CS 594: Advanced Topics in Machine Learning **Due: 17:00, Nov 4, 2016**

Project 1: Convex Optimization

Student: Your Name(s) *Email: NetID(s)@uic.edu*

In this project you will implement a conditional random field for optical character recognition (OCR), with an emphasis on optimization. You are provided with utility code in Torch and PETSc, and you must stick to the given framework.

You can work in teams of at most three people. **Only one member of the team needs to submit the work on Blackboard.** The due date for this project is **17:00 on Nov 4, 2016**. You are allowed to resubmit as often as you like and your grade will be based on the last version submitted. Late submissions will not be accepted in any case, unless there is a documented personal emergency. Arrangements must be made with the instructor as soon as possible after the emergency arises, preferably well before the due date. This project is worth **30%** of your final grade.

You must submit the following TWO files on Blackboard:

1. A PDF report with answers to the questions outlined below. **Your report should include the name and NetID of *all* team members.** The LaTeX source code of this document is provided with the package, and you may write up your report based on it.
2. A tarball or zip file which includes your source code. Your code should be well commented. If you changed the format of command line, then include a short readme.txt file that explains how to run your code.

Please submit **TWO files separately**, and do NOT include the PDF report in the tarball/zip.

Start working on the project early because a considerable amount of work will be needed. The workload is designed for 2.5 people, and a smaller group size does not warrant any extra credit or reduction in workload.

Overview

In this project, we will build a classifier which recognizes "words" from images. This is a great opportunity to pick up *practical experiences* that are crucial for successfully applying machine learning to real world problems, and evaluating their performance with comparison to other methods. To focus on learning, all images of words have been segmented into letters, with each letter represented by a 16*8 small image. Figure 1 shows an example word image with five letters. Although

Figure 1. Example word image

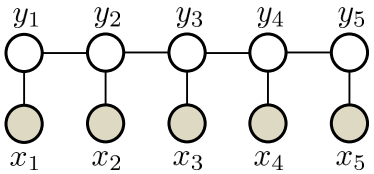


Figure 2. CRF for word-letter

recognition could be performed letter by letter, we will see that higher accuracy can be achieved by recognizing the word as a whole.

Dataset The original dataset is downloaded from http://www.seas.upenn.edu/~taskar/ocr. It contains the image and label of 6,877 words collected from 150 human subjects, with 52,152 letters in total. To simplify feature engineering, each letter image is encoded by a 128 (=16*8) dimensional vector, whose entries are either 0 (black) or 1 (white). The code that loads the data always appends a constant 1 to the feature vectors, leading to 129 features in total. The 6,877 words are divided evenly into training and test sets, provided in `data/train.txt` and `data/test.txt` respectively. The meaning of the fields in each line is described in `data/fields.txt` (the LIBSVM format).

Note in this dataset, only lowercase letters are involved, *i.e.* 26 possible labels. Since the first letter of each word was capitalized and the rest were in lowercase, the dataset has removed all first letters.

Performance measures We will compute two error rates: *letter-wise* and *word-wise*. Prediction/labeling is made on at letter level, and the percentage of incorrectly labeled letters is called letter-wise error. A word is correctly labeled if and only if *all* letters in it are correctly labeled, and the word-wise error is the percentage of words in which at least one letter is mislabeled.

1 Conditional Random Fields

Suppose the training set consists of n words. The image of the t-th word can be represented as $X^t = (\mathbf{x}_1^t, \ldots, \mathbf{x}_m^t)^\top$, where t is a superscript (not exponent) and each *row* of X^t is a letter. Here m is the number of letters in the word, and $\mathbf{x}_j^t$ is a 129 dimensional vector that represents its j-th letter image. To ease notation, we simply assume all words have m letters, and the model extends naturally to the general case where the length of word varies. The sequence label of a word is encoded as $\mathbf{y}^t = (y_1^t, \ldots, y_m^t)$, where $y_k^t \in \mathcal{Y} := \{1, 2, \ldots, 26\}$ represents the label of the k-th letter. So in Figure 1, $y_1^t = 2$, $y_2^t = 18$, $\ldots$, $y_5^t = 5$.

Using this notation, the Conditional Random Field (CRF) model for this task is a sequence shown in Figure 2, and the probabilistic model for a word/label pair $(X, \mathbf{y})$ can be written as

$$p(\mathbf{y}|X) = \frac{1}{Z_X} \exp\left(\sum_{s=1}^{m} \langle \mathbf{w}_{y_s}, \mathbf{x}_s \rangle + \sum_{s=1}^{m-1} T_{y_s, y_{s+1}}\right) \tag{1}$$

$$\text{where} \quad Z_X = \sum_{\hat{\mathbf{y}} \in \mathcal{Y}^m} \exp\left(\sum_{s=1}^{m} \langle \mathbf{w}_{\hat{y}_s}, \mathbf{x}_s \rangle + \sum_{s=1}^{m-1} T_{\hat{y}_s, \hat{y}_{s+1}}\right). \tag{2}$$

$\langle \cdot, \cdot \rangle$ denotes inner product between vectors. Two groups of parameters are used here:

- **Node weight:** Letter-wise discriminant weight vector $\mathbf{w}_k \in \mathbb{R}^{129}$ for each possible letter label $k \in \mathcal{Y}$;
- **Edge weight:** Transition weight matrix T which is sized 26-by-26. T_{ij} is the weight associated with the letter pair of the i-th and j-th letter in the alphabet. For example $T_{1,9}$ is the weight for pair ('a', 'i'), and $T_{24,2}$ is for the pair ('x', 'b'). In general T is not symmetric, *i.e.* $T_{ij} \neq T_{ji}$.

Given these parameters (*e.g.* by learning from data), the model (1) can be used to predict the sequence label (*i.e.* word) for a new word image $X^* := (\mathbf{x}_1^*, \dots, \mathbf{x}_m^*)^\top$ via the so-called maximum a-posteriori (MAP) inference:

$$\mathbf{y}^* = \operatorname*{argmax}_{\mathbf{y}\in\mathcal{Y}^m} p(\mathbf{y}|X^*) = \operatorname*{argmax}_{\mathbf{y}\in\mathcal{Y}^m} \left\{ \sum_{j=1}^{m} \langle \mathbf{w}_{y_j}, \mathbf{x}_j^* \rangle + \sum_{j=1}^{m-1} T_{y_j,y_{j+1}} \right\}. \tag{3}$$

Finally, given a training set $\{X^t, \mathbf{y}^t\}_{t=1}^n$ (n words), we can estimate the parameters $\{\mathbf{w}_k : k \in \mathcal{Y}\}$ and T by maximum a posteriori over the conditional distribution in (1), or equivalently

$$\min_{\{\mathbf{w}_k\},T} \; -\frac{1}{n}\sum_{t=1}^{n} \log p(\mathbf{y}^t|X^t) + \frac{\lambda}{2}\left(\sum_{k\in\mathcal{Y}} \|\mathbf{w}_k\|^2 + \frac{1}{2}\sum_{ij} T_{ij}^2 \right). \tag{4}$$

Here $\lambda > 0$ is a trade-off weight that balances log-likelihood and regularization.

