HOMEWORK #4: LOGISTIC REGRESSION

Probabilistic Learning: Theory and Algorithms
CS 274A, Winter 2018
Due: Friday, February 23rd, 2018, 11:55 PM
Submit code and report via EEE Dropbox
You should submit a zip file containing both your report and the code files.

INTRODUCTION

Model Definition. *Logistic regression* is a simple yet powerful and widely used binary classifier. Given a $(1 \times d)$-dimensional feature vector $\mathbf{x}$, a d-dimensional vector of real-valued parameters $\boldsymbol{\beta}$, a real-valued bias parameter β_0, and an output variable $\theta \in (0, 1)$, the logistic regression classifier is written as

$$\hat{\theta} = f(\mathbf{x}\boldsymbol{\beta} + \beta_0) \text{ where } f(z) = \frac{1}{1 + e^{-z}}.$$

$f(\cdot)$ is known as the *sigmoid* or *logistic* function, which is where *logitic regression* gets its name. For simplicity, from here forward, we will absorb the bias parameter into the vector by appending a 1 to all feature vectors, making $\mathbf{x}$ and $\boldsymbol{\beta}$ $(d + 1)$-dimensional, which allows us to write $\hat{\theta} = f(\mathbf{x}\boldsymbol{\beta})$.

Learning. To train a logistic regression model, we of course need labels: an observed class assignment $y \in \{0, 1\}$ for a given feature vector $\mathbf{x}$. Although logistic regression, at first glance, may seem different than the simple models we have been working with so far (ex: Gaussians, Betas, Gammas, Multinomials, etc), we still learn the parameters $\boldsymbol{\beta}$ via the *maximum likelihood* framework. Yet, note, here we are working with what's called a *conditional* model of the form $p(y|x, \theta)$, which assumes access to another data source (the features) to be fully specified.

The training objective is derived as follows; we start with just one observation $\{\mathbf{x}, y\}$ for notational simplicity. Because the variable we want to model—the class label y—is binary, we use the natural distribution for binary values: the *Bernoulli* model. And since $\hat{\theta}$ is between zero and one, we treat it as the Bernoulli's 'success probability' parameter—we defined the logistic regression model above with this goal in mind. Then, just as we usually do in the maximum likelihood framework, we start by writing down the log probability:

$$\begin{aligned} \log p(y|\mathbf{x}, \boldsymbol{\beta}) &= \log[\hat{\theta}^y (1 - \hat{\theta})^{1-y}] \text{ (inside the log is the Bernoulli p.m.f.)} \\ &= y \log \hat{\theta} + (1 - y) \log(1 - \hat{\theta}) \\ &= y \log f(\mathbf{x}\boldsymbol{\beta}) + (1 - y) \log(1 - f(\mathbf{x}\boldsymbol{\beta})). \end{aligned} \tag{1}$$

You may have seen this equation before as the *cross-entropy* error function. To arrive at the third line, all we did was substitute $\hat{\theta} = f(\mathbf{x}\boldsymbol{\beta})$. The intuition behind the equations above is that we are learning a Bernoulli model for the observed class labels, but instead of learning just one fixed Bernoulli parameter, we model the success parameter as function of the features $\mathbf{x}$. Defining the objective for a whole dataset $\mathcal{D} = \{\mathcal{D}_{\mathbf{x}}, \mathcal{D}_y\} = \{(\mathbf{x}_1, \ldots, \mathbf{x}_i, \ldots, \mathbf{x}_N), (y_1, \ldots, y_i, \ldots, y_N))$, we have

$$\begin{aligned} \log p(\mathcal{D}_y|\mathcal{D}_x, \boldsymbol{\beta}) &= \log \prod_{i=1}^{N} p(y_i|\mathbf{x}_i, \boldsymbol{\beta}) \\ &= \sum_{i=1}^{N} y_i \log f(\mathbf{x}_i\boldsymbol{\beta}) + (1 - y_i) \log(1 - f(\mathbf{x}_i\boldsymbol{\beta})). \end{aligned} \tag{2}$$

