PGM Programming Assignment: CRF Learning for OCR

1 Overview

Earlier in this course, you learned how to construct a Markov network to perform the task of optical character recognition (OCR). The values in each factor, however, were provided for you. This assignment returns to the same OCR task, but this time you will write the code to learn the factor values automatically from data.

This assignment is divided into two parts. In the first part, we use a small (but real) parameter learning task to introduce a handful of key concepts applicable to nearly any machine learning problem:

- Stochastic gradient descent for parameter learning
- Training, validation, and test data
- Regularization

In the second part of the assignment, you will apply these techniques to the OCR task. In particular, you will construct a Markov network similar to the one from before and use stochastic gradient descent to learn the parameters of the network from a set of training data.

Be sure to get a head start on this assignment! The first part introduces a set of new concepts, and the second is rather intricate. Good luck!

2 Machine Learning Primer

2.1 Stochastic Gradient Descent For Logistic Regression[1]

Consider the following task: given a dataset of images containing either the letter p or q and the corresponding labels for the images, we wish to build a classifier that will allow us to predict the labels of new unseen images as accurately as possible. More concretely, we have a dataset in which each data sample consists of 129 features (16×8 pixels + 1 bias term), $\mathbf{X} = X_1, \ldots, X_{129}$, derived from the images, and the corresponding labels Y for the image (0 if it is an image of a q, 1 if it is an image of a p).

We will learn a model of the conditional probability $P(Y|\mathbf{X})$ using maximum likelihood estimation, where we use a logistic function for the conditional probability, $P(Y = 1|\mathbf{X}) = \frac{1}{1+\exp(-\theta^T \mathbf{X})}$; the logistic model can be considered to be a simple conditional random field. Given a set of M training examples, $\mathcal{D} = \{(\mathbf{x}[m], y[m])\}_{m=1}^{M}$, we want to find the θ^* that maximizes the likelihood of the observed data:

$$\theta^* = \underset{\theta}{\operatorname{argmax}}\, L(\theta : \mathcal{D}) = \underset{\theta}{\operatorname{argmax}} \prod_{m=1}^{M} P(y[m]|\mathbf{x}[m]; \theta)$$

[1] In this section, we have deliberately glossed over many details as these are out of the scope of the class. If you are interested in a more complete treatment of the material in this section, you may find it helpful to review the lecture notes for Stanford classes CS229 (Machine Learning) or EE364a (Convex Optimization I). Their course websites are online at http://cs229.stanford.edu and http://ee364a.stanford.edu respectively.

Equivalently, we can minimize the negative log likelihood:

$$\begin{aligned}\theta^* &= \operatorname*{argmin}_\theta -l(\theta : \mathcal{D}) \\ &= \operatorname*{argmin}_\theta \sum_{m=1}^{M} -\log P(y[m]|\mathbf{x}[m];\theta)\end{aligned}$$

We will use stochastic gradient descent to find the parameter values θ^*. However, we will first revisit the regular (batch) gradient descent algorithm before describing the stochastic variant.

The key intuition behind the gradient descent algorithm is that by taking steps in the direction of the negative gradient of $-l(\theta : \mathcal{D})$, which is the direction of steepest descent, we will eventually converge to a minimizer of $-l(\theta : \mathcal{D})$. This intuition is formalized in the update rule for the parameter vector θ that is used on each iteration of the algorithm

$$\theta := \theta - \alpha \nabla_\theta [-l(\theta : \mathcal{D})]$$

where α is a parameter which determines the size of the steps we will take in parameter space (also known as the learning rate).

Given this update rule, the gradient descent algorithm takes an initial assignment to the parameters θ_0, and repeatedly uses the update rule described above to compute new values for θ, until we have converged (for instance, when the gradient is sufficiently small). Notice that this algorithm requires summing up the gradients over every single training example, which is why it is known as the batch gradient descent algorithm.

In the case where computing the gradient over a single example is computationally expensive (for instance, in CRF parameter learning), the batch algorithm can take very long to converge, as it requires the gradients over all training examples to be computed before the algorithm can perform an update. In such situations, an algorithm that approximates the gradient over the entire training set by the gradient over a single (or a few) training examples, can be a far more efficient alternative; such an algorithm is known as a stochastic gradient descent algorithm.

