COMP4680/8650: Advanced Machine Learning **Due: 23:55, 24 October 2014**

Programming Assignment 3

Student: Your Name(s) *Email: uniID(s)@anu.edu.au*

Logistics

In this project you will implement a conditional random field for optical character recognition (OCR). You are provided with Matlab utility code but may choose to write in the programming language of your choice—but unless otherwise specified you cannot use any libraries beyond the standard one provided with the language.

You can work individually or in teams of two. Only one member of the team needs to submit the work on Wattle. The due date for this project is **23:55 on 24 October, 2014**. You are allowed to resubmit as often as you like and your grade will be based on the last version submitted. Late submissions will be penalized (0.75 for up to one day late, 0.5 for up to two days late). This project is worth 20% of your final grade.

You must submit two files on Wattle: 1) a report with answers to the questions outlined below; 2) a tarball or zip file which includes your source code, and four output result files (underlined below in this document). Your report should include the names and university IDs of team members. Along with your source code you should submit a short readme.txt file that explains how to run it. Your code should also be well commented.

It is better to start working on the project early because a fair amount of effort will be needed to make the implementation *efficient*. This means that in Matlab, you may want to use matrices as much as possible, rather than `for` loops. Or you may push the computation intensive operations into a mex wrapper written in C/C++.

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 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 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`.

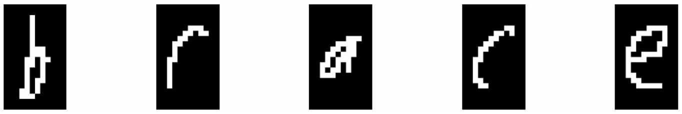

Figure 1. Example word image

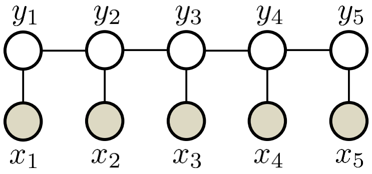

Figure 2. CRF for word-letter

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.

1 Conditional Random Field Modeling

Suppose the training set consists of n words. The image of the i-th word can be represented as $X^i = (\mathbf{x}_1^i, \ldots, \mathbf{x}_m^i)$ (a list), where m is the number of letters in the word, and $\mathbf{x}_j^i$ 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 $\mathbf{y}^i = (y_1^i, \ldots, y_m^i)$, where $y_j^i \in \mathcal{Y} := \{1, 2, \ldots, 26\}$ represents the label of the j-th letter. So in Figure 1, $y_1^i = 2$, $y_2^i = 18$, $\ldots$, $y_5^i = 5$.

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 $\mathbf{y} = (y_1, \ldots, y_m)$ given its image $X = (\mathbf{x}_1, \ldots, \mathbf{x}_m)$ can be written as

$$p(\mathbf{y}|X) = \exp\left(\sum_{j=1}^{m} \langle \mathbf{w}_{y_j}, \mathbf{x}_j \rangle + \sum_{j=1}^{m-1} T_{y_j, y_{j+1}}\right) \Bigg/ \sum_{\hat{\mathbf{y}} \in \mathcal{Y}^m} \exp\left(\sum_{j=1}^{m} \langle \mathbf{w}_{\hat{y}_j}, \mathbf{x}_j \rangle + \sum_{j=1}^{m-1} T_{\hat{y}_j, \hat{y}_{j+1}}\right). \quad (1)$$

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

- Letter-wise discriminant weight vector $\mathbf{w}_y \in \mathbb{R}^{128}$ for each possible label $y \in \mathcal{Y}$;
- 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 $T_{ij} \neq T_{ji}$.

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 $X^* := (\mathbf{x}_1^*, \ldots, \mathbf{x}_m^*)$ via:

$$\mathbf{y}^* = \underset{\mathbf{y} \in \mathcal{Y}^m}{\operatorname{argmax}}\, p(\mathbf{y}|X^*) = \underset{\mathbf{y} \in \mathcal{Y}^m}{\operatorname{argmax}} \left\{ \sum_{j=1}^{m} \langle \mathbf{w}_{y_j}, \mathbf{x}_j^* \rangle + \sum_{j=1}^{m-1} T_{y_j, y_{j+1}} \right\}. \quad (2)$$

(1a) **[5 Marks]** Write out the formulae of $\nabla_{\mathbf{w}_y} \log p(\mathbf{y}|X)$ and $\nabla_{T_{ij}} \log p(\mathbf{y}|X)$, *i.e.* the gradient of $\log p(\mathbf{y}|X)$ with respect to $\mathbf{w}_y$ and T_{ij}. Include your derivation.

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

Hint: $T_{y_j,y_{j+1}} = \sum_{p\in\mathcal{Y},q\in\mathcal{Y}} T_{pq} \cdot [\![(y_j, y_{j+1}) = (p,q)]\!]$, where $[\![\cdot]\!] = 1$ if $\cdot$ is true, and 0 otherwise.

(1c) **[20 Marks]** Implement the decoder (2) with computational cost $O(m|\mathcal{Y}|)$. 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. 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.

