getting close to the optimum in original space.

Luckily, the choice of *numRounds* is easy. In practice, we have seen that a value between 1 and 3 suffice for all class labels and both our application domains. In Table II, we compare the optimization time for three classes and ridge parameter 10^{-7} with several values of *numRounds*.

As is clear from the table, there is no clear winner among *numRounds* = 1 or 2. In practice, we choose the *numRounds* = 2, to ensure that we are not over doing any work and at the same time we have less fine patch-up iterations as well.

5) Comparing basic and cluster optimizers: Fixing the policies as above, we run our first comparison of basic optimizer and cluster optimizer. Figure 8 shows the time trace of objective $\ell(\beta)$ in the basic and cluster optimizers for two classes and ridge parameter 10^{-7}.

It is immediately visible that, as in BFGS, here, too, the

Class-Name	Policy	Training Time for ridge = 10^{-7} (in sec)
Grain	basic	33.7
	numRounds = 3	13.9
	numRounds = 2	9.7
	numRounds = 1	10.3
Money-fx	basic	338.5
	numRounds = 3	82.9
	numRounds = 2	52.3
	numRounds = 1	36.9
Gas	basic	22.0
	numRounds = 3	11.7
	numRounds = 2	10.3
	numRounds = 1	8.3

TABLE II

TRAINING TIME OF CLUSTER OPTIMIZER VS. *numRounds*.

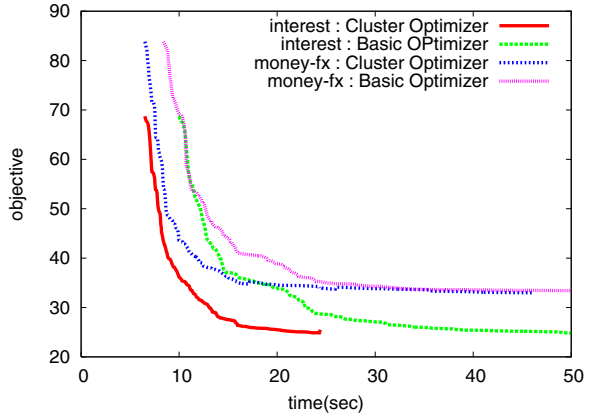

Fig. 8. Time trace of objective in basic vs. cluster optimizer for ridge = 10^{-7}. We show time only up to 50 seconds and the basic optimizer has not converged yet. For full time comparison view Table III.

clustered approach drives down the objective much quicker than the basic L-BFGS optimizer. What is not visible in Figure 8 is that the clustered approach also terminates faster, which is shown in Table III.

Class-Name		**ridge** = 10^{-7}		**ridge** = 10^{-10}	
		F1	time(sec)	F1	time(sec)
money-fx	basic	73.9	338.5	78.0	293.5
	cluster	76.1	46.8	77.1	32.0
wheat	basic	82.6	31.8	84.3	29.8
	cluster	82.0	11.7	82.4	11.8
interest	basic	76.0	162.7	75.6	48.7
	cluster	68.3	24.5	74.9	27.1

TABLE III

COMPARISON OF TRAINING TIME, F1 AND FUNCTION VALUE OF BASIC AND CLUSTER OPTIMIZER

For some of the major classes in the Reuters corpus, clustered L-BFGS is almost 10 times faster than basic L-BFGS; typically, it is a factor of 2–3 faster. We emphasize that this gain is CPU-invariant, in the sense that it will persist as CPUs become faster.

Table IV gives a break-up account of how the time is spent in `fineBFGS`, `coarseBFGS` and clustering itself by clustered L-BFGS. Clustering time itself is negligible, and `fineBFGS` and `coarseBFGS` divide up the time into fair-sized chunks, while their sum remains far below the time taken by basic L-BFGS.

Class-Name		**ridge** = 10^{-7}	**ridge** = 10^{-10}
money-fx	basic time	338.5	293.5
	fineBFGS time	17.1	14.3
	cluster time	0.115	0.11
	coarseBFGS time	47.3	32.0
interest	basic time	162.7	48.7
	fine time	7.4	7.6
	cluster time	0.054	0.059
	coarseBFGS time	24.5	27.1

TABLE IV

ACCOUNT OF TIME SPENT IN CLUSTER-BASED L-BFGS.