One disadvantage of using such a coarse approximation of the gradient, however, is that while the algorithm gets θ close to the minimum more quickly, the parameter vector may now oscillate about the true minimum. We therefore make one further tweak to the algorithm to help mitigate this behavior, by slowly decrease the learning rate α to zero as the algorithm progresses. The pseudocode of the stochastic gradient descent algorithm that we will use in this assignment is as follows:

`for k = 1 to max_iterations` [2]:
  Pick an arbitrary training example $(\mathbf{x}[m], y[m])$, then update
  $\theta := \theta - \alpha_k \nabla_\theta [-\log P(y[m]|\mathbf{x}[m];\theta)]$
where $\alpha_k = \frac{0.1}{1+\sqrt{k}}$.

You should now implement the stochastic gradient descent algorithm:

- `StochasticGradientDescent.m` (10 Points) This function runs stochastic gradient descent for the specified number of iterations and returns the value of θ upon termination. It takes in three inputs:

  1. a *function handle* for a function that computes the gradient of the negative log-likelihood of the data, at a particular θ and for a particular iteration number k. **Note:** this function will return the gradient for an arbitrary training example, so *you do not have to explicitly pick the training example in your code for* `StochasticGradientDescent`.

[2] We will use max_iterations=5000 in this part of the assignment.

2. an initial value for θ.
3. the number of iterations to run.

To test your implementation, we have provided you with `LRTrainSGD.m`. Sample data can be found in `Train1X.mat` and `Train1Y.mat` which contain, respectively, the features and ground truth labels for the training instances, in the right format for plugging into `LRTrainSGD.m`. Go ahead and run `LRTrainSGD.m` on the training data with $\lambda = 0$ to get the maximum likelihood estimates of the parameters. Then use `LRPredict.m` to get the predictions for the training data, and `LRAccuracy.m` to calculate the classification accuracy (where 1 = perfect accuracy) on the training data. You should obtain an accuracy of 0.96 on the training data.

2.2 Overfitting & Regularization

Now try your model out on the provided test data in `Test1X.mat` and `Test1Y.mat`. Use `LRPredict.m` and `LRAccuracy.m` to assess the performance of your model on the test set. Unfortunately, you will see that it's somewhat less than the training set accuracy, which suggests that the model has overfit the training data. This problem of overfitting can be addressed using regularization. Concretely, we will add a new term to the negative log likelihood that penalizes us for having large values for θ :

$$\theta^* = \underset{\theta}{\operatorname{argmin}} - l(\theta; \mathcal{D}) + \tfrac{\lambda}{2} \textstyle\sum_{i=1}^{n} \theta_i^2$$

This specific form of regularization is called L_2-regularization, since we are penalizing the square of the parameters (which is the square of their L_2-norm). The code we have provided you in `LRTrainSGD.m` already does this. But what value of λ should we pick?

One way to search for a good value of λ is by using a validation set. We can train our model using the training set using various values of λ and then evaluate the models' accuracies on the validation set. Then, we can pick the value of λ that yields the best accuracy on the validation set. Note that we **do not** use the test data to pick the value for λ, as we wish to use the test data to obtain an estimate for the generalization performance of our model – that is, how well our model will perform on unseen data. If we were to use the test data to pick λ, our model will be fit to this data, and the model's performance on this data will no longer be representative of its performance on unseen data.

You will now implement a function that searches for a good value for λ:

- `LRSearchLambdaSGD.m` (10 Points) This function takes as inputs:

  1. A matrix of features for the training data
  2. A vector of ground truth labels for the training data
  3. A matrix of features for the validation data
  4. A vector of ground truth labels for the validation data
  5. A vector of candidate values for the regularization parameter λ.

  For each value of λ the function should fit a logistic regression model to the training data using `LRTrainSGD.m` and then evaluate the accuracy of the resulting model on the validation data. The output should be a vector of the accuracy in the validation set for each λ.

Test your function with `Train1X.mat`, `Train1Y.mat`, `Validation1X.mat` and `Validation1Y.mat`, as well as the vector of candidate lambdas in `Part1Lambdas.mat`. Expected outputs from your function can be found in `ValidationAccuracy.mat`. You should see that a little bit of regularization helps performance in the test data!