1.1 Gradient formula

For a single word-label pair $(X^t, \mathbf{y}^t)$, it is not hard to derive the formulae of $\nabla_{\mathbf{w}_k} \log p(\mathbf{y}^t|X^t)$ and $\nabla_{T_{ij}} \log p(\mathbf{y}^t|X^t)$, *i.e.* the gradient of $\log p(\mathbf{y}^t|X^t)$ with respect to $\mathbf{w}_k$ and T_{ij}. Here i, j, k all range in $\mathcal{Y}$, indexing the classes (possible label of letters). By (1),

$$\log p(\mathbf{y}^t|X^t) = -\log Z_{X^t} + \sum_{s=1}^{m} \langle \mathbf{w}_{y_s^t}, \mathbf{x}_s^t \rangle + \sum_{s=1}^{m-1} T_{y_s^t, y_{s+1}^t}. \tag{5}$$

Step 1. Let us first compute $\nabla_{\mathbf{w}_k} \log p(\mathbf{y}^t|X^t)$, and to this end we start with the gradient of the first term $\log Z_{X^t}$. Denote $[\![x]\!] = 1$ if x is true, and 0 otherwise. Then

$$\nabla_{\mathbf{w}_k} \log Z_{X^t} = \frac{1}{Z_{X^t}} \nabla_{\mathbf{w}_k} Z_{X^t} \tag{6}$$

$$= \frac{1}{Z_{X^t}} \sum_{\mathbf{y}\in\mathcal{Y}^m} \nabla_{\mathbf{w}_k} \exp\left(\sum_{s=1}^{m} \langle \mathbf{w}_{y_s}, \mathbf{x}_s^t \rangle + \sum_{s=1}^{m-1} T_{y_s,y_{s+1}} \right) \tag{7}$$

$$= \frac{1}{Z_{X^t}} \sum_{\mathbf{y}\in\mathcal{Y}^m} \exp\left(\sum_{s=1}^{m} \langle \mathbf{w}_{y_s}, \mathbf{x}_s^t \rangle + \sum_{s=1}^{m-1} T_{y_s,y_{s+1}} \right) \left(\sum_{s=1}^{m} \langle \nabla_{\mathbf{w}_k} \mathbf{w}_{y_s}, \mathbf{x}_s^t \rangle + \nabla_{\mathbf{w}_k} \sum_{s=1}^{m-1} T_{y_s,y_{s+1}} \right) \tag{8}$$

$$= \sum_{\mathbf{y}\in\mathcal{Y}^m} p(\mathbf{y}|X^t) \sum_{s=1}^{m} [\![y_s = k]\!] \mathbf{x}_s^t \tag{9}$$

$$= \sum_{s=1}^{m} p(y_s = k|X^t) \mathbf{x}_s^t. \tag{10}$$

So it requires the marginal distribution of each node y_s given X^t. Note in (10), $p(y_s = k|X^t)$ means y_s is a random variable. We do not write $p(y_s^t = k|X^t)$ since y_s^t is the given label. Of course it *is*

valid to write $p(y_s = y_s^t|X^t)$. The derivative of the second term in (5) with respect to $\mathbf{w}_k$ is simply

$$\nabla_{\mathbf{w}_k} \sum_{s=1}^{m} \langle \mathbf{w}_{y_s^t}, \mathbf{x}_s^t \rangle = \sum_{s=1}^{m} [\![y_s^t = k]\!] \mathbf{x}_s^t. \tag{11}$$

So in total, for a single sequence/word $X^t = (\mathbf{x}_1^t, \ldots, \mathbf{x}_m^t)^\top$ with label $\mathbf{y}^t \in \mathcal{Y}^m$, we have

$$\nabla_{\mathbf{w}_k} - \log p(\mathbf{y}^t|X^t) = \sum_{s=1}^{m} (p(y_s = k|X^t) - [\![y_s^t = k]\!]) \mathbf{x}_s^t, \tag{12}$$

which can be compactly written in a matrix form as

$$G^t := -(\nabla_{\mathbf{w}_1} \log p(\mathbf{y}^t|X^t), \ldots, \nabla_{\mathbf{w}_{26}} \log p(\mathbf{y}^t|X^t)) = (X^t)^\top C^t \in \mathbb{R}^{129 \times 26}, \tag{13}$$

$$\text{where} \quad C^t \in \mathbb{R}^{m \times 26} \quad \text{and} \quad C_{sk} = p(y_s = k|X^t) - [\![y_s^t = k]\!]. \tag{14}$$

We will call C^t the coefficient matrix. If we further stack all training *words* into

$$\mathbf{y}_{\text{train}} := \begin{pmatrix} \mathbf{y}^1 \\ \vdots \\ \mathbf{y}^n \end{pmatrix} \in \mathbb{R}^{mn}, \qquad X_{\text{train}} := \begin{pmatrix} X^1 \\ \vdots \\ X^n \end{pmatrix} \in \mathbb{R}^{mn \times 129}, \qquad C_{\text{train}} := \frac{1}{n} \begin{pmatrix} C^1 \\ \vdots \\ C^n \end{pmatrix} \in \mathbb{R}^{mn \times 26}, \tag{15}$$

then the aggregated average gradient can be written as

$$G_{\text{train}} := \frac{1}{n} \sum_{t=1}^{n} G^t = -\frac{1}{n} \left(\nabla_{\mathbf{w}_1} \sum_{t=1}^{n} \log p(\mathbf{y}^t|X^t), \ldots, \nabla_{\mathbf{w}_{26}} \sum_{t=1}^{n} \log p(\mathbf{y}^t|X^t) \right) = X_{\text{train}}^\top C_{\text{train}}. \tag{16}$$

This will be exactly the formula used in our PETSc implementation, modulo some matrix replication trick that will be detailed in Section 2.1. All we need to do is to compute the C_{train} matrix, where the key quantity is the marginal distributions $p(y_s = k|X^t)$. That requires dynamic programming and the cost $O(m\,|\mathcal{Y}|^2)$ for each word.

Step 2. We next compute $\nabla_{T_{ij}} \log p(\mathbf{y}^t|X^t)$. Running the above derivation analogously, we derive

$$\nabla_{T_{ij}} - \frac{1}{n} \sum_{t=1}^{n} \log p(\mathbf{y}^t|X^t) = \frac{1}{n} \sum_{t=1}^{n} \sum_{s=1}^{m-1} \left\{ p(y_s = i, y_{s+1} = j|X^t) - [\![y_s^t = i \text{ and } y_{s+1}^t = j]\!] \right\}. \tag{17}$$

Our code will compute these quantities explicitly, and the key challenge here is to compute the edge marginals $p(y_s = i, y_{s+1} = j|X^t)$. They can be computed together with the node marginals in Step 1, with the overall computational cost being $O(m\,|\mathcal{Y}|^2)$ for each word.