Another important measurement is the F_1 score on a separate test set as the optimizer spends time on the training data. Figure 9 shows that initial growth in test F_1 using clustered L-BFGS is indistinguishable from baseline L-BFGS, but over-iterating can lead to some instability. Therefore test-set validation is recommended.

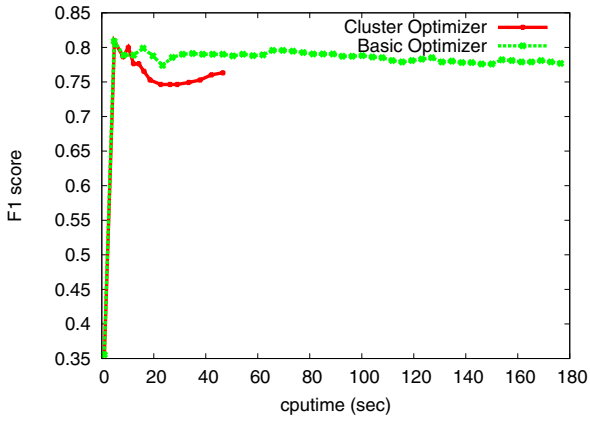

Fig. 9. Test F1 vs. time spent by basic and clustered L-BFGS on fixed training data

6) clusterFactor *and feature pre-aggregation:* While calculating $\ell(\gamma)$ and $\nabla_\gamma \ell$, rather than first calculate $\nabla_\beta \ell$ and then transform them to $\nabla_\gamma \ell$ as in (14), we can exploit the linear interaction between β and γ with f by pre-aggregating feature values down to the γ space. For $k = 1, \ldots, d'$, we define new "pseudofeatures"

$$\hat{f}_k(x, y) = \sum_{j:cmap(j)=k} f_j(x, y). \tag{15}$$

and then compute $\ell(\gamma)$ using $\hat{f}$ alone.

Pre-aggregation is needed after every round of clustering, but the results are exploited repeatedly inside `coarseBFGS`. A large *clusterFactor* can save a large number of floating point operations per iteration of `coarseBFGS`. But at the same time, larger the *clusterFactor*, the worse is the approximation

involved in `coarseBFGS` and hence more work is required during the final fine patch-up. There is a trade-off in the quality of solution (and time required for training) and *clusterFactor*. Therefore, we study their combined effect in Table V.

Class-Name	*clusterFactor*	Training Time for ridge = 10^{-7} (in seconds)
Wheat	10	16.0
	20	15.7
	50	17.3
Money-fx	10	64.3
	20	44.5
	50	39.4
Interest	10	61.3
	20	44.3
	50	87.6

TABLE V

EFFECT OF FEATURE PRE-AGGREGATION AND *clusterFactor* ON TRAINING TIME OF CLUSTER L-BFGS.

Increasing *clusterFactor* beyond 50 adversely affects our solution. The higher training time at 50, as compared to other factors, can be attributed to increased fine patch-up iterations, due to poorer quality of approximation in the reduced space. There are some other classes that are not so sensitive the quality of approximation, and for those cases we do save training time. Empirically, 50 is too large a value, and lower values are preferred. At lower values, the benefits of pre-aggregation are modest. Only some classes like *money-fx* with ridge parameter 10^{-7} show significant reduction in time by pre-aggregation. What this implies is that the reduction in training time we observed is, to a large extent, because of the simplification of feature space in `coarseBFGS`.

IV. SPEEDING UP CRF TRAINING

We chose Named Entity Recognition (NER) as our task of sequential labeling. In NER, each word is labeled with an entity tag indicating whether it is a mention of a person, an organization, a place or none of those. We use the CoNLL dataset, which provides the part-of-speech (POS) tags and entity tags for Reuters-21578 data. We have used the current word, the previous word and POS tag of the current word as the attributes which form our symbol emission features.

A. β evolution in NER

To see that our approach is promising for NER as well, we repeat the β evolution study from Section III-A. We ran a CRF (`http://crf.sf.net`) trainer with standard L-BFGS for NER on CoNLL data. Figure 10 shows the results of some features evolving with time. A close-up is shown in Figure 11.

Crossovers appear more frequent than in LR-based text classification, but Figure 11 shows that there are a lot of similarly evolving features, too. These observations hint that we need to keep *clusterFactor* small, and that we can expect more iterations during the final patch-up.