3 Learning CRF Parameters For OCR

3.1 Introduction

Now that you have the basic tools for parameter learning, gradient descent and regularization, we can apply those tools to more complex models than logistic regression. For the remainder of this assignment, you will implement the necessary functions to learn the parameters of a conditional random field for optical character recognition (OCR).

In the CRF model, there are two kinds of variables: the hidden variables that we want to model and those that are always observed. In the case of OCR, we want to model the character assignments (such as 'a' or 'c'), and we always observe the character images, which are arrays of pixel values. Typically, the unobserved variables are denoted by Y and the observed variables are denoted by X. The CRF seeks to model $P(Y|X)$, the conditional distribution over character assignments given the observed images. We will be working with the OCR model from programming assignment 3 that includes only the singleton and pairwise factors. The structure of the model is shown below:

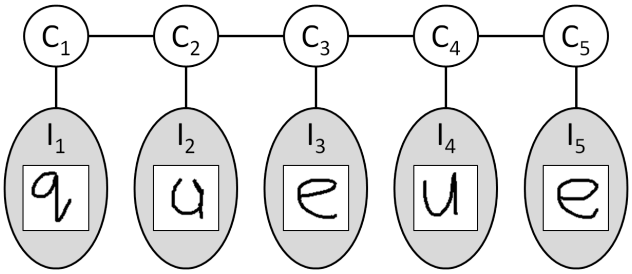

Figure 1: CRF with singleton and pairwise factors.

3.2 The Log-Linear Representation

Up until now, the graphical models you have constructed have focused on the table factor representation. In this assignment, you will instead build the model around log-linear features. A feature is just a function $f_i(\mathbf{D}_i) : \text{Val}(\mathbf{D}_i) \to \mathbb{R}$, where $\mathbf{D}_i$ is the set of variables in the scope of the ith feature; in this assignment, all features are binary indicator features (taking on a value of either 0 or 1). Each feature has an associated weight θ_i. Given the features $\{f_i\}_{i=1}^k$ and the weights $\{\theta_i\}_{i=1}^k$, the distribution is defined as:

$$P(\mathbf{Y}|\mathbf{x} : \theta) = \frac{1}{Z_{\mathbf{x}}(\theta)} \exp\left\{\sum_{i=1}^{k} \theta_i f_i(\mathbf{D}_i)\right\}.$$

The term $Z_{\mathbf{x}}(\theta)$ is the partition function:

$$Z_{\mathbf{x}}(\theta) \equiv \sum_{\mathbf{Y}} \exp\left\{\sum_{i=1}^{k} \theta_i f_i(\mathbf{D}_i)\right\}.$$

Note that computing $Z_{\mathbf{x}}(\theta)$ requires summing over an exponential number of assignments to the variables, so we won't be able to use a simple brute-force technique to compute $P(\mathbf{Y}|\mathbf{x} : \theta)$ even though the feature values can be computed efficiently.

In our CRF, we have three types of features:

- $f^C_{i,c}(Y_i)$, which operates on single characters / hidden states (an indicator for $Y_i = c$).
- $f^I_{i,j,c,d}(Y_i, x_{ij})$, which operates on a single character / hidden state and an image pixel associated with that state (an indicator for $Y_i = c$, $x_{ij} = d$). These are collectively used to encode the individual probability that $Y_i = c$ given $\mathbf{x}_i$.
- $f^P_{i,c,d}(Y_i, Y_{i+1})$ which operates on a pair of adjacent characters / hidden states (an indicator for $Y_i = c$, $Y_{i+1} = d$).

3.3 Learning CRF Parameters: Theory

You will learn the parameters of the CRF in the same way you did for logistic regression in Section 2. You need to write a function that takes a single data instance and computes the "cost" of that instance and the gradient of the parameters with respect to the cost. Given such a function, you can then use stochastic gradient descent to optimize the parameter values.