1.2 IID model

If we treat all letter images as IID (independent and identically distributed) samples, then the model (1) can be simplified by fixing T_{ij} to 0. As a result, the node marginals can be trivially computed by

$$p(y_s = k|X^t) = \exp(\langle \mathbf{w}_k, \mathbf{x}_s^t \rangle) \Big/ \sum_{k'=1}^{26} \exp(\langle \mathbf{w}_{k'}, \mathbf{x}_s^t \rangle). \tag{18}$$

The gradient formulae (for $\mathbf{w}_k$) in (13, 16) remain unchanged. Of course, we do not need the gradient in T_{ij} any more.

2 Parallel Optimization: Implementation Details and Facilities Provided

You do not have to work from scratch. The following routines have been provided in the folder `code_PETSc`, allowing you to focus on optimization. In my own implementation, only **100 lines** of new code were needed to complete the implementation of CRF! And it's C++ code, not Matlab! Of course, you will need to read and digest a lot of code before being able to fill in the blanks. To help you get familiar with the PETSc functions that are important for this project, a demo program `demo_PETSc.cpp` is provided, and we will refer to its functions later. To run the demo with two cores, just type

```
make demo
mpirun -n 2 ./demo_PETSc
```

Currently, the code can run IID training which treats each letter image as an independent training example and learns a 26-class classifier. To run the IID training with 2 cores, copy `train.txt` and `test.txt` to the current directory, and then type

```
make
mpirun -n 2 ./seq_train -data ./train.txt -tdata ./test.txt -lambda 1e-3 -loss IID -tol 1e-3
```

The meaning of the command line arguments are:

- `./train.txt`: the training data filename,
- `./test.txt`: the test data filename,
- `-lambda`: the value of the λ in (4),
- `-tol`: the tolerance for optimization termination,
- `-loss`: must be `IID` or `CRF`.

A sample console printout is given in `sample_out_IID.txt`. To switch to CRF training AFTER completing your implementation, just change the last `IID` into `CRF`. The code currently dumps the following performance metrics to STDOUT at each iteration:

```
iter fn.val gap time feval.num  train_lett_err  train_word_err  test_lett_err  test_word_err
```

They stand for:

1. the current iteration number,
2. the objective value of the current solution,
3. some measure indicating the distance from optimality, *e.g.* square of the gradient norm, duality gap, etc,
4. the number of times that the callback function (of objective and gradient) has been called,
5. letter-wise error rate on the training data (out of 100),
6. word-wise error on the training data,
7. letter-wise error on the test data,
8. word-wise error on the test data.

2.1 Data loading

The data is loaded into the application context structure "`AppCtx user`" in the `main` function (line 39 of `seq_train.cpp`). Go to `appctx.hpp` to peruse the fields and detailed comments on the `SeqCtx` and `AppCtx` structures. The vector `labels` records $\mathbf{y}_{\text{train}}$ (the labels of the letters in the training data), and the matrix `data` *corresponds to* the training data matrix X_{train} (but not exactly; see the matrix replication below). Their definitions follow (15) exactly. Note in C++, array indices start from 0. So the labels (the values in the `labels` vector) are encoded from 0 to 25. We also added a constant 1 to each image feature vector, and therefore each $\mathbf{x}_s^t \in \mathbb{R}^{129}$ (X_{train} has 129 columns).

In parallel computing, it is important to specify the range of rows and columns that belong to each process. We allow PETSc to apply its own heuristics to partition the columns. However we must take control over the row boundaries (letters), because we do not want any word to be split into multiple processes. So we first evenly divided the 6,877 *words* throughout the processes, and then all letters of each word must go to the same process. As a result, when computing the node and edge marginal distributions, each process can process its own set of local words independently.

To record the range of words stored in my process, as well as the number of letters in each word, a structure `SeqCtx` is introduced in `appctx.hpp`. It records word related information, *e.g.* what is the (global) index of the first and last word/letter stored in this process, what is the length of each word (#letters). Detailed comments have been attached to this structure in `appctx.hpp`. Make sure you understand it.

The job of loading the data into the `AppCtx` structure is implemented in `loaddata_libsvm.cpp/hpp`. But **you do NOT need to read any line in it**. Just focus on the code of optimization by directly using the resulting label vector, data matrix, and sequence contexts.

The last important technique we exploited is the *virtual* replication of the data matrix X_{train}, using the PETSc function MatCreateMAIJ. This is highly effective and cool. Recall to compute the model output for $\mathbf{x}_s^t$, we need to compute $\langle \mathbf{x}_s^t, \mathbf{w}_k \rangle$. PETSc allows us to compute all these numbers (all t, s, k) by a *single* matrix-vector multiplication (equivalent to the matrix-matrix multiplication in (16), but differ in implementation). To avoid writing a for-loop with 26 iteration, PETSc allows us creates a new matrix that replicates X_{train} for 26 times:

`MatCreateMAIJ`(X_{train}, 26, X_{rep})

Of course X_{rep} does NOT physically replicate X_{train}; it just takes a note in its data structure, and the repetition is only logical. If we query its size by MatGetSize, it is 26 times of that of X_{train} in both rows and columns. In `loaddata_libsvm.cpp`, we did first create X_{train}, and then call `MatCreateMAIJ`. The local #row and #column of X_{rep} is also 26 times of those of X_{train}. However, X_{rep} is NOT really equal to replicating X_{train} 26 times in both row and column directions (making a 26-by-26 block matrix). The only property we need to understand about X_{rep} is the result when it is multiplied with a vector. Here is the detailed explanation.

In TAO (indeed most existing solvers), the optimization variable must be a vector (not a matrix). So our solution is to collect all weight entries into a long vector ($26 \cdot 129$ dimensional). For some reason that will become clear later, we do NOT just concatenate $\{\mathbf{w}_k\}$ into $(\mathbf{w}_1^\top, \ldots, \mathbf{w}_{26}^\top)^\top$. Instead

it uses the *row-major* representation of the matrix $(\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_{26})$:

$$\mathbf{w} := \begin{pmatrix} w_{1,1} \\ w_{2,1} \\ \vdots \\ w_{26,1} \\ w_{1,2} \\ w_{2,2} \\ \vdots \\ w_{26,2} \\ \vdots \\ w_{1,129} \\ w_{2,129} \\ \vdots \\ w_{26,129} \end{pmatrix} \qquad \text{where} \quad w_{k,d} \text{ is the } d\text{-th entry of } \mathbf{w}_k. \tag{19}$$

Since C++ uses row-major and PETSc is based on C++, this choice is not surprising. Here it can be seen that the index of class (1 to 26) runs first, followed by the index of features (1 to 129). If you really wonder why such an order is adopted, here is a sketch reason. The layout of $\mathbf{w}$ in (19) allows the local range of columns of X_{train} to be directly mapped to the local range of elements in $\mathbf{w}$. Suppose the first 11 columns of X_{train} belong to process 0, and the next 12 columns belong to process 1. Then we can naturally determine the layout of the entries in $\mathbf{w}$ by assigning the first $11 \cdot 26$ entries to process 0, and the next $12 \cdot 26$ entries to process 1. Recall that the vector entries belonging to a process must be **contiguous (cannot jump)**. If we assemble $\{\mathbf{w}_k\}$ into $(\mathbf{w}_1^\top, \ldots, \mathbf{w}_{26}^\top)^\top$, then this natural correspondence of local range will no longer be available.

