CS 512: Advanced Machine Learning

Lab 1: Graphical Models

Student: Your Name *Email: netid@uic.edu*

Deadline: 4 PM, Feb 23, 2022

This lab is for group work. Unless otherwise specified you cannot use any library beyond the standard ones provided with Python, Numpy, and Scipy. That is, the use of machine learning libraries such as sklearn is prohibited. We provided some utility code.

How to submit. Only one member of each team needs to submit a zip file on Gradescope under Lab_1. The filename should be `Firstname_Lastname.zip`, where both names correspond to the member who submits it.

Inside the zip file, the following contents should be included:

1. A PDF report named `Report_Firstname_Lastname.pdf` with answers to the questions detailed 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 folder named `result` containing **<u>four</u> output result files** (underlined below in this document).
3. A folder named `code` that contains your source code, along with a short `readme.txt` file (placed inside `code/`) that explains how to run it. Your code should be well commented.

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 deadline. Assignment 1 contributes **11%** to your final grade.

Start working on the assignment early because a considerable amount of work will be needed, especially for making the implementation *efficient*. The workload of the programming part is designed for 4 people, and a smaller group size does not warrant any extra credit or reduction in workload.

You will need to use numpy a lot in this lab. Here is a numpy primer: Python Data Science Handbook, covering numpy, Pandas, Matplotlib. You should at least know that `y = xMatrix[0]` is a shallow copy, where `xMatrix` is a 2-D numpy array. Understand how to make a deep copy. The book provides many notebooks for learning. You can create your Jupyter notebook to run on Google Cloud, or locally on your own machine via VS Code and Anaconda (as opposed to directly downloading Python from the official site). Anaconda is the choice for data science. Data used in this lab has been processed for your convenience.

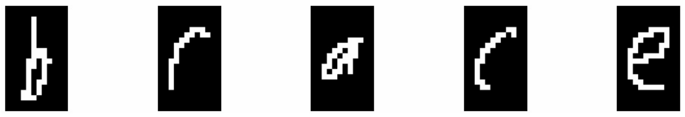


Figure 1. Example word image

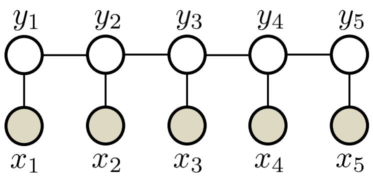


Figure 2. CRF for word-letter

Overview In this part, you will implement a conditional random field for optical character recognition (OCR), with emphasis on inference and performance test.

In particular, 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 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 was maintained by Ben Taskar. 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 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_crf.txt`.

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 $X^t = (\mathbf{x}_1^t, \ldots, \mathbf{x}_m^t)'$, where $'$ means transpose, t is a superscript (not exponent), and each *row* of X^t (*e.g.* $\mathbf{x}_m^t$) represents a letter. Here m is the number of letters in the word, and $\mathbf{x}_j^t$ is a 128 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[1]

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

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

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

- **Node weight:** Letter-wise discriminant weight vector $\mathbf{w}_c \in \mathbb{R}^{128}$ for each possible letter label $c \in \mathcal{Y}$. Note c is different from $\mathbf{y}$, with the former being a letter label in $\mathcal{Y}$, while the latter being a *sequence* of letter labels.

- **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 maximum a-posteriori (MAP) inference:

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

(1a) **[5 Marks]** Show that for any $c \in \mathcal{Y}$, $\nabla_{\mathbf{w}_c} \log p(\mathbf{y}^t|X^t)$—the derivative of $\log p(\mathbf{y}^t|X^t)$ with respect to $\mathbf{w}_c$—can be written as:

$$\nabla_{\mathbf{w}_c} \log p(\mathbf{y}^t|X^t) = \sum_{s=1}^{m} (\llbracket y_s^t = c \rrbracket - p(y_s = c|X^t)) \mathbf{x}_s^t, \tag{4}$$

where $\llbracket \cdot \rrbracket = 1$ if $\cdot$ is true, and 0 otherwise. Here $p(y_s|X^t)$ is the marginal distribution of y_s given X^t (based on (1)). Show your derivation step by step.

Now derive the similar expression for $\nabla_{T_{ij}} \log p(\mathbf{y}^t|X^t)$.

(1b) **[5 Marks]** A feature is a function that depends on X and $\mathbf{y}$, but not $p(\mathbf{y}|X)$. Show that the gradient of $\log Z_X$ with respect to $\mathbf{w}_c$ and T is exactly the expectation of some features with respect to $p(\mathbf{y}|X)$, and what are the features? Include your derivation.

Hint: $T_{y_s, y_{s+1}} = \sum_{p \in \mathcal{Y}, q \in \mathcal{Y}} T_{pq} \cdot \llbracket (y_s, y_{s+1}) = (p, q) \rrbracket$.