Recall that our goal is to maximize the log-likelihood of the parameters given the data. Thus the "cost" we minimize is simply the negative log-likelihood, together with a L_2-regularization penalty on the parameter values to prevent overfitting. The function we seek to minimize is:

$$\text{nll}(\mathbf{x}, \mathbf{Y}, \theta) \equiv \log(Z_{\mathbf{x}}(\theta)) - \sum_{i=1}^{k} \theta_i f_i(\mathbf{Y}, \mathbf{x}) + \frac{\lambda}{2} \sum_{i=1}^{k} \theta_i^2.$$

As you saw in lecture, the partial derivatives for this function have an elegant form:

$$\frac{\partial}{\partial \theta_i} \text{nll}(\mathbf{x}, \mathbf{Y}, \theta) = E_\theta[f_i] - E_D[f_i] + \lambda \theta_i.$$

In the derivative, there are two expectations: $E_\theta[f_i]$, the expectation of feature values with respect to the model parameters, and $E_D[f_i]$, the expectation of the feature values with respect to the given data instance $D \equiv (X, y)$. Using the definition of expectation, we have:

$$E_\theta[f_i] = \sum_{\mathbf{Y}'} P(\mathbf{Y}'|\mathbf{x}) f_i(\mathbf{Y}', \mathbf{x}),$$

$$E_D[f_i] = f_i(\mathbf{Y}, \mathbf{x}).$$

Note that we drop the θ from $P(\mathbf{Y}'|\mathbf{x})$ for convenience.

In the first equation, we sum over all possible assignments to the $\mathbf{Y}$ variables in the scope of feature f_i. Since each feature has a small number of $\mathbf{Y}$ variables in its scope, this sum is tractable. Unfortunately, computing the conditional probability $P(\mathbf{Y}'|\mathbf{x})$ for each assignment requires performing inference for the data instance $\mathbf{x}$.

There is one more detail to take care of: shared parameters across multiple features. As mentioned in Week 2's lecture on Shared Features in Log-Linear Models, parameter sharing is a form of templating used to reduce the total number of parameters we need to learn. Let $\left\{f^{(i)}\right\}$ be the set of features that share parameter θ_i. Then we can expand the equations above as:

$$\text{nll}(\mathbf{x}, \mathbf{Y}, \theta) \equiv \log(Z_{\mathbf{x}}(\theta)) - \sum_{i=1}^{k} \theta_i \left(\sum_{f_j \in \left\{f^{(i)}\right\}} f_j(\mathbf{Y}, \mathbf{x}) \right) + \frac{\lambda}{2} \sum_{i=1}^{k} \theta_i^2,$$

$$\frac{\partial}{\partial \theta_i}\text{nll}(\mathbf{x}, \mathbf{Y}, \theta) \quad = \quad \sum_{f_j \in \left\{f^{(i)}\right\}} E_\theta[f_j] - \sum_{f_j \in \left\{f^{(i)}\right\}} E_D[f_j] + \lambda\theta_i.$$

Note that in this case, θ_i is not necessarily related to f_i any longer. Instead, θ_i is associated with a set of features $\left\{f^{(i)}\right\}$. k is the total number of parameters θ_i, and not necessarily the total number of features f_j.

The parameters that we use in the CRF are:

- θ^C_c, shared by $\left\{f^C_{i,c}(Y_i)\right\}_i$.
- $\theta^I_{c,d}$, shared by $\left\{f^I_{i,j,c,d}(Y_i, x_{ij})\right\}_{i,j}$.
- $\theta^P_{c,d}$, shared by $\left\{f^P_{i,c,d}(Y_i, Y_{i+1})\right\}_i$.

Essentially, this parameter tying scheme ties parameters across different locations together; that is, a character at the start of the word shares the same parameters as a character at the end of the word.

Given this discussion, we can now state your mission: for a given data instance $(\mathbf{x}, \mathbf{Y})$ and a parameter setting θ, you must compute the cost function (regularized negative log-likelihood) and the gradient of parameters with respect to that cost. Doing so involves six terms (ignoring the issue of shared parameters in these):

- The log partition function: $\log(Z_\mathbf{x}(\theta))$
- The weighted feature counts: $\theta_i f_i(\mathbf{Y}, \mathbf{x})$
- The regularization cost: $\frac{\lambda}{2}\sum_{i=1}^k \theta_i^2$
- The model expected feature counts: $\sum_{\mathbf{Y}'} P(\mathbf{Y}'|\mathbf{x}) f_i(\mathbf{Y}', \mathbf{x})$,
- The data feature counts: $f_i(\mathbf{Y}, \mathbf{x})$
- The regularization gradient term: $\lambda\theta_i$

Take note that you will have to incorporate parameter sharing into these terms.