The most appealing consequence of this design is that the product of X_{rep} and $\mathbf{w}$ is very nice:

$$X_{\text{rep}} \cdot \mathbf{w} \qquad \text{produces} \qquad \begin{pmatrix} \langle \mathbf{x}_1^1, \mathbf{w}_1 \rangle \\ \langle \mathbf{x}_1^1, \mathbf{w}_2 \rangle \\ \vdots \\ \langle \mathbf{x}_1^1, \mathbf{w}_{26} \rangle \\ \langle \mathbf{x}_2^1, \mathbf{w}_1 \rangle \\ \vdots \\ \langle \mathbf{x}_2^1, \mathbf{w}_{26} \rangle \\ \vdots \\ \langle \mathbf{x}_m^1, \mathbf{w}_{26} \rangle \\ \vdots \\ \langle \mathbf{x}_m^n, \mathbf{w}_{26} \rangle \end{pmatrix}, \tag{20}$$

which is exactly the row-major representation of the matrix $X_{\text{train}} \cdot (\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_{26})$.

Take this as a DEFINITION resulting from the logical replication of X_{rep}, and do NOT try to find a matrix representation of X_{rep} that yields this result. If you really wonder why it is defined in such a way, consider once again the local storage/partitioning. In (20), the index of class (1 to

26) again runs first, then the letter index within a word (subscript of $\mathbf{x}$), and finally the word index (superscript of $\mathbf{x}$). This is extremely useful, because the result is automatically consistent with our design principle that the local partitioning is based on dividing the *word set* evenly. So if we put the first 15 words on process 0 (*i.e.* $15m$ rows of X_{train}), then here the first $15 \cdot m \cdot 26$ entries of the resulting product in (20) will reside on process 0. In consequence, when a process accesses its local chunk of the vector via `VecGetArray`, it will directly obtain the $\langle \mathbf{x}_s^t, \mathbf{w}_k \rangle$ for all the words t belonging to itself. This is the key benefit of using `MatCreateMAIJ`, and it saves considerable inter-process communication (imagine otherwise we will have to send the elements in the product to their rightful process). This multiplication is performed in line 73 of `iid_loss.cpp`.

In `appctx.hpp`, the matrix variable `data` in the `AppCtx` structure is already X_{rep}, *i.e.* already replicated logically for 26 times. So is the test data matrix `tdata`.

To see how many words/letter are allocated to my process, check the following line in `iid_loss.cpp`:

```
ierr = MatGetLocalSize(user->data, &m_local, &dim_local);
```

The value of `m_local` is 26 times the number of local letters. The (global) indices of the first and last local words are `user->seq.wBegin` and `user->seq.wEnd`, respectively. These indices start from 0, and `user->seq.wEnd` is indeed one plus the last index, so that the number of local words is `user->seq.wEnd - user->seq.wBegin`, and a for loop can be written as

```
for (i = user->seq.wBegin; i < user->seq.wEnd; i++)
```

The number of letters in each word can be found in `user->seq.wLen[]`, using the *global* index. See more details in `appctx.hpp`.

2.2 Implementation of IID modeling

The implementation of IID modeling has been provided to illustrate the relevant usage of TAO and PETSc. It is recommended to peruse `iid_loss.cpp`.

The multiplication of data matrix with model vector in (20) is done in the following line of `iid_loss.cpp`:

```
ierr = MatMult(user->data, w, user->fx);
```

The resulting vector is stored in the `user->fx` variable. Then it calls the function `loss_coef` to compute $p(y_s = k|X^t)$. As explained above, the local entries of `user->fx` will automatically encompass all the values $\langle \mathbf{x}_s^t, \mathbf{w}_k \rangle$ over $k \in \mathcal{Y}$, $s \in \{1, \ldots, m\}$, and t (word index) over the local words. These are sufficient for the local process to compute the node probabilities $p(y_s^t = k)$. It is furthermore sufficient for CRF to compute the node and edge probabilities, because for each word these quantities can be computed without accessing other words.

The function `loss_coef` computes $p(y_s = k|X^t)$, storing the results in the vector `user->c_node`:

```
ierr = loss_coef(user->fx, user->labels, user->c_node, f, user, &user->seq);
```

More accurately, it only computes $p(y_s = k|X^t)$ for the local letters, and stores the results in the *local memory* of the vector `user->cnode`. Here the t index only covers the local words. Since it

is doing IID modeling, one can simply enumerate over all local letters, ignoring their membership in words. Hence in line 33 of `iid_loss.cpp`, it directly loops over the letter, rather than using a two-level loop (first over the words and then over the letters in each word). For each letter, it grabs 26 entries from `user->fx` (or called `fx_array` inside the `loss_coef` function), computes the probabilities, and then appends the results to `cnode_array`. Suppose the local range of words is from t_1 to t_2. Then upon completion, `cnode_array` contains entries

$$\mathbf{c}_{\text{local}} := \frac{1}{n}\begin{pmatrix} p(y_1 = 1|X^{t_1}) \\ \vdots \\ p(y_1 = 26|X^{t_1}) \\ p(y_2 = 1|X^{t_1}) \\ \vdots \\ p(y_2 = 26|X^{t_1}) \\ \vdots \\ p(y_m = 26|X^{t_1}) \\ \vdots \\ p(y_m = 26|X^{t_2}) \end{pmatrix} - \frac{1}{n}\begin{pmatrix} [\![y_1^{t_1} = 1]\!] \\ \vdots \\ [\![y_1^{t_1} = 26]\!] \\ [\![y_2^{t_1} = 1]\!] \\ \vdots \\ [\![y_2^{t_1} = 26]\!] \\ \vdots \\ [\![y_m^{t_1} = 26]\!] \\ \vdots \\ [\![y_m^{t_2} = 26]\!] \end{pmatrix} \tag{21}$$

This is again nice because the entries are naturally arranged in the order. Finally to compute the gradient according to (13, 16), we only need to compute $X_{\text{rep}}^{\top} * \mathbf{c}$ (line 88 of `iid_loss.cpp`). This is really neat, and the reasoning is left as an exercise (just re-apply the definition in (20)).