[1] In statistics, random variables are generally written as capital letters. However using capital letters in the subscript really looks awkward, *e.g.*, $\mathbf{w}_{Y_s}$, therefore we stick with lower case letter $\mathbf{y}$.

(1c) **[12 Marks]** Implement the decoder (3) with computational cost $O(m|\mathcal{Y}|^2)$. You may use the max-sum algorithm introduced in the course, or any simplified dynamic programming method that is customized to the simple sequence structure (see Appendix 6.2). To keep it simple, you do not need to implement a full-fledged message passing algorithm. It is also fine to use the recursive functionality in the programming language.

Also implement the brute-force solution by enumerating $\mathbf{y} \in \mathcal{Y}^m$, which costs $O(|\mathcal{Y}|^m)$ time. Try small test cases to make sure your implementation of dynamic programming is correct. You may find `Itertools.combinations_with_replacement` useful.

The project package includes a test case stored in `data/decode_input.txt`. It has a single word with 100 letters $(\mathbf{x}_1, \ldots, \mathbf{x}_{100})$, $\mathbf{w}_c$, and T, stored as a column vector in the form of

$$[\mathbf{x}'_1, \ldots, \mathbf{x}'_{100}, \mathbf{w}'_1, \ldots, \mathbf{w}'_{26}, T_{1,1}, T_{1,2}, \ldots, T_{1,26}, T_{2,1}, \ldots, T_{2,26}, \ldots, T_{26,1}, \ldots, T_{26,26}]'. \quad (5)$$

All $\mathbf{x}_s \in \mathbb{R}^{128}$ and $\mathbf{w}_c \in \mathbb{R}^{128}$.

Be careful when loading T. It is NOT a symmetric matrix. Some languages store matrices in a row-major order while some use column-major. So you need to decide whether the loaded T needs to be transposed (I'm not hinting either way).

In your submission, create a folder `result` and store the result of decoding (the optimal $\mathbf{y}^* \in \mathcal{Y}^{100}$ of (3)) in `result/decode_output.txt`. It should have 100 lines, where the i-th line contains one integer in $\{1, \ldots, 26\}$ representing y_i^*. In your report, provide the maximum objective value $\sum_{s=1}^{m} \langle \mathbf{w}_{y_s}, \mathbf{x}_s \rangle + \sum_{s=1}^{m-1} T_{y_s, y_{s+1}}$ for this test case. If you are using your own dynamic programming algorithm (*i.e.* not max-sum), give a brief description especially the formula of recursion.

2 Training Conditional Random Fields

Finally, given a training set $\{X^t, \mathbf{y}^t\}_{t=1}^n$ (n words), we can estimate the parameters $\{\mathbf{w}_c : c \in \mathcal{Y}\}$ and T by maximizing the likelihood of the conditional distribution in (1), or equivalently

$$\min_{\{\mathbf{w}_c\}, T} \; -\frac{C}{n} \sum_{t=1}^{n} \log p(\mathbf{y}^t | X^t) + \frac{1}{2} \sum_{c \in \mathcal{Y}} \|\mathbf{w}_c\|^2 + \frac{1}{2} \sum_{ij} T_{ij}^2. \quad (6)$$

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

(2a) **[12 Marks]** Implement a dynamic programming algorithm to compute $\log p(\mathbf{y}^t | X^t)$ and its gradient. Recall that the gradient is nothing but the expectation of features, and therefore it suffices to compute the marginal distribution of y_j and (y_s, y_{s+1}). See Appendix 6.3. The underlying dynamic programming principle is common to the computation of $\log p(\mathbf{y}^t | X^t)$, its gradient, and the decoder of (3). See Appendix 6.1.

For numerical robustness (overflow or underflow), the following trick[2] is widely used when computing $\log \sum_i \exp(x_i)$ for a given array $\{x_i\}$. If we naively compute and store $\exp(x_i)$ as

[2] https://en.wikipedia.org/wiki/LogSumExp

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 $\exp(x_1)/\sum_i \exp(x_i)$, or its logarithm. Do not use `scipy.special.logsumexp`.

To ensure your implementation is correct, it is recommended that the computed gradient be compared against the result of auto-differentiation (which is based only on the objective value). In Python, use `scipy.optimize.check_grad`. In general, it is a very good practice to use these tools to test the implementation of function evaluator. Since numerical differentiation is often computation intensive, you may want to design small test cases (*e.g.* a single word with 4 letters, 4 randomly valued pixels, and 3 letters in the alphabet). An error level of 10^{-4} will be good enough.