We need to solve Equation 2 for the maximum likelihood estimate $\hat{\boldsymbol{\beta}}^{\text{MLE}}$. This is where the process changes from what we did previously. If we try to set $\nabla_{\boldsymbol{\beta}} p(\mathcal{D}_y|\mathcal{D}_x, \boldsymbol{\beta}) = 0$ and solve for $\boldsymbol{\beta}$, we will quickly find ourselves stuck, unable to find a closed-form solution. This is because our use of the logistic function has made our function *non-convex* with respect to $\boldsymbol{\beta}$. Thus we must optimize via

an iterative *gradient ascent*[1] method. In other words, we need to iterate the following equation to gradually send $\nabla_{\boldsymbol{\beta}} p(\mathcal{D}_y|\mathcal{D}_x, \boldsymbol{\beta}) \to 0$, which will indicate we're at a local maximum:

$$\hat{\boldsymbol{\beta}}_{t+1} = \hat{\boldsymbol{\beta}}_t + \alpha \, \nabla_{\boldsymbol{\beta}} \log p(\mathcal{D}_y|\mathcal{D}_x, \hat{\boldsymbol{\beta}}_t) \tag{3}$$

where $\hat{\boldsymbol{\beta}}_t$ is the current estimates of the parameters and α is known as the *learning rate*, the size of the steps we take when ascending the function. α can be fixed, but usually we have a gradually decreasing learning rate (i.e. $\alpha_t \to 0$ as t increases).

Using Stochastic Gradients. In Part 2 of Problem #2, we will vary what is called the training *batch size*. All this means is that we will approximate the gradient of the full dataset using only a subset of the data:

$$\nabla_{\boldsymbol{\beta}} \log p(\mathcal{D}_y|\mathcal{D}_x, \boldsymbol{\beta}) \approx \sum_{k=1}^{K} \nabla_{\boldsymbol{\beta}} \log p(y_k|\mathbf{x}_k, \boldsymbol{\beta}) \;\text{ such that } K << N, \tag{4}$$

where K is the batch size parameter. Training using this approximation is known as *stochastic* gradient ascent/descent, as we are using a stochastic approximation of the gradient. This method is commonly used because the gradient is usually well approximated by only a few instances, and therefore we can make parameter updates much faster than if we were computing the gradient using all N data points.

One important implementation detail to note when using and comparing various batch sizes is to take the *average* when computing Equation 2, not a straight sum. This is necessary because averaging standardizes the learning rate across various batch sizes. This doesn't change the optimization objective since $1/K$ is a constant that can be absorbed into the learning rate.

Prediction and Evaluation. After the training procedure has converged (i.e. the derivatives are close to zero), we can use our trained logistic regression model on a never-before-seen (test) dataset of features $\mathcal{D}_{\mathbf{x}}^{\text{test}}$ as follows. For a feature vector $\mathbf{x}_j \in \mathcal{D}_{\mathbf{x}}^{\text{test}}$, we predict it's label according to

$$\hat{y}_j = \begin{cases} 1, & \text{if } \hat{\theta}_j \geq 0.5 \\ 0, & \text{if } \hat{\theta}_j < 0.5 \end{cases} \text{ where } \; \hat{\theta}_j = f(\mathbf{x}_j \hat{\boldsymbol{\beta}}_T). \tag{5}$$

The subscript T on the parameters denotes that these are the estimates as found on the last update (i.e. $t \in [0, T-1]$). Notice that we are generating the prediction by taking the *mode* of the predicted Bernoulli distribution, or in other words, rounding $\hat{\theta}_j$.

