CS 594: Advanced Machine Learning **Due: 17:00, April 10, 2018**

Project 2: Convex Optimization

Student: *Amlaan Bhoi, Somshubra Majumdar, Ganesh Jagadeesan* *Email:* *abhoi3, smajum6, cjagad2@uic.edu*

In this project you will continue the project on a conditional random field for optical character recognition (OCR), but the emphasis will shift to optimization—parallel and stochastic. For parallel optimization, you are provided with utility code in PETSc and you must stick to the given framework. For stochastic optimization algorithms, you may choose to write in any programming language. However, **unless otherwise specified you cannot use any library beyond the standard ones provided with the language** (same as in project 1).

Only one member of the team needs to submit the work on Blackboard. The deadline of this project is **17:00 on April 10, 2018**. 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 **20%** of your final grade.

You must submit the following TWO files on Blackboard, under `Evaluations/Project_2`:

1. A PDF report named `Report.pdf` which answers 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 named `code.tar` or `code.zip` which includes your source code. Your code should be well commented. If you changed the format of command line for PETSc, 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.

Below we will specify the project in a self-contained fashion. Please read it carefully.

Overview

In this project, we will build upon the CRF classifier that we experimented on in Project 1. However, the focus will be convex optimization algorithms. In particular, we will a) implement a parallel optimization algorithm based on the LBFGS solver provided by PETSc, leveraging its parallel primitives and testing the scalability on UIC's Extreme cluster; b) implement stochastic gradient solvers (SGD and ADAM), and test their efficiency in comparison with LBFGS.

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 PETSc code that loads the data always appends a constant 1 to the feature vectors, leading to 129 features in total. Make sure your own SGD implementation also includes this additional feature. This is different from Project 1.

The 6,877 words are divided evenly into training and test sets, and the package provides two versions of data files under the folder `data/`:

- For PETSc experiments, the files are `train_petsc.txt`, `test_petsc.txt`, and a file that describes the fields: `fields_petsc.txt`.
- For SGD and ADAM experiments, the files are `train_sgd.txt`, `test_sgd.txt`, and a file that describes the fields: `fields_sgd.txt`. Note they are the same as `train.txt`, `test.txt`, and `fields_crf.txt` used in Project 1, and they are just renamed to avoid confusion.

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, $y_1^t = 2$, $y_2^t = 18$, $\ldots$, $y_5^t = 5$.

Figure 1. Example word image

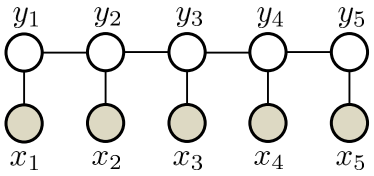


Figure 2. CRF for word-letter

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 $\mathbf{w}_k \in \mathbb{R}^{128}$ 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.* $T_{ij} \neq T_{ji}$, or written as $T' \neq T$ where T' is the transpose of T.

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 $X^* := (\mathbf{x}_1^*, \ldots, \mathbf{x}_m^*)^\top$ via the so-called maximum a-posteriori (MAP) inference:

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

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

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

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

> Here we place the adjustable weight λ on the regularizer, while in Project 1 we put the adjustable weight C on the negative log-likelihood.

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),

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

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

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

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

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

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. S1, S2, etc will be used for the stochastic optimization 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
```

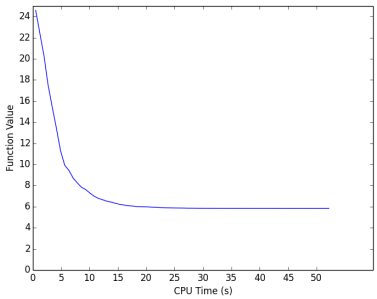
Copy and paste to your report i) the first 11 lines (*i.e.* 10 iterations) of the output, and ii) the last line when it converges. For example (extracted from `sample_out_IID.txt`)