The project package includes a (big) test case in `data/model.txt`. It specifies a value of $\mathbf{w}_c$ and T as a column vector (again $T \neq T'$):

$$[\mathbf{w}_1', \ldots, \mathbf{w}_{26}', T_{1,1}, T_{1,2}, \ldots, T_{1,26}, T_{2,1}, \ldots, T_{2,26}, \ldots, T_{26,1}, \ldots, T_{26,26}]'. \tag{7}$$

Compute the gradient $\frac{1}{n}\sum_{t=1}^{n} \nabla_{\mathbf{w}_c} \log p(\mathbf{y}^t | X^t)$ and $\frac{1}{n}\sum_{t=1}^{n} \nabla_T \log p(\mathbf{y}^t | X^t)$ (*i.e.* averaged over the training set provided in `data/train.txt`) evaluated at this $\mathbf{w}_c$ and T. Store them in <u>`result/gradient.txt`</u> as a column vector following the same order as in (7). Pay good attention to column-major / row-major of your programming language when writing T.

Provide the value of $\frac{1}{n}\sum_{t=1}^{n} \log p(\mathbf{y}^t | X^t)$ for this case in your report.

For your reference, in your instructor's Python implementation, it takes 5 seconds to compute the gradient on the whole training set. Single core of Intel(R) i7-10510U CPU @ 1.80GHz.

(2b) [**12 Marks**] We can now learn $(\{\mathbf{w}_c\}, T)$ by solving the optimization problem in (6) based on the training examples in `data/train.txt`. Set $C = 1000$. Typical off-the-shelf solvers rely on a routine which, given as input a feasible value of the optimization variables $(\mathbf{w}_c, T)$, returns the objective value and gradient evaluated at that $(\mathbf{w}_c, T)$. This routine is now ready from the above task, although you still need to compute the gradient of $\frac{1}{2}\sum_{c \in \mathcal{Y}} \|\mathbf{w}_c\|^2 + \frac{1}{2}\sum_{ij} T_{ij}^2$ in (6).

In this lab, we will use `fmin_tnc` (LBFGS) from `scipy.optimize`, with the input argument `bounds=none`. Set the initial values of $\{\mathbf{w}_c\}$ and T to zero. Here are some examples of its use.

Optimization solvers usually involve a large number of parameters. Some default settings for Python solvers are provided in `code/ref_optimize.py`, where comments are included on the meaning of the parameters and other heuristics. It also includes some pseudo-code of CRF objective/gradient, to be used by various solvers. Feel free to tune the parameters of the solvers if you understand them.

In your submission, include

- The optimal solution $\{\mathbf{w}_c\}$ and T. Store them as <u>`result/solution.txt`</u>, in the format of (7).
- The predicted label for each letter in the test data `data/test.txt`, using the decoder implemented in (1c). Store them in <u>`result/prediction.txt`</u>, with each line having one

integer in $\{1, \ldots, 26\}$ that represents the predicted label of a letter, in the same order as it appears in `data/test.txt`.

In your report, provide the optimal objective value of (6) found by your solver.

3 Benchmarking with Other Learning Models

Now we can perform some benchmarking by comparing with two alternative approaches: multi-class linear SVM on individual letters (SVM-MC), and structured SVM (SVM-Struct). SVM-MC treats each pair of *letter* image and label as a training/test example. We will use the LibLinear package[3], which provides a Python wrapper. In order to keep the comparison fair, we will use linear kernels only (there are kernelized versions of CRF), and for linear kernels LibLinear is much faster than the general-purpose package LibSVM[4],

For SVM-Struct, we will use the off-the-shelf implementation from the SVM$^{\text{hmm}}$ package[5], where some parameters are inherited from the SVM$^{\text{Struct}}$ package[6]. No Matlab/Python wrapper for SVM$^{\text{hmm}}$ is available. So write scripts in your favorite language to call the binary executables and to parse the results.

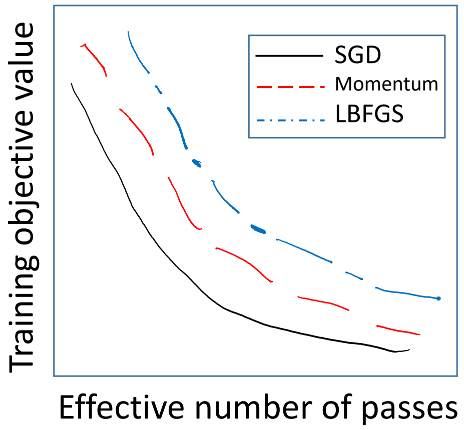
SVM$^{\text{hmm}}$ requires that the input data be stored in a different format. This conversion has been done for you, and the resulting data files are `data/train_struct.txt` and `data/test_struct.txt`. The meaning of the fields in each line is described in `data/fields_struct.txt`.

(3a) [**10 Marks**] SVM$^{\text{hmm}}$ has a number of parameters related to modeling, such as `-c`, `-p`, `-o`, `--t`, and `--e`. Use the default settings for all parameters except `-c`, which serves the same role as C in (6) for CRF. In the sequel, we will also refer to the C in (6) as the `-c` parameter. LibLinear, which is used for SVM-MC, also has this parameter. But note that different from SVM$^{\text{hmm}}$ and (6), the objective function used by LibLinear does NOT divide C by the number of training examples (*i.e.* letters). Keep the default value of other parameters in LibLinear.