For numerical robustness, the following trick is widely used when computing $\log \sum_i \exp(x_i)$ for a given array of real numbers $\{x_i\}$. If we naively compute and store $\exp(x_i)$ as intermediate results, underflow and overflow could often occur. So we resort to computing an equivalent form $M + \log \sum_i \exp(x_i - M)$, where $M := \max_i x_i$. This way, the numbers to be exponentiated are always non-positive (eliminating overflow), and one of them is 0 (hence underflow is not an issue). Similar tricks can be used for computing $\exp(x_1)/\sum_i \exp(x_i)$.

2.3 Suggestions on CRF implementation

Compared with the IID model, the CRF introduces the edge weights T_{ij}. Since we still need to represent the optimization variable as a vector, we have to concatenate the node weights and edge weights. So the new optimization variable is

$$\mathbf{w}_{\text{CRF}} := \begin{pmatrix} \mathbf{w}_{\text{node}} \\ \mathbf{w}_{\text{edge}} \end{pmatrix} \quad \text{where} \quad \mathbf{w}_{\text{node}} \text{ is as in (19)}, \quad \text{and} \quad \mathbf{w}_{\text{edge}} = \begin{pmatrix} T_{1,1} \\ \vdots \\ T_{1,26} \\ T_{2,1} \\ \vdots \\ T_{2,26} \\ \vdots \\ T_{26,26} \end{pmatrix}. \tag{22}$$

Then at each iteration we will first need to extract the $\mathbf{w}$ part for nodes, and then the T part. PETSc does provide tools to extract sub-vectors, but let us do it with a simple hack. Obviously,

$$\mathbf{w}_{\text{node}} := \underbrace{\begin{pmatrix} 1 & & & \dots & \dots \\ & 1 & & \dots & \dots \\ & & \ddots & \dots & \dots \\ & & & 1 \dots & \dots \end{pmatrix}}_{:=M_1} \mathbf{w}_{\text{CRF}} \tag{23}$$

where in M_1 all elements *not* set to 1 take the value 0. The size of M_1 can be easily inferred from the length of $\mathbf{w}_{\text{CRF}}$ and the $\mathbf{w}$ in (19). Similarly, we have

$$\mathbf{w}_{\text{edge}} = \underbrace{\begin{pmatrix} \dots & \dots & 1 & & \\ \dots & \dots & & 1 & \\ & & & & \ddots \\ \dots & \dots & & & \\ \dots & \dots & & & 1 \end{pmatrix}}_{:=M_2} \mathbf{w}_{\text{CRF}} \tag{24}$$

Trivially, we can recover $\mathbf{w}_{\text{CRF}}$ from $\mathbf{w}_{\text{node}}$ and $\mathbf{w}_{\text{edge}}$ by using the transpose of M_1 and M_2:

$$\mathbf{w}_{\text{CRF}} = M_1^\top \mathbf{w}_{\text{node}} + M_2^\top \mathbf{w}_{\text{edge}}. \tag{25}$$

In fact, you may use (25) to compute the *gradient* of $\mathbf{w}_{\text{CRF}}$ by first computing the gradients in $\mathbf{w}_{\text{node}}$ and $\mathbf{w}_{\text{edge}}$ (called `g_node` and `g_edge` in the `AppCtx` structure respectively), and then concatenate them by multiplying with $M_1^\top$ and $M_2^\top$ as in (25). The PETSc function MatMultTranspose can be useful here. In `crf_loss.cpp`, the line

```
ierr = make_sparse_matrix(&user->M1, *w, user->w_node, 0);
```

has allocated the M_1 matrices with the proper size (similarly for M_2), and you can directly use them. Since these are extremely sparse matrices, the multiplications above are indeed very efficient.

Since $\mathbf{w}_{\text{edge}}$ is needed for all processes to compute the node and edge marginal distributions, it needs to be broadcast to all processes. Since $\mathbf{w}_{\text{edge}}$ is a distributed vector, one needs to use the scatter utility in PETSc, so that all elements residing on other processes will be copied to my process, and vice versa. This is readily facilitated by the `VecScatterCreateToAll` function, and the demo code `demo_PETSc.cpp` shows how to use it. The `w_edgeloc` variable in `AppCtx` is a local vector (called sequential by PETSc), storing the *entire* $\mathbf{w}_{\text{edge}}$ locally.

Given $X_{\text{rep}} * \mathbf{w}_{\text{node}}$ (as in IID) and $\mathbf{w}_{\text{edge}}$, we need to compute the node and edge marginal distributions, so that they can be further used to compute he gradients in $\mathbf{w}_{\text{node}}$ and $\mathbf{w}_{\text{edge}}$ by (13, 16) and (17) respectively. The efficient computation here requires dynamic programming, and the code has been provided. See Section 2.4.

Finally, after all processes complete computing the gradients on the edge weights, they need to be summed up as in (17). This can again be done by the scatter, but using the `ADD_VALUES` argument. See again `demo_PETSc.cpp`.

2.4 Inference

The following inference routines have been provided with the package.

1. The MAP inference in (3) has been implemented in the `get_errors` function in `crf_loss.cpp`. In fact, this function does more; it returns (by setting via call-by-reference) the letter-wise and word-wise errors (`lError` and `wError`). It takes three vectors/arrays as input:

- `fx`: the $X_{\text{rep}} * \mathbf{w}$ in (20), and the caller (function `Evaluate`) needs to invoke the multiplication. `get_errors` needs to be called twice, once for the training data, and once for the test data (change X_{rep} to `user->tdata`).
- `labels`: simply `user->labels`.
- `w_edgeloc`: a vector that encodes the entire $\mathbf{w}_{\text{edge}}$. It should be a sequential vector, not distributed. It collects *all* the elements of $\mathbf{w}_{\text{edge}}$, and stores them in a vector in my process. See how it is created in `AllocateWorkSpace` via `VecScatterCreateToAll` (`crf_loss.cpp`).

Both `fx` and `labels` are distributed vectors, and the function `get_errors` extracts their *local* elements by `VecGetArray`.

2. The marginal inference for $p(y_s = k|X^t)$ and $p(y_s = i, y_{s+1} = j|X^t)$ is performed by `loss_coef` in `crf_loss.cpp`. It takes three inputs and produces two outputs. The inputs are `fx`, `labels`, and `w_edgeloc`, which are all identical to those in the above `get_errors` function.

Outputs (set via call-by-reference):

- `c_node`: the $\mathbf{c}_{\text{local}}$ vector in (21), and it has already done the division by n. It is a distributed vector, and `loss_coef` only sets its local elements.
- `g_edgeloc`: a 26^2 dimensional vector that encodes this process' contribution to the gradient of $\mathbf{w}_{\text{edge}}$ (cf. (17) and (22)). Concretely, if the process hosts words from t_1 to t_2, then it gives

$$\frac{1}{n} \sum_{t=t_1}^{t_2} \sum_{s=1}^{m-1} \left\{ p(y_s = i, y_{s+1} = j|X^t) - [\![y_s^t = i \text{ and } y_{s+1}^t = j]\!] \right\}. \tag{26}$$

  Note `g_edgeloc` has already done the division by n. It is a sequential vector (not distributed), and all processes have its own copy. The caller `LossGrad` needs to sum them up to form the true gradient stored in a distributed fashion. This can be done by scatter as explained above.