```
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.
```

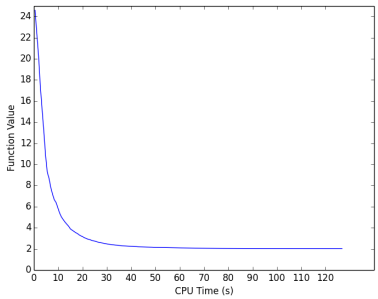
Note the command line controls termination by `-tol 1e-3`. See line 75 of `seq_train.cpp` for its meaning. In my implementation, the program terminated after 95 iterations, with a word-wise test error of 49.52 and objective value 3.3395. If you would like to further confirm the correctness, run your code from Project 1 on this dataset and compare the optimal objective value. Recall that we now use 129 features, so we can't directly copy the result from Project 1. Also note that the LBFGS solver implemented in PETSc/Matlab/Python can be different, so they can be different in many aspects although the optimal objective value should be close.

Answer P1. The first eleven lines (10 iterations) of the output as well as the last line when it converges:

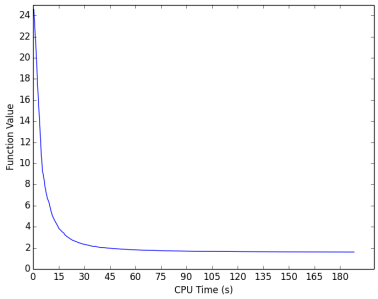
```
iter fn.val gap time feval.num train_lett_err train_word_err test_lett_err test_word_err
0 24.5949 4.55704 0.480855 1 92.309174 100.000000 92.220780 100.000000
1 20.2619 5.09505 1.91933 3 71.379031 100.000000 71.421483 100.000000
2 17.428 3.65693 2.64741 4 66.666667 99.825480 66.382930 99.883687
3 15.3061 2.66886 3.37782 5 53.446615 97.643979 53.500267 97.702821
4 13.2133 2.13058 4.0924 6 45.382037 96.160558 45.644706 95.347485
5 10.8931 2.51846 4.8255 7 39.671714 91.826643 40.426750 92.294272
6 9.33426 1.1719 5.5542 8 33.803414 87.085515 34.647683 87.699913
7 8.75664 0.875296 6.28745 9 31.553192 84.293194 32.326895 85.199186
8 7.89758 0.851361 7.01093 10 28.551613 80.337405 29.361020 81.913347
9 7.2986 1.54779 7.7407 11 27.129812 77.370564 28.128101 78.045944
10 6.79134 0.731165 8.45953 12 24.552075 72.949389 25.608825 74.469322
:
:
114 3.07372 0.00436801 87.6235 120 10.545987 39.383362 14.268265 48.706019
Optimization converged with status CONVERGED_GTTOL.
```



(a) $\lambda =$ 1e-2



(b) $\lambda =$ 1e-4



(c) $\lambda =$ 1e-6

Figure 3. Comparison between $\lambda \in \{$1e-2, 1e-4, 1e-6$\}$.

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).

Answer P2. A larger value of λ constrains the objective function due to its behavior as a regularizer, therefore, as λ increases, the constraints on the possible solutions increases. Due to the use of line search in LBFGS, a more constrained problem requires a fewer number of function evaluations. Therefore, the optimizer converges on the solution faster.

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 4 and 5.

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?

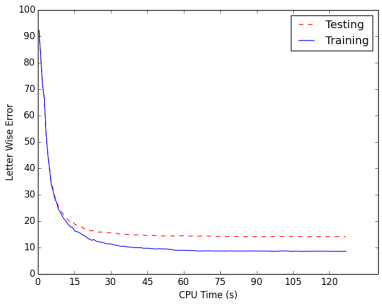


Figure 4. Letter-wise error v.s. CPU time (λ=1e-4)

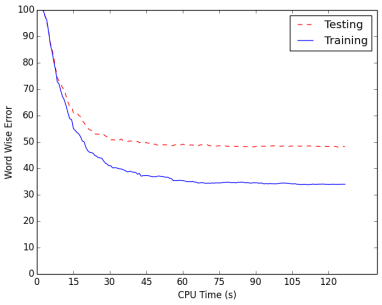


Figure 5. Word-wise error v.s. CPU time (λ=1e-4)

Answer P3. One main observation is that the margin between the training/test Word-wise error is much larger than the margin between the training/test Letter-wise error. When compared to the plots in P2, we can see that as the objective function value decreases, the training/test error keeps decreasing till an extent. The errors do not decrease after a certain point and just hover around a value. The decrease in training/test error and function value is quite correlated to an extent.

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).

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 6. 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.

Answer P4. Based on the plot, one can deduce the curve is not linear. This indicates that the speedup with respect to the *# of cores* and *speedup factor* is **sublinear speedup**. This observation is common and expected. The speedup is NOT linear because of factors such as *overhead* and *contention for resources*. If there were none, we could expect a **linear speedup**. If we have the case:

$$S(p) = \frac{T_s}{T_{par}(p)}$$

where:

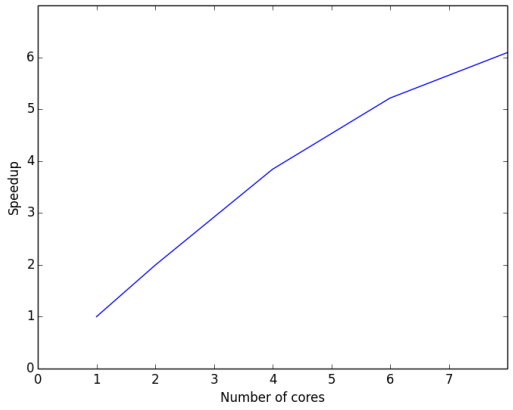


Figure 6. Speedup as a function of #core

T_s = time to converge with 1 core and $T_{par}(p)$ = time to converge with p cores. If $S(p) = p$, then we have linear speedup. Also, the speedup has to obey Amdahl's Law. Thus, this observation is consistent with said theory.

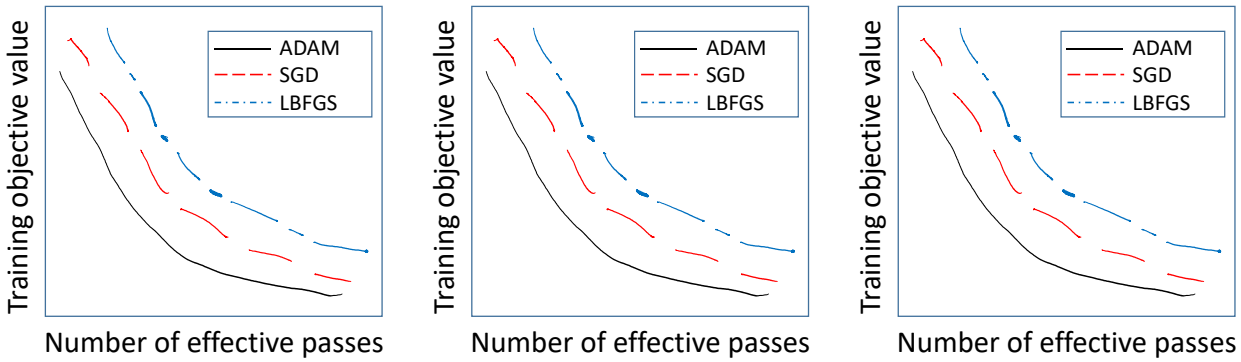