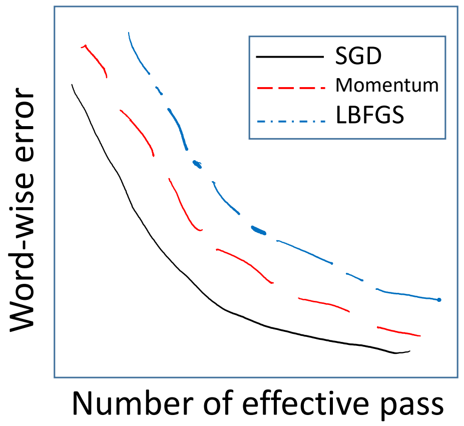
The performance measure can be a) accuracy on letter-wise prediction, *i.e.* the percentage of correctly predicted letters on the whole test set[7], or b) word-wise prediction, *i.e.* the percentage of words whose constituent letters are all predicted correctly. For multi-class problems, accuracy is more commonly used than error.

For each of CRF, SVM-Struct, and SVM-MC, plot a curve in a separate figure where the y-axis is the letter-wise prediction accuracy on test data, and the x-axis is the value of `-c` varied in a range that you find reasonable, *e.g.* $\{1, 10, 100, 1000\}$. Theoretically, a small `-c` value will ignore the training data and generalize poorly on test data. On the other hand, overly large `-c` may lead to overfitting, and make optimization challenging (taking a lot of time to converge).

[3] http://www.csie.ntu.edu.tw/~cjlin/liblinear/
[4] http://www.csie.ntu.edu.tw/~cjlin/libsvm/
[5] http://www.cs.cornell.edu/People/tj/svm_light/svm_hmm.html
[6] http://www.cs.cornell.edu/people/tj/svm_light/svm_struct.html
[7] This is different from computing the percentage of correctly predicted letters in each word, and then averaging over all words, which is the last line of console output of svm_hmm_classify. Both measures, of course, make sense. You may use the letter-wise prediction that svm_hmm_classify writes to the file specified by the third input argument.



(a) Training objective v.s. effective pass



(b) Test word-wise error v.s. effective pass

Figure 3. Comparison of SGD, momentum, and LBFGS

Hint: to roughly find a reasonable range of `-c`, a commonly used heuristic is to try on a small sub-sample of the data, and then apply it to a larger data set (be wary of normalization by the number of training example for LibLinear as mentioned above).

What observation can be made on the result?

(3b) **[5 Marks]** Produce another three plots for word-wise prediction accuracy on test data. What observation can be made on the result?

4 Stochastic Optimization

So far, we have trained CRF (*i.e.*, solving (6)) by batch optimization. We now test stochastic optimization in two senses: stochastic mini-batch and sampling based inference. For this section, you should fix the value of C in the CRF to the best value found by the previous section.

(4a) **[8 Marks]** The idea of stochastic optimization is very simple. At each iteration, we randomly sample B number of training examples (denoted as $\mathcal{B}$), and approximate $\frac{1}{n}\sum_{t=1}^{n}\log p(\mathbf{y}^t|X^t)$ by $\frac{1}{B}\sum_{t\in\mathcal{B}}\log p(\mathbf{y}^t|X^t)$. Using the (stochastic) gradient computed from the latter, we can run multiple solvers such as stochastic gradient descent (SGD) and SGD with momentum. Read the following blog and import its implementation of the two algorithms to your project.

Our goal is to compare the efficiency of SGD, momentum, and LBFGS in driving down the training objective (6) and in reducing the test error. To this end, we will plot two figures, each including three curves corresponding to the three methods (see an example plot in Figure 3). The first figure illustrates the decline of training objective value (*i.e.*, (6)) as a function of effective number of passes. Suppose SGD is run for k iterations/steps, with each iteration sampling B examples/words. Then the effective number of pass is kB/n, facilitating the comparison with LBFGS, where the effective number of pass is exactly the number of objective function evaluation. It is crucial to note that each iteration of LBFGS may evaluate the function **more than once**. `fmin_tnc` takes a callback function, which will help us to track the progress. However, the callback function only takes as input the current solution, and does not provide the count of function evaluation so far. As a hack, you can put a counter in the function evaluation subroutine written by yourself, and make it a global variable which can be accessed inside the callback function.

Within the callback function, evaluate and print two numbers: 1) the word-wise test error given by the current value of $\{w_c\}$ and T; 2) the current training objective value of (6). For the latter, simply (re)compute the objective value instead of buffering the value from the last function evaluation; the mechanism of LBFGS may mess it up. You can also save the values of 1) and 2) as global variables, which can be used for plotting after the solver terminates.

All the three solvers have hyperparameters, including B and learning rate for SGD and momentum. Tune them well for all the methods respectively before plotting. You can tune them based on how fast the training objective decays.