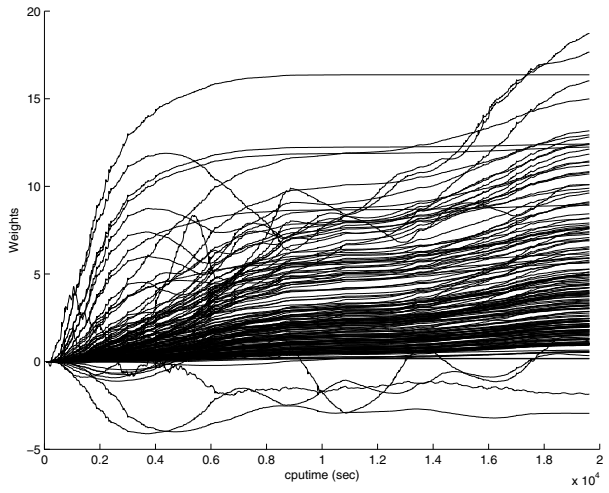

Fig. 10. Evolution of β in NER.

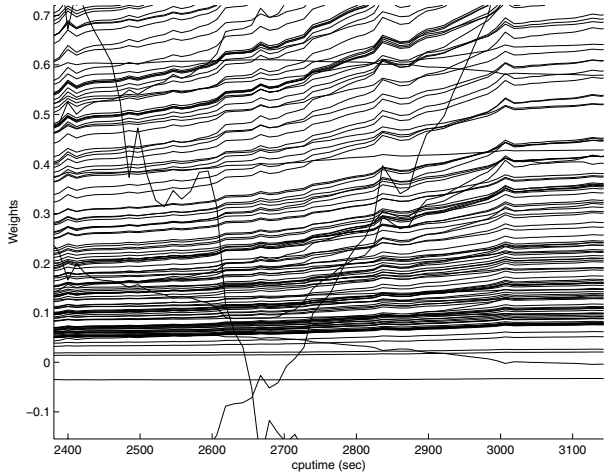

Fig. 11. Evolution of β in NER, close up. Crossovers are more common than in LR.

B. Computing $\ell(\gamma)$ and $\nabla_\gamma \ell$

We limit feature clustering to the symbol emission features $s(y_p, x)$ because for NER tasks too much of the model information would get corrupted if we allowed transition features to be clustered.

A brief inspection of the formulas in Section II-C leading up to expressions for $\ell(\beta)$ and $\nabla_\beta \ell$ readily reveals that the transformation (14) can still be computed using the same code style outlined in Section III-C.

Pre-aggregation of features takes slightly more involved software engineering and is deferred to future work.

C. Experiments and results

Gaining experience from the experiments in Section III-E, we set the policies of our algorithm as follows.

- We chose the contiguous chunking approach to clustering instead of `KMEANS`.
- *numRounds* was chosen to be one.

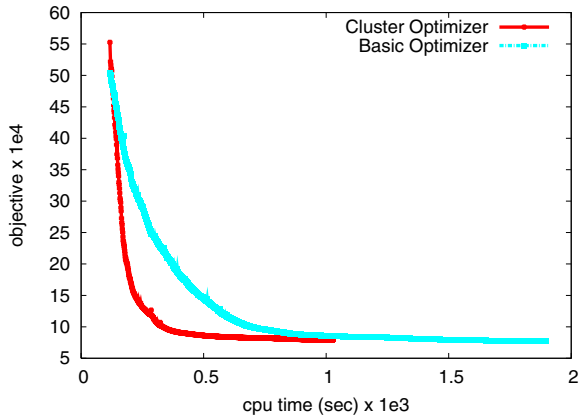

Fig. 12. Objective vs. CPU time for CRF

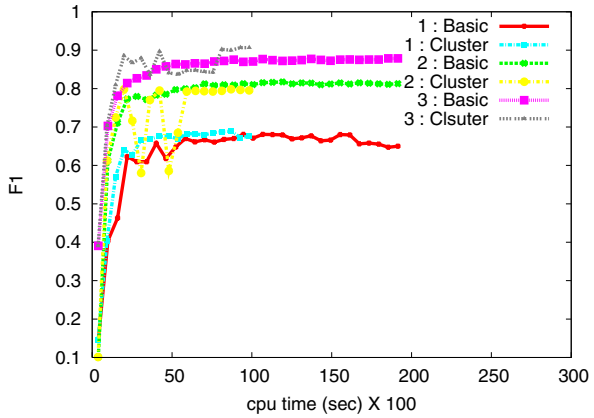

Fig. 13. F_1 vs. CPU time of basic and clustered optimizers for some entities types. Type 1 is organization, 2 is location and 3 is person.