2.5 Other miscellaneous

A timer class is included to facilitate the measurement of computational cost. The implementation is in `timer.cpp` and `timer.hpp`. You definitely do NOT need to read these two files. Just check how the three timers are used in `seq_train.cpp` (the `wallclock_total` field) and `iid_loss.cpp` (start and stop).

TAO can be terminated in many different ways, and users can set it easily. For example, reaching the maximum number of iterations or function evaluation, falling below some threshold of gradient norm or step size. See `seq_train.cpp` for some sample usage.

In very rare cases, the program crashes with strange runtime errors (e.g. returning `nan`). Then try a clean re-compiling by "`make clean`" followed by "`make`".

3 Parallel Optimization: Tasks for Experiment

First complete the implementation of CRF based on the given framework. **The implementation should be filled into `crf_loss.cpp` and `crf_loss.hpp`**, especially the two functions `LossGrad` and `Evaluate`. New functions and variables can be added there. If you create new files, then the `makefile` will need to be updated.

Then answer the following questions in your report. We will use P1, P2, etc to label the questions for the parallel optimization experiment. T1, T2, etc will be used for the Torch experiment.

Question P1. Run your CRF implementation using the following command

```
mpirun -n 3 ./seq_train -data ./train.txt -tdata ./test.txt -lambda 1e-3 -loss CRF -tol 1e-3
```

Copy and paste to your report i) the first 11 lines (*i.e.* 10 iterations) of the output, and ii) the last line when it converges. For example (extracted from `sample_out_IID.txt`)

```
iter fn.val gap time feval.num train_lett_err train_word_err test_lett_err test_word_err
0 24.5949 4.51757 0.259464 1 92.309174 100.000000 92.220780 100.000000
1 20.3838 4.97532 0.687281 3 71.683428 100.000000 71.745935 100.000000
2 17.6704 3.53674 0.793315 4 67.556737 99.883653 67.318116 99.941844
3 15.7212 2.60525 0.897278 5 55.165106 98.283886 55.141614 98.255307
4 13.858 2.06517 1.1684 6 48.117751 97.207679 48.244141 96.888630
5 11.8215 2.43501 1.40405 7 43.459330 94.560791 43.999542 94.765920
6 10.527 1.11771 1.51042 8 37.968636 91.128563 38.556378 91.596394
7 10.0913 0.820058 1.6166 9 36.219320 89.470622 36.911215 90.375109
8 9.44112 0.820068 1.71968 10 33.210034 86.823735 34.380487 88.107008
9 9.10282 1.64436 1.82459 11 33.129118 86.474695 34.059852 86.827566
10 8.69891 0.656452 1.92768 12 30.763303 83.449680 31.758149 84.210526
:
:
107 6.24397 0.00412999 15.8922 112 20.221169 68.499127 22.944500 73.480663
Optimization converged with status CONVERGED_GTTOL.
```

Note the command line controls termination by `-tol 1e-3`. See line 75 of `seq_train.cpp` for its meaning. In my implementation, the program terminated after 102 iterations (77 seconds).

Question P2. The test error obviously depends on the regularization parameter λ. For each λ value in `{1e-2, 1e-4, 1e-6}`, plot a figure with a single curve where the x-axis is the CPU time elapsed (in seconds), and the y-axis is the objective value of the current solution. One dot for each iteration's output. Again use three cores, and CRF loss. If it takes too long to converge, you may relax the tolerance to *e.g.* `-tol 1e-2`, or set the max number of iteration or function evaluation to some smaller value (line 77 and 79 of `seq_train.cpp`), provided that the trend has leveled off.

Does a larger value of λ allow the optimization to converge faster in CPU time, or a smaller value of λ? Why? Hint: the solver is quasi-Newton (like LBFGS).

Question P3. Pick the λ from the last question that gives the lowest test word-wise error. Plot a figure with two curves: one for the test letter-wise error as time goes by (x-axis is CPU time in seconds), and one for the training letter-wise error. Then plot another figure for word-wise error. Made-up plots are shown in Figure 3 and 4.

What can be observed from these plots? In addition, compare these plots with those in Question P2. Does training/test error keep decreasing as the objective function is being reduced?

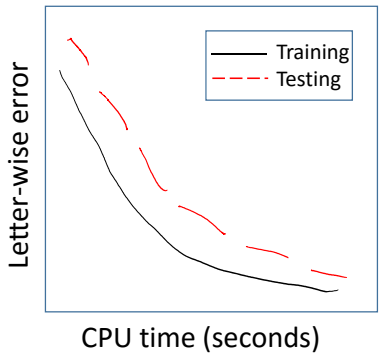


Figure 3. Letter-wise error v.s. CPU time (λ=???)

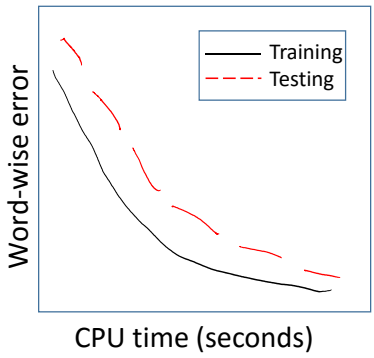


Figure 4. Word-wise error v.s. CPU time (λ=???)

Question P4. Scalability test. In parallel computing, it is standard to study how the speedup depends on the number of cores. Suppose some job costs T_r time if run on r cores. Then the speedup of r cores is defined as T_1/T_r. To rigorously define the "time to converge", one approach is to first set the tolerance to very tight (*e.g.* `-tol 1e-5`) and get a highly accurate estimate of the true minimum objective value. Denote it as $f^* \geq 0$. Then for any given new parameter setting of the solver (the optimization problem itself is kept intact), define the "time to converge" as the time required to find an solution whose objective value falls below $1.01 \cdot f^*$ (or $1.001 \cdot f^*$, etc).

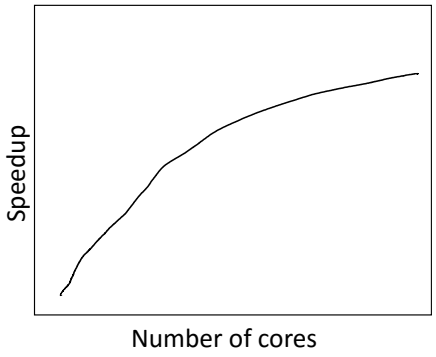


Figure 5. Speedup as a function of #core

Fix `-lambda 1e-4 -loss CRF`. Vary the number of core $r \in \{1, 2, 4, 6, 8\}$ and plot a curve of T_1/T_r as a function of r. A made up figure is given in Figure 5. You may use 8 cores to first get a highly accurate estimate f^* with `-tol 1e-5`. When r is small, it will take longer, and you may relax

the tolerance to "`-tol 1e-3`" provided that the solver does not terminate before finding a solution whose objective value is less than $1.01 \cdot f^*$ (or $1.001 \cdot f^*$ if you feel proper).