(4b) **[8 Marks]** Since the dynamic programming based gradient computation can still be expensive, an alternative is to evaluate the gradient by sampling, noting that the gradient only requires the marginal distribution on edges and nodes. Here, let us use a block Markov Chain Monte Carlo (MCMC) sampler, which is the method of training restricted Boltzmann machines:

```
1. Ignore the edge potential (i.e., set $T = \mathbf{0}$), which leaves all $y_s$ as independent given $X$.
2. Sample the value of $y_s$ for all $s$ based on such independent distributions.
3. For $k = 1, 2, \ldots, S$ ($S$ being a hyperparameter for tuning, akin to $B$)
4.         Given/fixing $y_1, y_3, y_5, \ldots$, sample for $y_2, y_4, y_6, \ldots$ ($T$ is no longer set to $\mathbf{0}$).
5.         Given/fixing $y_2, y_4, y_6, \ldots$, sample for $y_1, y_3, y_5, \ldots$ ($T$ is no longer set to $\mathbf{0}$).
6.         Call the current $\{y_1, y_2, y_3, y_4, \ldots\}$ as a **sample**.
7. Use the $S$ samples to compute the marginal distribution of nodes and edges.
```

The key convenience is that given $y_1, y_3, y_5, \ldots$, the remaining variables $y_2, y_4, y_6, \ldots$ are independent, hence can be sampled independently and efficiently. Overall we draw S number of samples.

Answer the following questions in your report.

(i) Write out the formula of $p(y_k | \ldots, y_{k-3}, y_{k-1}, y_{k+1}, y_{k+3}, \ldots, X)$. There is no need to distinguish odd or even values of k. What is the overall computational complexity (in big O notation) of drawing a sample, *i.e.*, completing steps 4 and 5 for one iteration of k. Compare it with the cost of dynamic programming. Your complexity should be expressed in the number of letters in the word (m), and the size of the alphabet (*i.e.*, $|\mathcal{Y}| = 26$).

Incidentally, you may have noticed that it is even more time consuming to draw a reasonable number of samples than to run the dynamic programming. This is expected because we are working on a linear chain on which dynamic programming is feasible. But sampling can be applied to a far broader range of graphs, and our focus in this lab is to explore how many samples are needed to get a good gradient.

(ii) Implement the sampling procedure and supply the resulting gradient to SGD, momentum, LBFGS. In a sense, it is doubly stochastic for SGD and momentum, because both the mini-batch and MCMC introduce noise. In contrast, the gradient used by LBFGS is noisy only due to MCMC, and it still uses the entire training set. Now reproduce the two plots in the same way as in Question 4a.

What observations can you make? How does S impact the performance of the three solvers? You can directly reuse the best hyperparameters of the three solvers from Question 4a.

Hint: You do not need to use a large number of samples. $S = 10$ or even 5 might work.

How to debug your sampler? The easiest way is to take an arbitrary word, and compute the node and edge marginal distributions based on samples. Then compare them with the result of dynamic programming. The difference should decay to 0 as S grows.

(4c) [**8 Marks**] Rao-Blackwellization. MCMC can require a lot of samples if we really want high-quality estimates of node and edge distributions. It may also suffer from high variance. To alleviate this problem, the Rao-Blackwell approach is handy. Instead of counting hard samples (0/1), it accumulates fractional samples. For example, previously, when a sample 'a' is drawn for y_k, we just increment the count of y_k ='a' by 1 for the purpose of estimating the marginal probability of y_k. Now suppose $p = (p_a, \ldots, p_z)$ is the conditional probability of y_k given $y_1, \ldots, y_{k-1}, y_{k+1}, \ldots$ (step 4 or 5 in the above algorithm), then we will increment the count of y_k ='a' by p_a, the count of y_k ='b' by p_b, etc.

Similar ideas can be extended to edge marginals. Suppose, when we draw sample for y_k, the current sampled letter value of y_{k-1} and y_{k+1} is 'r' and 's', respectively. Then we will increment the count for $P_{k-1,k}(r, a)$ by p_a, the count for $P_{k-1,k}(r, b)$ by p_b, the count for $P_{k,k+1}(a, s)$ by p_a, the count for $P_{k,k+1}(b, s)$ by p_b, etc. Here $P_{k-1,k}$ is the marginal distribution of (y_{k-1}, y_k), and $P_{k,k+1}$ is the marginal distribution of (y_k, y_{k+1}).

Now do the following in the report.

(i) Write out the pseudo-code for Rao-Blackwellized MCMC sampling. You may doubt when to update the count: after sampling the entire sequence (as in the above algorithm), or right after each y_k is drawn. You may also consider how to do the normalization. Use your own intuition and resort to experiment to figure out a reasonable algorithm. The 'correct' algorithm is not unique, and it works when the divergence against the dynamic programming result decays to 0 as more and more samples are drawn. See the next sub-question (ii).