- *clusterFactor* is set to 10, a lower value than in Section III-E, considering the increased rate of crossovers.
- *fineIters* was set to 100, as before.
- We did not do any pre-aggregation of features.
- The ridge parameter was set to 10^{-7}.

Fixing the policies as above, we show the objective attained by the basic and clustered optimizers against CPU time in Figure 12.

As with LR, the clustered optimizer reduces the objective much more quickly, and attains convergence in half the time taken by the baseline NER implementation.

To ensure that this has not led to any significant damage to the quality of the solution, we plot the F_1 score of some sample entity types for both the basic and clustered optimizers. The result is shown in Figure Figure 13. The F_1 score grows more quickly in the clustered optimizer, and quickly becomes essentially comparable to the baseline accuracy. In fact, about half the time, the clustered tagger achieves test accuracy slightly higher than the baseline tagger.

Summarizing, the clustered optimization approach speeds up NER tasks substantially, achieving a 50% reduction in time to convergence. It reaches baseline F_1 scores even faster, and there is no systematic loss of accuracy.

V. Conclusion

We have proposed a very simple and easy-to-implement "wrapper" around state-of-the-art quasi-Newton optimizers that can speed up training text learning applications by large factors between 2 and 9. We have achieved this by exploiting natural co-evolving clusters of model parameters in high-dimensional text learning applications, as well as the specific form of log-linear models for learning in vector spaces and graphical models. No understanding of the math behind the Newton optimizer is required to implement our proposal. A mechanical and local rewriting of the routines that compute the objective and gradient suffice. However, a slightly more involved feature pre-aggregation step may buy even larger performance benefits; this is ongoing work. We are also interested in extending the approach to other general graphical inference and training algorithms, and exploring more complex β to γ transformations.

References

[1] K. Nigam, J. Lafferty, and A. McCallum, "Using maximum entropy for text classification," in *IJCAI-99 Workshop on Machine Learning for Information Filtering*, 1999, pp. 61–67, see http://www.cs.cmu.edu/~knigam/ and http://www.cs.cmu.edu/~mccallum/papers/maxent-ijcaiws99.ps.gz.

[2] R. Jin, R. Yan, J. Zhang, and A. G. Hauptmann, "A faster iterative scaling algorithm for conditional exponential model," in *ICML*, 2003, pp. 282–289. [Online]. Available: http://www.hpl.hp.com/conferences/icml2003/papers/79.pdf

[3] J. Lafferty, A. McCallum, and F. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *ICML*, 2001.

[4] F. Sha and F. Pereira, "Shallow parsing with conditional random fields," in *HLT-NAACL*, 2003, pp. 134–141. [Online]. Available: http://acl.ldc.upenn.edu/N/N03/N03-1028.pdf

[5] S. Godbole, A. Harpale, S. Sarawagi, and S. Chakrabarti, "Document classification through interactive supervision of document and term labels," in *PKDD*, 2004, pp. 185–196.

[6] D. C. Liu and J. Nocedal, "On the limited memory BFGS method for large scale optimization," *Math. Programming*, vol. 45, no. 3, (Ser. B), pp. 503–528, 1989. [Online]. Available: citeseer.ist.psu.edu/liu89limited.html

[7] S. J. Benson and J. J. Moré, "A limited memory variable metric method for bound constraint minimization," Argonne National Laboratory, Tech. Rep. ANL/MCS-P909-0901, 2001.

[8] S. V. N. Vishwanathan, N. N. Schraudolph, M. Schmidt, and K. P. Murphy, "Accelerated training of CRFs with stochastic gradient methods," in *ICML*, 2006. [Online]. Available: http://users.rsise.anu.edu.au/~vishy/papers/VisSchSchMur06.pdf

[9] A. L. Berger, S. A. D. Pietra, and V. J. D. Pietra, "A maximum entropy approach to natural language processing," *Computational Linguistics*, vol. 22, no. 1, p. 3971, 1996.

[10] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004. [Online]. Available: http://www.stanford.edu/~boyd/cvxbook/

[11] C. G. Broyden, "A class of methods for solving nonlinear simultaneous equations," *Mathematics of Computation*, vol. 19, no. 92, pp. 577–593, Oct. 1965.

[12] D. D. Lewis, "The reuters-21578 text categorization test collection," 1997, available at http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.html.

[13] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. Springer, 2001.