Question P5. Do we really need to store the whole C_{train} matrix in memory (cf. (15)) via storing $\mathbf{c}_{\text{local}}$ over all processes (cf. (21))? If yes, explain why. If no, explain how this can be saved and how the computational performance will be impacted.

Answer P5. The answer is **yes**, we need to store the whole C_{train} matrix in memory (cf. (15)) via storing $\mathbf{c}_{\text{local}}$ over all processes (cf. (21)). This restriction is placed because when calculating $\mathbf{c}_{\text{local}}$, we cannot scatter the values over all workers as we need to sequentially perform the gradient calculation depending on the value from the previous word.

4 Comparison with Stochastic Optimization

We next compare LBFGS (as we used in PETSc) with two stochastic optimization algorithms: SGD and ADAM. You need to implement SGD and ADAM in your favorite languages (probably the one you used in Project 1). **Remember that our input $\mathbf{x}_i$ for each letter is now augmented with a constant 1, hence having 129 features.** Since you have already implemented the gradient computation in Project 1, you can directly use it, with the obvious adaption that SGD/ADAM only needs the gradient on a single word rather than over the entire training set. Don't forget to test the routine again by using `gradest` or equivalent.

The result of LBFGS can be 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/ADAM which use a single core. So instead of comparing in terms of CPU time, we resort to the *effective number of passes*, i.e., 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/ADAM, it is the number of words sampled so far divided by 3438 (#words in the training set).



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

Figure 7. Comparison of LBFGS, SGD, and ADAM over $\lambda \in \{$1e-2, 1e-4, 1e-6$\}$.

For the two stochastic methods, each update (based on a randomly sampled word) is cheap, much cheaper than measuring the performance, e.g., the training objective of CRF in (4), or the two types of error on the test set. So you can choose to compute these measures every 1000 update, or pick a frequency that you feel reasonable.

Answer the following questions in your report.

Question S1. Now for each $\lambda \in$ `{1e-2, 1e-4, 1e-6}`, plot a figure that compares the three methods (LBFGS, SGD, ADAM) over the training objective value. See Figure 7 for made-up plots, where the x-axis uses the effective number of passes. **You may tune the parameters of SGD/ADAM** (but not LBFGS/PETSc), such as step size, exponential decay rate, etc. Use whatever parameter that you feel is good for the solver on this problem, and mention them in the report. What conclusion can be drawn from these figures?

Answer S1.

4.1 Observations from loss plot

We find that from the observed global minimum values of $\{6.38, 2.12, 1.61\}$ for the values of $\lambda =$ {1e-2, 1e-4, 1e-6} respectively, the stochastic optimization algorithms reach the same values, or are close to it.

Due to the use of a cyclic learning rate with Adam + AMSGrad, we observe a more erratic learning curve for that optimizer, which is due the cyclic increase of the learning rate from the triangular waveform schedule.

In addition, we also observe cases where SGD does worse than Adam, especially in the case where $\lambda = 1e-6$. In this case, the global minima is *1.61*, however Adam only approaches *1.73* and SGD only approaches *1.77*. We can observe during the final stages of 150 epochs of training, that the average gradient value is in the magnitude of *1e-8*, and therefore this observation is due to simply insufficient training.

Question S2. What is your general experience of tuning the parameters of the solvers in **Question S1**? You do not need to be exhaustive, and just briefly describe what you tried and what you found.

Answer S2.

S2.1 Stochastic Gradient Descent with Nesterov Momentum

We utilize Nesterov Momentum with Stochastic Gradient Descent to increase the speed of convergence on all of our models, with the default *momentum* hyperparameter value of γ to be *0.9*.