Based on the plot, what conclusion can be drawn? If the curve is not linear, explain why.

Question P5. LBFGS (to be exact, the LMVM algorithm in TAO) keeps a number of previous gradients to approximate the inverse of the Hessian. Call this number p (default is 5 in TAO). Although a larger value of p leads to more accurate approximation, it also incurs higher computational cost. So there seems to be a delicate tradeoff. Set #core to 4, and fix λ to the optimal value in Question P2. Vary the value of p in $\{1, 5, 20\}$ ($p = 5$ has already been done in previous questions). To set p to 1, just append the following to the command line in Question P1:

```
-tao_lmm_vectors 1
```

Reproduce the plot in Question P2 for this optimal λ. Still use "`-n 3 -loss CRF -tol 1e-3`". But instead of plotting 3 figures (for the three values of p), plot 3 curves in a single figure. What conclusion can be drawn?

Question P6. Do we really need to store the whole C_{train} matrix in memory (cf. (15)) via storing $\mathbf{c}_{\text{local}}$ over all processes (cf. (21))? If yes, explain why. If no, explain how this can be saved and how the computational performance will be impacted.

4 Torch: Implementation Details and Utilities Provided

You do not have to work from scratch. The following routines have been provided in the folder `code_Torch`, allowing you to focus on optimization. In my implementation, only **40** new lines were added to complete the implementation of CRF for **Question T1** below.

Currently, the code only provides a skeleton of the program, and it cannot be run as is. No complete implementation of IID models is provided, because it really take little effort to enhance it into CRF.

4.1 Data

The following data files are provided in the `data` folder:

`X_torch.dat`, where `X` $\in$ {`train, test, valid`}.

The `train` and `test` files correspond to those in the PETSc part (but in a new format). The file `valid` is a small dataset used for debugging with `checkgrad` (see below).

To load the data, see lines 34-40 in `train_CRF.lua`, where the variable names are self-explanatory. The length of the `wLen` list is equal to the number of words, with `wLen[t]` equal the number of letters in the word `t`. The `data` field is a tricky table, and its usage can be found in the function `updateOutput` of `LinearCRF.lua`.

- `data[i]` returns the `i`-th letter in the whole set.

- `data[i][1]` $\in \{1, \ldots, 26\}$ is the letter's label.
- `data[i][2]` encodes the *sparse* features of the letter `i` as follows.
- `data[i][2][1][k]` $\in \{1, \ldots, 129\}$ is the index of the k-th nonzero feature. k ranges from 1 to 10 if the letter has only 10 nonzero features. Note the bias (id: 129) is already added.
- `data[i][2][2][k]` $\in \mathbb{R}$ is the value of that feature.

4.2 Computation modules

The code provides two new modules: `LinearCRF.lua` and `ClassCRFCriterion.lua`. You may consider the whole model as chaining two "layers":

- Layer 1: `LinearCRF.lua`. Its input is X^t, and its output is a *table* with two components (right, the output can be any data format, not necessarily a tensor or vector):

$$\text{outNode} = X^t \cdot (\mathbf{w}_1, \ldots, \mathbf{w}_{26}) = \{\langle \mathbf{x}_s^t, \mathbf{w}_k \rangle : s \in [1, m], k \in [1, 26]\} \in \mathbb{R}^{m \times 26} \tag{27}$$

$$\text{wEdge} = T \in \mathbb{R}^{26 \times 26} \quad \text{(yes, independent of the input } X^t\text{)}. \tag{28}$$

- Layer 2: `ClassCRFCriterion.lua`. Its input is the above table {`outNode`, `wEdge`}, and its output is $\log p(\mathbf{y}^t | X^t)$ as shown in (5).

Note all the learning parameters $\{\mathbf{w}_1, \ldots, \mathbf{w}_{26}, T\}$ are encapsulated in `LinearCRF.lua`, while no parameter is contained in `ClassCRFCriterion.lua`. To utilize Torch's engine of gradient computation, we only need to instantiate the following routines:

1. `LinearCRF:updateOutput`: Given X^t, compute the output {outNode, wEdge} for the layer of `LinearCRF`. This has been implemented in the provided package. Sparsity in the data X^t has been exploited.
2. `ClassCRFCriterion:updateOutput`: Given {outNode, wEdge}, compute $\log p(\mathbf{y}^t | X^t)$—the output of the `ClassCRFCriterion` layer. This requires marginal inference on a linear chain, which has been implemented in the provided package.
3. `ClassCRFCriterion:updateGradInput` (first step of gradient back-propagation): compute the gradient of $\log p(\mathbf{y}^t | X^t)$ with respect to {outNode, wEdge}. These turn out to be exactly the C^t matrix in (14), along with the gradient of T formula in (17).
4. `LinearCRF:accGradParameters` (second step of gradient back-propagation): compute the gradient of $\log p(\mathbf{y}^t | X^t)$ with respect to $\{\mathbf{w}_1, \ldots, \mathbf{w}_{26}, T\}$ based on the gradient of $\log p(\mathbf{y}^t | X^t)$ with respect to {outNode, wEdge}—the result of `ClassCRFCriterion:updateGradInput`. Indeed, the only computation is the multiplication $G^t = (X^t)^\top C^t$ in (13). The gradient in T is simply copied from the wEdge part of the output of `ClassCRFCriterion:updateGradInput`.

You need to implement the last two routines for gradient computation. In general, the `LinearCRF` should also implement the gradient in its input X^t (to facilitate further back-propagation to its preceding layers). But since this is not needed for our training, we just put a dummy zero there.

The `feval` function in `train_CRF.lua` demonstrates how to put together these two modules in order to compute the objective value and gradient, which are used by the LBFGS solver for training. The MAP inference is implemented in `find_MAP` function, which is in turn used by the `Monitor` function to compute the training (and test) error at each iteration.

4.3 Other miscellaneous

The off-the-shelf `optim.lbfgs` does not print intermediate results. I hacked it and now it can take a monitor to print training/test errors at each iteration. See the line `optimState.monitor = monitor` in `train_CRF.lua`. On cs594.cs.uic.edu, I have recompiled Torch to deploy this hack and you can use it directly. If you use your own machine, please replace `torch/pkg/optim/lbfgs.lua` with the one provided in the project package. Then go to your torch directory and run `./install.sh` to recompile Torch.