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

$$[\mathbf{x}_1', \ldots, \mathbf{x}_{100}', \mathbf{w}_1', \ldots, \mathbf{w}_{26}', T_{1,1}, T_{2,1}, \ldots, T_{26,1}, T_{1,2}, \ldots, T_{26,2}, \ldots, T_{1,26}, \ldots, T_{26,26}]'. \quad (3)$$

All $\mathbf{x}_i \in \mathbb{R}^{128}$ and $\mathbf{w}_j \in \mathbb{R}^{128}$. In your submission, create a folder `result` and store the result of decoding (the optimal $\mathbf{y}^* \in \mathcal{Y}^{100}$ of (2)) 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 $\sum_{j=1}^{m} \langle \mathbf{w}_{y_j}, \mathbf{x}_j \rangle + \sum_{j=1}^{m-1} T_{y_j,y_{j+1}}$ 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

Given a training set $\{X^i, \mathbf{y}^i\}_{i=1}^n$, the parameters $\{\mathbf{w}_y : y \in \mathcal{Y}\}$ and T can be estimated by maximum a posteriori over the conditional distribution in (1), or equivalently

$$\min_{\{\mathbf{w}_y\},T} \; -\frac{C}{n} \sum_{i=1}^{n} \log p(\mathbf{y}^i|X^i) + \frac{1}{2} \sum_{y\in\mathcal{Y}} \|\mathbf{w}_y\|^2 + \frac{1}{2} \sum_{ij} T_{ij}^2. \quad (4)$$

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

(2a) **[20 Marks]** Implement a dynamic programming algorithm to compute $\log p(\mathbf{y}^i|X^i)$ 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_j, y_{j+1}). The underlying dynamic programming principle is common to the computation of $\log p(\mathbf{y}^i|X^i)$, its gradient, and the decoder of (2).

For numerical robustness, the following trick 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 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.

To make sure your implementation is correct, it is recommended that the computed gradient be compared against the result of numerical differentiation (which is based on the objective value only). In Python, use `scipy.optimize.check_grad`. In Matlab, use `gradest.m` from the DERIVESTsuite[1]. 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).

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

$$[\mathbf{w}_1', \ldots, \mathbf{w}_{26}', T_{1,1}, T_{2,1}, \ldots, T_{26,1}, T_{1,2}, \ldots, T_{26,2}, \ldots, T_{1,26}, \ldots, T_{26,26}]'. \tag{5}$$

Compute the gradient $\frac{1}{n}\sum_{i=1}^{n} \nabla_{\mathbf{w}_y} \log p(\mathbf{y}^i|X^i)$ and $\frac{1}{n}\sum_{i=1}^{n} \nabla_T \log p(\mathbf{y}^i|X^i)$ (*i.e.* averaged over the training set) evaluated at this $\mathbf{w}_y$ and T. Store them in `result/gradient.txt` as a column vector following the same order as in (5). Provide the value of $\frac{1}{n}\sum_{i=1}^{n} \log p(\mathbf{y}^i|X^i)$ for this case in your report.

For your reference, in your lecturer's Matlab implementation (65 lines), it takes 5 seconds to compute the gradient on the whole training set. Single core.

(2b) **[20 Marks]** We can now learn $(\mathbf{w}_y, T)$ by solving the optimization problem in (4) 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 $(\mathbf{w}_y, T)$, returns the objective value and gradient evaluated at that $(\mathbf{w}_y, T)$. This routine is now ready from the above task.

In Matlab, you can use `fminunc` from the optimization toolbox. In Python, you can use `fmin_l_bfgs_b`, `fmin_bfgs`, or `fmin_ncg` from `scipy.optimize`. Although `fmin_l_bfgs_b` is for constrained optimization while (4) has no constraint, one only needs to set the bound to $(-\inf, \inf)$. Set the initial values of $\mathbf{w}_y$ and T to zero.

Optimization solvers usually involve a large number of parameters. Some default settings for Matlab solvers are provided in `code/ref_optimize.m`, 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. Read it even if you do not use Matlab, because similar settings might be used in Python. Feel free to tune the parameters of the solvers if you understand them.

In your submission, include

- The optimal solution $\mathbf{w}_y$ and T. Store them as `result/solution.txt`, in the format of (5).
- The predicted label of each letter in the test data `data/test.txt`, using the decoder implemented in (1c). Store them in `result/prediction.txt`, 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 (4) found by your solver.

[1] http://www.mathworks.com.au/matlabcentral/fileexchange/13490-adaptive-robust-numerical-differentiation

3 Benchmarking with Other Methods

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[2], which provides both Matlab and Python wrappers. 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[3],

For SVM-Struct, we will use the off-the-shelf implementation from the SVM$^{\text{hmm}}$ package[4], where some parameters are inherited from the SVM$^{\text{Struct}}$ package[5]. 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, and the resulting data files are `data/train_struct.txt` and `data/test_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 (4) for CRF. In the sequel, we will also refer to the C in (4) as the `-c` parameter. LibLinear, which is used for SVM-MC, also has this parameter. But note that different from SVM$^{\text{hmm}}$ and (4), 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[6], 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).

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?

[2] http://www.csie.ntu.edu.tw/∼cjlin/liblinear/

[3] http://www.csie.ntu.edu.tw/∼cjlin/libsvm/

[4] http://www.cs.cornell.edu/People/tj/svm_light/svm_hmm.html

[5] http://www.cs.cornell.edu/people/tj/svm_light/svm_struct.html

[6] 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.

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

4 Robustness to Tampering

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.

Matlab scripts have been provided that implement rotation and translation. See `rotation.m` and `translation.m` under the `code` folder. They both take some parameters (*e.g.* degree of rotation), which are documented in the `.m` files. They are quite straightforward and can be implemented in Python in a similar way (see comments in the .m files). They both take images represented as a matrix. So if your image is represented as a 128 dimensional vector, first reshape it by `reshape(x, 8, 16)`, then apply these functions, followed by vectorizing it back.

In this experiment we randomly select a subset of training examples to temper with. All test examples remain unchanged. The proposed transformations are listed in `data/transform.txt`, where the lines are like the following:

```
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.

(4a) **[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 constitutes 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?

(4b) **[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?