(ii) Take the model in `data/model.txt` and the first word in `data/train.txt`. Then plot in one figure the KL-divergence between MCMC estimate $\tilde{p}$ and the dynamic programming result p, with or without Rao-Blackwellization. The x-axis is the number of samples, and the y-axis is $\sum_{s=1}^{m} \mathrm{KL}(\tilde{p}_s, p_s)$ (for node marginal) and $\sum_{s=1}^{m-1} \mathrm{KL}(\tilde{p}_{s,s+1}, p_{s,s+1})$ (for edge marginal). Multiplied with using and not using Rao-Blackwellization, there should be **four** curves in the plot. If it helps, use logarithmic scale for some axis. What observation can be made?

5 Robustness to Distortion

An evil machine learner tampered with the training and test images by rotation and translation. However, the labels (letter/word annotation) are still clean, and so one could envisage that the structured models (CRF and SVM-Struct) will be less susceptible to such tempering.

In Python, you can use functions from OpenCV such as `getRotationMatrix2D` and `warpAffine` to rotate and translate an image. They take some parameters such as offset and degree of rotation. Make sure that the image size is not changed. This means in `warpAffine`, set the argument `dsize` appropriately. You can choose any interpolation method. Both functions take images represented as a matrix. So if your image is represented as a 128 dimensional vector, first reshape it into a matrix sized 8×16, then apply these functions, followed by vectorizing it back.

The images stored in the data files are in column-major order. In contrast, Python uses row-major. As you need to translate and rotate images here, you may want to ensure that the 2-D image is properly loaded before applying the transformations. Just check by visualizing the images, e.g. by `imshow`.

In this experiment we randomly select a subset of training examples to distort. All test examples remain unchanged. A randomly generated list of transformations are given in `data/transform.txt`, where the lines are in the following format:

```
r 317 15
t 2149 3 3
```

The first line means: on the 317-th word of the training data (in the order of `train.txt`), apply counterclockwise rotation by 15 degrees *to all its letters*. The second line means on the 2149-th word of the training data, apply translation with offset $(3, 3)$. Note in each line the first number (*i.e.* second column: 317, 2149, ...) is random and *not* sorted. All line numbers appear exactly once.

(5a) [**10 Marks**] In one figure, plot the following two curves where the y-axis is the letter-wise prediction accuracy on test data. We will apply to the training data the first x lines of transformations specified in `data/transform.txt`. x is varied in $\{0, 500, 1000, 1500, 2000\}$ and serves as the value of x-axis.

1) CRF where the `-c` parameter is set to any of the best values found in (3a);

2) SVM-MC where the `-c` parameter is set to any of the best values found in (3a).

What observation can be made on the result?

(5b) [**5 Marks**] Generate another plot for word-wise prediction accuracy on test data. The `-c` parameter in SVM-MC may adopt any of the best values found in (3b). What observation can be made on the result?

6 Appendix: Dynamic programming details

6.1 Computing the partition function

In order to compute Z, let us define $f_1(y_1) = 1$ for all $y_1 \in \mathcal{Y}$, and then for all $i = 2, \ldots, m$

$$f_i(y_i) = \sum_{y_1, \ldots, y_{i-1}} \exp\left(\sum_{j=1}^{i-1} \langle \mathbf{w}_{y_j}, \mathbf{x}_j \rangle + \sum_{j=1}^{i-1} T_{y_j, y_{j+1}} \right), \quad \forall y_i \in \mathcal{Y}. \tag{8}$$

Then

$$Z = \sum_{y_m \in \mathcal{Y}} \exp\left(\langle \mathbf{w}_{y_m}, \mathbf{x}_m \rangle \right) f_m(y_m). \tag{9}$$

To compute $f_i(y_i)$, we use recursion by

$$f_1(y_1) = 1, \quad \forall y_1 \in \mathcal{Y} \tag{10}$$

$$f_i(y_i) = \sum_{y_{i-1}} \exp\left(\left\langle \mathbf{w}_{y_{i-1}}, \mathbf{x}_{i-1} \right\rangle + T_{y_{i-1},y_i}\right) \sum_{y_1,\ldots,y_{i-2}} \exp\left(\sum_{j=1}^{i-2} \left\langle \mathbf{w}_{y_j}, \mathbf{x}_j \right\rangle + \sum_{j=1}^{i-2} T_{y_j,y_{j+1}} \right) \tag{11}$$

$$= \sum_{y_{i-1}} \exp\left(\left\langle \mathbf{w}_{y_{i-1}}, \mathbf{x}_{i-1} \right\rangle + T_{y_{i-1},y_i}\right) \cdot f_{i-1}(y_{i-1}) \qquad (\forall\ y_i \in \mathcal{Y},\ i = 2, 3, \ldots, m). \tag{12}$$