If we do have labels for the test dataset—call them $\mathcal{D}_y^{\text{test}}$—we can use them to evaluate our classifier via the following two metrics. The first one is *test-set log likelihood*, and it is calculated using the same function we used for training:

$$\begin{aligned} \log p(\mathcal{D}_y^{\text{test}}|\mathcal{D}_x^{\text{test}}, \hat{\boldsymbol{\beta}}_T) &= \log \prod_{j=1}^{M} p(y_j^{\text{test}}|\mathbf{x}_j^{\text{test}}, \hat{\boldsymbol{\beta}}_T) \\ &= \sum_{j=1}^{M} y_j^{\text{test}} \log \hat{\theta}_j + (1 - y_j^{\text{test}}) \log(1 - \hat{\theta}_j), \end{aligned} \tag{6}$$

where, as above, $\hat{\theta}_j = f(\mathbf{x}_j^{\text{test}} \hat{\boldsymbol{\beta}}_T)$. The second metric calculates classification performance directly. Define the *classification accuracy* metric as follows:

$$\mathcal{A} = \frac{1}{M} \sum_{j=1}^{M} \mathbb{I}[y_j^{\text{test}} = \hat{y}_j] \tag{7}$$

where $\mathbb{I}$ is the indicator function and takes value 1 if the true test label matches the predicted label and 0 if there is a mis-match. $\mathcal{A}$, then, simply counts the number of correct predictions and divides by the size of the test set, M.

[1] Gradient *descent* and gradient *ascent* are essentially the same procedure. Turning one into the other is just a matter of placing a negative sign in front of the objective and moving the opposite direction (adding or subtracting the gradient term in the update).

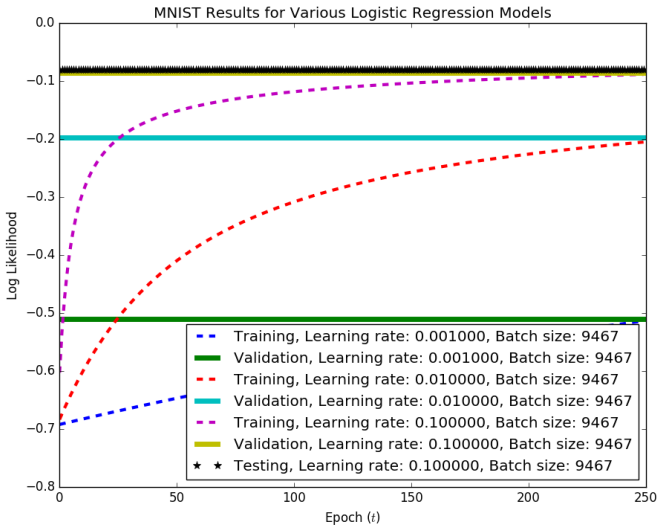

Figure 1: Example plot for Problem 2.1. The plot shows the log likelihood for during the training of three models, the log likelihood on the validation data for each model, and the log likelihood on the test set for the best model.

Dataset. In this homework, we will be implementing a logistic regression classifier and therefore need some dataset to train it on. For this we will use a very popular dataset of handwritten digit images called *MNIST*. You will find this dataset in many machine learning publications; it is mostly used for illustrative purposes or as a 'sanity check' since it's easy to train a high accuracy model for it. The full version of MNIST has ten classes, but we've reduced the dataset down to only 2's and 6's to formulate a binary classification problem. We have already segmented the data into training, validation, and testing splits of 9467 instances, 2367 instances, and 1915 instances respectively. Each image is represented as a vector with 784 dimensions, each dimension corresponding to pixel intensity between zero and one, and therefore the training features, for instance, comprise a matrix of size 9467×784. The data can be downloaded in CSV format from the EEE dropbox for the assignment under the **CourseFiles** folder.

PROBLEM #1: GRADIENT DERIVATION

Given: We start by filling in the details of Equation 2. Be sure to be explicit and precise in your solutions so even if someone did not have the prompt, they could follow your work. You can start from the last line of Equation 2. You can also assume the derivative of the logistic function is given: $\frac{\partial}{\partial z} f(z) = f(z)(1 - f(z)) \frac{\partial}{\partial z}$.

Part 1 (15 points): Derive an expression for $\nabla_{\boldsymbol{\beta}} \log p(\mathcal{D}_y | \mathcal{D}_x, \hat{\boldsymbol{\beta}}_t)$.

Part 2 (5 points): State the intuition behind the expression—i.e. why should sending the gradient to zero make the classifier 'work'?