The format of the printout should be the same as PETSc. See the bottom of `train_CRF.lua`. Of course, the numbers change and now the gap column looks less meaningful. The implementation of LBFGS in PETSc is highly optimized, and thus it is in general much faster than Torch. The sample output of my implementation is enclosed as `sample_out_Torch.txt` *just for your reference.*

5 Torch: Tasks for Experiment

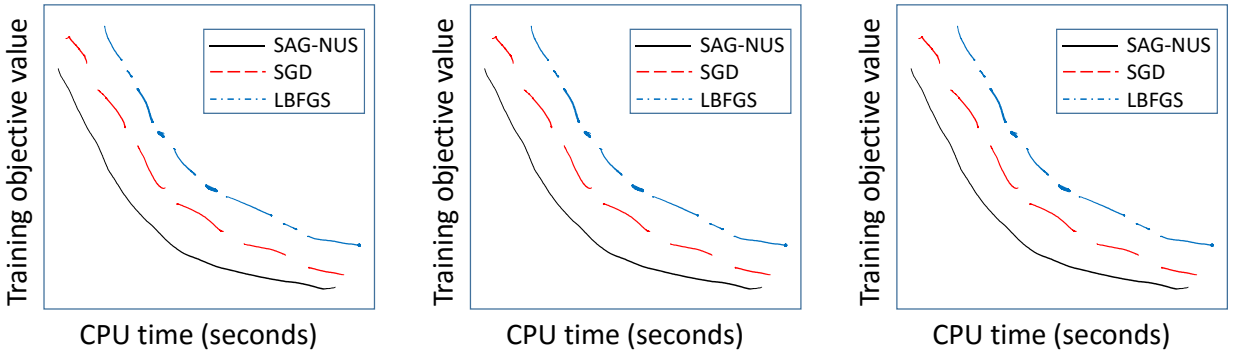
First complete the implementation of CRF in Torch based on the given framework. **Your implementation should be filled into** `ClassCRFCriterion.lua` and `LinearCRF.lua`, especially `ClassCRFCriterion:updateGradInput` and `LinearCRF:accGradParameters`. New functions and variables can be added there. Besides, in `train_CRF.lua`, also add the code of computing the test errors in the function `Monitor`.

Important: you should always verify that your routine for computing the gradient is correct. This is commonly done by comparing your result with the one computed by finite-difference.[1] The `optim.checkgrad` function does exactly this job. I also provided three files in the package to help with this check: `debug_CRFCriterion.lua`, `debug_LinearCRF.lua`, and `debug_CRF.lua`. They are used for testing the gradient of `ClassCRFCriterion`, `LinearCRF`, and their composition (the complete CRF) respectively. Since finite difference is usually expensive in computation, this checking can be done on small datasets, *e.g.* `debug_torch.dat`.

Answer the following questions in your report.

Question T1. Run your implementation with the setting in the first 19 lines of `train_CRF.lua`. That is, use the first 100 words for training. Copy and paste to your report i) the first 11 lines (*i.e.* 10 iterations) of the printout, and ii) the last line when it converges. See `sample_out_Torch.txt` for an example (only 4 lines are shown there, and the last line 49 is dummy).

[1] See https://en.wikipedia.org/wiki/Finite_difference.



(a) $\lambda =$ 1e-2 (b) $\lambda =$ 1e-4 (c) $\lambda =$ 1e-6

Figure 6. Comparison of LBFGS, SGD, and SAG-NUS over $\lambda \in \{$1e-2, 1e-4, 1e-6$\}$.

For the questions below, you may create new files, *e.g.* if you feel it more convenient to collect SGD related code in a separate file.

Question T2. Implement the vanilla SGD algorithm and do not use `optim.sgd`. Also implement the Stochastic Average Gradient method with Non-Uniform Sampling (SAG-NUS) introduced by

Non-Uniform Stochastic Average Gradient Method for Training Conditional Random Fields. M. Schmidt, R. Babanezhad, M.O. Ahmed, A. Defazio, A. Clifton, A. Sarkar. AISTATS, 2015. Their talk and Matlab code (for reference) can be found at `http://www.cs.ubc.ca/~schmidtm/Documents`.

Now for each $\lambda \in$ `{1e-2, 1e-4, 1e-6}`, plot a figure that that compares the three methods (LBFGS, SGD, SAG-NUS) over the training objective value. See Figure 6 for made-up plots. Still use the first 19 lines of `train_CRF.lua` for the problem setting. However, **you may now tune the parameters of the solvers.** This includes, but not limited to, the `optimState` structure in `train_CRF.lua` for LBFGS, as well as the step size rules for SGD and SAG-NUS. Use whatever parameter that you feel is good for this problem, and mention them in the report. What conclusion can be drawn from these figures?

Note it can be expensive if we compute the objective value after *each single* update of SGD/SAG-NUS. So you may compute the value every 1000 updates (or pick a frequency that you feel reasonable). The training set is again the first 100 words, and SGD/SAG-NUS randomly picks one out of them at each step.

Question T3. What is your general experience of tuning the parameters of the solvers in **Question T2**? You do not need to be exhaustive, and just briefly describe what you tried and what you found.

Question T4. The advantage of stochastic optimization lies in the low cost per step. Batch solvers like LBFGS require a lot of computation to process the whole data set, which made it computationally challenging to train on the entire 3438 words. So in the previous questions, we only used the first 100 words for training. Now let us try SGD and SAG-NUS on the whole training set by randomly picking a word out of the **3438 words**.

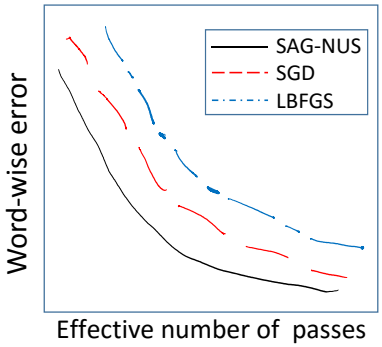


Figure 7. Comparison of word-wise error on the test data for λ =???.

Fix the value of λ to what you used in **Question P3**, whose results were given in Figures 3 and 4. Re-produce Figure 4 (word-wise error) but do *not* include the training error any more. Instead, include the test error of LBFGS, SGD, and SAG-NUS. Now it should look like Figure 7.

The result of LBFGS is just copied from the PETSc results. However, its timing was based on running on the cluster with multiple cores, and is therefore not comparable with that of SGD/SAG-NUS which use Torch on a single core. So we changed the x-axis in Figure 7 to the *effective number of passes*, which is essentially how many passes the solvers have swept over the entire training set. For LBFGS, it is the 5-th number (integer) in the printout shown in **Question P1**. For SGD/SAG-NUS, it is the number of words sampled so far divided by 3438 (#words in the training set). Again for stochastic methods, the error can be computed every, say, 1000 steps.

What conclusions can be drawn from the plot?

Extra credit. You may also plot a similar figure for the test letter-wise error, and that will earn you extra 10 points out of 100.