Comment 1: We can also use the log-sum-exp trick if we stay in the logarithmic space. Define $\alpha_i(y_i) = \log f_i(y_i)$. Then $\alpha_1(y_1) = 0$, and

$$\alpha_i(y_i) = \log \sum_{y_{i-1}} \exp\left(\left\langle \mathbf{w}_{y_{i-1}}, \mathbf{x}_{i-1} \right\rangle + T_{y_{i-1},y_i} + \alpha_{i-1}(y_{i-1})\right) \qquad \forall i \geq 2, y_i \in \mathcal{Y} \tag{13}$$

$$\log Z = \log \sum_{y_m} \exp(\langle \mathbf{w}_{y_m}, \mathbf{x}_m \rangle + \alpha_m(y_m)). \tag{14}$$

Comment 2: We might attempt to incorporate the node factor $\exp(\langle \mathbf{w}_{y_i}, \mathbf{x}_i \rangle)$ into the message, i.e.,

$$g_i(y_i) := \exp(\langle \mathbf{w}_{y_i}, \mathbf{x}_i \rangle) \cdot f_i(y_i) = \sum_{y_1,\ldots,y_{i-1}} \exp\left(\sum_{j=1}^{i} \left\langle \mathbf{w}_{y_j}, \mathbf{x}_j \right\rangle + \sum_{j=1}^{i-1} T_{y_j,y_{j+1}} \right), \quad \forall y_i \in \mathcal{Y}. \tag{15}$$

Then the recursion can be written as

$$g_1(y_1) = \exp(\langle \mathbf{w}_{y_1}, \mathbf{x}_1 \rangle), \quad \forall y_1 \in \mathcal{Y} \tag{16}$$

$$g_i(y_i) = \exp(\langle \mathbf{w}_{y_i}, \mathbf{x}_i \rangle) \cdot \sum_{y_{i-1}} \exp\left(T_{y_{i-1},y_i}\right) \sum_{y_1,\ldots,y_{i-2}} \exp\left(\sum_{j=1}^{i-1} \left\langle \mathbf{w}_{y_j}, \mathbf{x}_j \right\rangle + \sum_{j=1}^{i-2} T_{y_j,y_{j+1}} \right) \tag{17}$$

$$= \exp(\langle \mathbf{w}_{y_i}, \mathbf{x}_i \rangle) \cdot \sum_{y_{i-1}} \exp\left(T_{y_{i-1},y_i}\right) g_{i-1}(y_{i-1}) \qquad (\forall\ i \geq 2,\ y_i \in \mathcal{Y}), \tag{18}$$

$$\text{and} \quad Z = \sum_{y_m} f_m(y_m). \tag{19}$$

This works well for a linear chain. But it is not recommended because it does not extend well to general tree structure. Suppose we want to compute the message from j to i, and j has two other neighbors k and k'. Then the messages $k \to j$ and $k' \to j$ have **both** contained the node potential of j (i.e. $f_j(y_j) = \exp(\langle \mathbf{w}_{y_j}, \mathbf{x}_j \rangle)$). So when we multiply together the messages $k \to j$ and $k' \to j$ (as in the message formula), this node potential will be *double counted.*

6.2 MAP inference

Here we define

$$h_i(y_i) = \max_{y_1,\ldots,y_{i-1}} \exp\left(\sum_{j=1}^{i-1} \left\langle \mathbf{w}_{y_j}, \mathbf{x}_j \right\rangle + \sum_{j=1}^{i-1} T_{y_j,y_{j+1}} \right), \quad \forall y_i \in \mathcal{Y}. \tag{20}$$

Then all the recursion expressions in (33) to (35) almost remain unchanged, except that all summations are replaced by max:

$$h_1(y_1) = 1, \quad \forall y_1 \in \mathcal{Y} \tag{21}$$

$$h_i(y_i) = \max_{y_{i-1}} \exp\left(\langle \mathbf{w}_{y_{i-1}}, \mathbf{x}_{i-1}\rangle + T_{y_{i-1},y_i}\right) \max_{y_1,\ldots,y_{i-2}} \exp\left(\sum_{j=1}^{i-2} \langle \mathbf{w}_{y_j}, \mathbf{x}_j\rangle + \sum_{j=1}^{i-2} T_{y_j,y_{j+1}}\right) \tag{22}$$

$$= \max_{y_{i-1}} \exp\left(\langle \mathbf{w}_{y_{i-1}}, \mathbf{x}_{i-1}\rangle + T_{y_{i-1},y_i}\right) h_{i-1}(y_{i-1}) \qquad (\forall\ y_i \in \mathcal{Y},\ i = 2, 3, \ldots, m). \tag{23}$$