Unlike previous assignments, how you structure your code is almost entirely up to you. We will test only two functions:

1. `CliqueTreeCalibrate.m`: You should modify this function (from PA 4) to also compute $\log(Z_\mathbf{x}(\theta))$.
2. `InstanceNegLogLikelihood.m`: You should fill out this function to compute the regularized negative log-likelihood and its gradient for a single data instance $(\mathbf{x}, \mathbf{Y})$.

But we are getting ahead of ourselves...

3.4 Learning CRF Parameters: Implementation

At a high level, the entirety of the assignment is to implement `InstanceNegLogLikelihood.m`. Make sure to read the comments in the file: it contains a helpful discussion of the data structures you are using. The features above might look very complicated, but in `InstanceNegLogLikelihood`

we've provided you with a function that generates all of the required features and stores them in `featureSet`.

First, we should clarify what exactly we mean by a data "instance." The input to `InstanceNegLogLikelihood.m` includes `X` and `y` which together form a single instance (X, y). This instance corresponds to one word, that is, a sequence of characters. In fact, the data we have given you contains short sub-sequences of words; we do this to make sure the model is fast enough to train.

Thus, the variable `y` is a vector of character values. For example, $y = (3, 1, 20)$ means that this instance contains the word "cat." The variable `X` is a matrix where the ith row contains the pixel values for the ith character in the example. Each row has 32 values, since the images are 8x4 pixels. Again, these are smaller images than one would use in a full-strength OCR system, but we want to keep the model small. You can visualize one of these character sequences by calling `VisualizeCharacters(X)`, where `X` is a matrix such as the one just described (i.e., `numCharacters x 32`). The feature sharing described above comes into play due to the fact that our data instances are not single images (as they were in the first part of the assignment) but instead sequences of images that we want to consider together.

The function `InstanceNegLogLikelihood` computes the negative log likelihood and gradient with respect to a one of these examples that is a single word (or really a subsequence of a word). The steps you need to take in this function are:

1. Convert the `featureSet` into a clique tree with its factors filled in. This tree should be a chain, so the first clique has scope $\{Y_1, Y_2\}$, the second has scope $\{Y_2, Y_3\}$, and so on.

   (a) Modify `CliqueTreeCalibrate.m` and then call it to calibrate the tree and simultaneously compute $\log(Z_{\mathbf{x}}(\theta))$.

2. Use the calibrated clique tree in conjunction with `featureSet` to compute the weighted feature counts, the data feature counts, and the model expected feature counts.

3. Use these counts to compute the regularized negative log-likelihood and gradient.

We recommend that you do not try to compute all of these within `InstanceNegLogLikelihood` itself. For reference, our solution defines six helper functions. We ask that you define helper functions within `InstanceNegLogLikelihood.m` so that we get these functions when you submit your code. You are of course free to keep the helper functions in separate files during your own development.

In terms of testing your code, there are three tests for this part:

- `LogZ` (10 points): This test checks that you have correctly modified `CliqueTreeCalibrate.m` (originally from PA4) to compute the variable `logZ`. Be sure to read the comment at the top: we have modified the code for you to also keep track of message that are not normalized (in the variable `unnormalizedMessages`). You will need this to compute `logZ`.

- `CRFNegLogLikelihood` (30 points): This test checks that you correctly compute `nll`, the first return value of `InstanceNegLogLikelihood.m`.

- `CRFGradient` (40 points): This test checks that you correctly compute `grad`, the second return value of `InstanceNegLogLikelihood.m`.

For debugging, you can use the variables from `Part2Sample.mat`. It contains the following values:

- `sampleX`, `sampleY`, `sampleTheta`, `sampleModelParams`: Input to `InstanceNegLogLikelihood.m`.