For the learning rate decay schedule, we chose a custom exponential decay schedule, with a *rate* value of 0.99 multiplied by the previous learning rate at the end of each iteration. As the optimization procedure must be performed for large number of iterations for smaller values of λ, we cap the minimum learning rate to be a dynamic variable *min_lr*, whose initial value is determined by observing the loss curve for the first 10 epochs. We compute the objective function's value every epoch, and maintain a history of the past three values. If we find that the loss has not decreased in the past three epochs, we halve *min_lr*. This causes a dynamic reduction of the current learning rate which attempts to use finer learning rate if stagnation occurs.

We opt to break out of the optimization loop if the loss value is lesser than a certain *threshold*, which is determined by observing the loss value over a predetermined number of epochs over the entire dataset.

We observe that SGD + Momentum is very sensitive to the choice of the initial learning rate and the schedule which updates it. More so than any other choice of γ for momentum, or whether we chose to use Nesterov Momentum or not, the key hyperparameter was learning rate. Next to this, choosing an appropriate learning rate schedule was highly important as well. Choosing an exponential decay factor too high caused divergence, and very low factors caused very slow learning.

S2.2 Adam & AMSGrad

We utilize the Adam optimizer with the *AMSGrad* correction from the paper *On the Convergence of Adam and Beyond*. We opt to use the default hyperparameters of Adam, mainly $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1e{-}8$, having searched for different combinations of β_2 to improve convergence speed.

For the learning rate decay schedule, we use a cyclic learning rate schedule, with a linear triangular waveform determining the increase and decrease of the base learning rate. For the cyclic learning rate function, we determine an appropriate value of *stepsize, minimum learning rate and maximum learning rate* that is chosen via careful observation of the loss plot in the first 20 epochs. We still follow the learning rate decay pattern as SGD + Momentum, and also cap the learning rates to a dynamic minimum and maximum value for the cyclic learning rate, using the same methodology as used in SGD.

We opt to break out of the optimization loop if the loss value is lesser than a certain *threshold*, which is determined by observing the loss value over a predetermined number of epochs over the entire dataset.

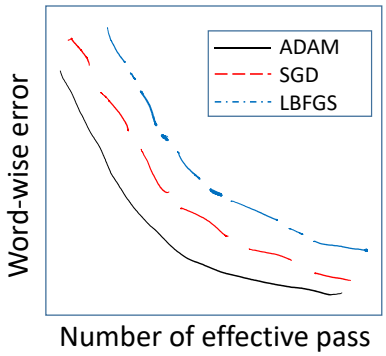


Figure 8. Comparison of word-wise error on the test data for λ =???.

We observed that choosing an appropriate learning rate for Adam was much less of an issue as compared to SGD, and the cyclic learning rate schedule allowed for faster optimization overall compared to SGD in some cases. We attempted to use different values of $\beta_2 = 0.99, 0.999$ and found that even though we use AMSGrad stabilization, there was no marked improvement in the overall convergence speed or value over Adam in the beginning of training. It is only near the end, that AMSGrad's stability is beneficial, whereas Adam rapidly fluctuates as it reaches close to the global minima.

Question S3. Fix the value of λ to what you used in **Question P3**, whose results were given in Figures 4 and 5. Re-produce Figure 5 (word-wise error) but do *not* include the training error any more. Instead, include the word-wise test error of LBFGS, SGD, and ADAM. Now it should look like Figure 8.

What conclusions can be drawn from the plot?

Extra credit [15 pt]. In SGD and ADAM, although considerable computational savings have been achieved by using stochastic gradient based on a single randomly sampled word, the gradient on that word is still computed exactly via message passing. This is fine for linear chain because exact inference is not hard. However, let us just assume that we can only do sampling (e.g., MCMC), and therefore two levels of stochasticity have to be in place: randomly sampled word and approximate inference.

Now add three curves to Figure 8 using ADAM+MCMC, with the number of samples varied from low to medium to high. You need to properly choose the exact number for "low", "medium", and "high" to make the result more illustrative. How should we change the definition of "number of effective pass" to account for the number of samples? For a randomly sampled word with m characters, let us think of the message-passing based exact inference as drawing m samples.