After obtaining $h_m(y_m)$, we recover the MAP by backtracking

$$y_m^* = \operatorname*{argmax}_{y_m} \{\exp(\langle \mathbf{w}_{y_m}, \mathbf{x}_m\rangle) h_m(y_m)\} \tag{24}$$

$$y_{i-1}^* = \operatorname*{argmax}_{y_{i-1}} \exp\left(\langle \mathbf{w}_{y_{i-1}}, \mathbf{x}_{i-1}\rangle + T_{y_{i-1},y_i^*}\right) h_{i-1}(y_{i-1}) \qquad (\forall\ i = m, m-1, \ldots, 2). \tag{25}$$

Comment 3: The above algorithm is called max-product. We can also take the log of $h_i(y_i)$ and get the max-sum algorithm:

$$l_i(y_i) = \log h_i(y_i) = \max_{y_1,\cdots,y_{i-1}} \left\{\sum_{j=1}^{i-1} \langle \mathbf{w}_{y_j}, \mathbf{x}_j\rangle + \sum_{j=1}^{i-1} T_{y_j,y_{j+1}}\right\}. \tag{26}$$

And the recursion goes by

$$l_1(y_1) = 0, \quad \forall y_1 \in \mathcal{Y} \tag{27}$$

$$l_i(y_i) = \max_{y_{i-1}} \left\{\langle \mathbf{w}_{y_{i-1}}, \mathbf{x}_{i-1}\rangle + T_{y_{i-1},y_i} + l_{i-1}(y_{i-1})\right\}, \quad \forall y_i \in \mathcal{Y}. \tag{28}$$

And the recovery is

$$y_m^* = \operatorname*{argmax}_{y_m} \{\langle \mathbf{w}_{y_m}, \mathbf{x}_m\rangle + l_m(y_m)\} \tag{29}$$

$$y_{i-1}^* = \operatorname*{argmax}_{y_{i-1}} \left\{\langle \mathbf{w}_{y_{i-1}}, \mathbf{x}_{i-1}\rangle + T_{y_{i-1},y_i^*} + l_{i-1}(y_{i-1})\right\} \qquad (\forall\ i = m, m-1, \ldots, 2). \tag{30}$$

6.3 Marginal distribution

To compute the marginal distribution, we need the backward messages. Define $b_m(y_m) = 1$ for all $y_m \in \mathcal{Y}$ and then for all $i = m-1, \ldots, 1$

$$b_i(y_i) = \sum_{y_{i+1},\ldots,y_m} \exp\left(\sum_{j=i+1}^{m} \langle \mathbf{w}_{y_j}, \mathbf{x}_j\rangle + \sum_{j=i}^{m-1} T_{y_j,y_{j+1}}\right), \quad \forall y_i \in \mathcal{Y}. \tag{31}$$

This is exactly the message $m_{(i+1)\to i}(y_i)$ defined in our lecture. Then the partition function can be computed by

$$Z = \sum_{y_1} \exp\left(\langle \mathbf{w}_{y_1}, \mathbf{x}_1\rangle\right) b_1(y_1). \tag{32}$$

To compute $b_i(y_i)$, we use recursion by

$$
\begin{aligned}
b_m(y_m) &= 1, \quad \forall y_i \in \mathcal{Y} && (33)\\
b_i(y_i) &= \sum_{y_{i+1}} \exp\left(\left\langle \mathbf{w}_{y_{i+1}}, \mathbf{x}_{i+1} \right\rangle + T_{y_i, y_{i+1}}\right) \sum_{y_{i+2},\ldots,y_m} \exp\left(\sum_{j=i+2}^{m} \left\langle \mathbf{w}_{y_j}, \mathbf{x}_j \right\rangle + \sum_{j=i+1}^{m-1} T_{y_j, y_{j+1}}\right) && (34)\\
&= \sum_{y_{i+1}} \exp\left(\left\langle \mathbf{w}_{y_{i+1}}, \mathbf{x}_{i+1} \right\rangle + T_{y_i, y_{i+1}}\right) b_{i+1}(y_{i+1}) \qquad (\forall\ y_i \in \mathcal{Y},\ i = m-1, m-2, \ldots, 1). && (35)
\end{aligned}
$$

Finally, the marginal distribution of y_i is

$$
p(y_i) \propto f_i(y_i) \cdot b_i(y_i) \cdot \exp\left(\left\langle \mathbf{w}_{y_i}, \mathbf{x}_i \right\rangle\right), \tag{36}
$$

followed by local normalization. Furthermore, the marginal distribution of (y_i, y_{i+1}) is

$$
p(y_i, y_{i+1}) \propto f_i(y_i) \cdot b_{i+1}(y_{i+1}) \cdot \exp\left(\left\langle \mathbf{w}_{y_i}, \mathbf{x}_i \right\rangle + \left\langle \mathbf{w}_{y_{i+1}}, \mathbf{x}_{i+1} \right\rangle + T_{y_i, y_{i+1}}\right). \tag{37}
$$

Comment 4: For numerical robustness, we can also turn the backward messages into log space, similar to Comment 1.