- `sampleUncalibratedTree`: The resulting clique tree you should construct from the `featureSet`. This is also the input to `CliqueTreeCalibrate` to test the computation of `logZ`.
- `sampleCalibratedTree`: The same clique tree after calibration.
- `sampleLogZ`: The log partition function for this data instance (computed at the same time as clique tree calibration).
- `sampleFeatureCounts`: The data feature counts for the input.
- `sampleWeightedFeatureCounts`: The weighted feature counts for the input.
- `sampleModelFeatureCounts`: The model expected feature counts for this input.
- `sampleRegularizationCost`: The regularization cost for the setting of theta.
- `sampleRegularizationGradient`: The gradient term introduced by regularization.
- `sampleNLL`: The output regularized negative log-likelihood.
- `sampleGrad`: The output gradient.

You can check for yourself that the equations for `nll` and `grad` hold. That is:

- `sampleNLL == sampleLogZ - sum(sampleWeightedFeatureCounts)`
  `+ sampleRegularizationCost`

and

- `sampleGrad == sampleModelFeatureCounts - sampleFeatureCounts`
  `+ sampleRegularizationGradient`.

3.5 Training the Full Model and Computational Concerns

You might have noticed that we are not asking you to train a full model with stochastic gradient descent as part of the assignment. This is not because we feel it is unimportant: the entire point of implementing `InstanceNegLogLikelihood` is to use it with stochastic gradient descent and a large dataset to train a set of parameters that can make good predictions on new data.

Instead, we have left it out due to computational concerns. Our implementation is reasonably optimized and takes about two seconds per call to `InstanceNegLogLikelihood`. It is for this reason that stochastic gradient descent is a good choice of optimization algorithm. Other methods require that we compute the average gradient over the entire dataset before making a single parameter update. With 100 examples and two seconds per example, we would require 3 minutes for every parameter update (and batch gradient descent will typically require hundreds of updates to converge).

With stochastic gradient descent, one can get reasonable results after only a few passes through the data. Even so, training takes on the order of ten to twenty minutes. In the grand scheme of things, this is not that much, but we do not want the bulk of implementation effort to be spent on some of the optimization tricks we employed.

As a result, training and testing the full CRF model is an entirely optional component of the assignment. In `Part2FullDataset.mat`, we provide `trainData` and `testData`. Each is a struct array with `X` and `y` fields. In both, each `X`/`y` pair is the input the `InstanceNegLogLikelihood`: `X` gives the observations and `y` gives the labels. Even though we provide you with the correct

output for the test data, you should not use it during training (if you want your numbers to be fairly comparable).

To train the full model, you should initialize the `theta` vector to `zeros(1,2366)` (the features have a total of 2366 parameters); the model parameters are the same as the sample data, so you can use those. You can then use stochastic gradient descent from Section 2 to update θ based on the gradient for each instance that you compute using `InstanceNegLogLikelihood`.

With a learning rate of $\alpha_k = \frac{1}{1+0.05k}$ and a value of $\lambda = 0.003$, we obtained train and test accuracies of 73% and 62%, respectively, after 5 passes through the data with stochastic gradient descent.

Here are some possible ideas that you could explore to improve test accuracy if you would like to try to beat our scores:

- Use cross-validation on the `trainData` to pick a good value of λ. If you don't do this, you could try $\lambda = 0.003$; we've found that it works pretty reasonably.
- Create more training data (e.g., by distorting the provided samples, or even collecting new handwriting samples).
- Swap out stochastic gradient descent for a more sophisticated algorithm, e.g., L-BFGS with mini-batching. Matlab and Octave both come with built-in function optimizers (see `fminunc`).
- For the truly ambitious, add in more features (e.g., triplet features)! This will require understanding how `featureSet` is generated, so read that code carefully.

Some of these are beyond the scope of this class, and you will have to do your own reading up to implement them. Good luck!

4 Conclusion

Congratulations on completing a very challenging assignment! This assignment ties together much of the course material, from representation (CRFs and template models) to inference, and finally learning. We hope that you have managed to get a feel for the interplay between these various aspects of graphical models. In particular, we hope that you have managed to see for yourself why training an undirected model is a computationally difficult task, due to the fact that we have to perform inference on every step of the optimization process; those of you who decided to train the full model would be well aware of this fact. There are a variety of methods to help improve the speed of learning a CRF, but many of these are beyond the scope of this class. If you are interested, a detailed tutorial is available online at `http://arxiv.org/pdf/1011.4088v1`.