PROBLEM #2: IMPLEMENTATION

Given: Now we can implement a logistic regression classifier. Using the programming language of your choice, write a script that trains a logistic regression model and evaluates the model by calculating log likelihood (Equation 6) and accuracy (Equation 7). Some skeleton Python code is provided in the Appendix (and in EEE dropbox). You can choose not to use this code at all if you wish. The only requirement is that *no* logistic regression or auto-differentiation libraries can

be used—i.e. a library that implements/trains a logistic regression model as a function call or that calculates gradients automatically. For all parts, take the number of *epochs*, i.e. passes through the training data, to be 250. The total number of updates will then be $T = 250 \times N/\text{batch\_size}$.

Each subproblem below gives three parameter settings for learning rate (part 1) or batch size (problem 2). Train three models, one for each setting, and once training is complete, calculate the accuracy and log likelihood on the validation set. And then, for the model that had the highest validation accuracy, calculate the test-set accuracy and log likelihood. Report the log likelihood numbers in a plot similar to Figure 1; report the accuracies numerically. **Include all code for your implementation.**

Part 1 (20 points): The first task is to train three logistic regression models, one for each of the following learning rates (α in Equation 3): 0.001, 0.01, 0.1. Report the results as described above. In a few sentences, comment on the results.

Part 2 (20 points): The second task is to train three more logistic regression models, one for each of the following batch sizes (K in Equation 4): 1000, 100, 10. Report the results as described above. In a few sentences, comment on the results.

PROBLEM #3: ADAPTIVE LEARNING RATES

Given: In Problem # 2, we fixed the learning rate across all updates. Here we will look at how to intelligently change the learning rate as training progresses. We will experimentally compare the following three ways to change the step size.

1. *Robbins-Monro Schedule*: One method is to use a deterministic schedule to decay the learning rate as training progresses. The Robbins-Monro schedule, suggested because of some theoretical properties we won't discuss here, is written as

$$\alpha_t = \alpha_0/t \text{ for } t \in [1, T-1]. \tag{8}$$

   In other words, we set the learning rate at time t by dividing some fixed, initial learning rate α_0 by the current time counter.

2. *AdaM*: A second method is called *AdaM*, for (Ada)ptive (M)oments. The method keeps a geometric mean and variance of the gradients observed so far and updates the parameters according to

$$\hat{\boldsymbol{\beta}}_{t+1} = \hat{\boldsymbol{\beta}}_t + \alpha_0 \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}, \tag{9}$$

   where α_0 is some fixed constant, $\hat{\mathbf{m}}_t$ is the geometric mean of the past gradients, $\hat{\mathbf{v}}_t$ the geometric variance, and ϵ is some small value to keep the denominator from being zero. Complete Python code for performing AdaM updates is provided in the Appendix.

3. *Newton-Raphson*: The last method we will consider is *Newton-Raphson*, also called *Newton's Method*. This is somewhat of the gold-standard for adaptively setting the learning rate as there are *no* parameters to choose ourselves. We simply set the learning rate to the inverse Hessian matrix, which in the case of logistic regression is:

$$\alpha_t = (\mathbf{X}^T \mathbf{A}_t \mathbf{X})^{-1} \tag{10}$$

   where $\mathbf{A}_t$ is a diagonal matrix with $a^t_{i,i} = \hat{\theta}_i(1 - \hat{\theta}_i)$ (i.e. the predicted Bernoulli probability using the parameter estimates at time t) and $\mathbf{X}$ is the $N \times d$ matrix of features ($K \times d$ for batch size K when using stochastic gradient methods). The problem with Newton's method is that computing the matrix inverse for each update is very costly. It's around $\mathcal{O}(d^3)$, which for MNIST is $\mathcal{O}(784^3) = \mathcal{O}(481,890,304)$.

Part 1 (30 points): Implement *Robbins-Monro*, *AdaM*, and *Newton-Raphson* updates for the logistic regression model you implemented in Problem #2. Python code for AdaM is provided in the Appendix. Set α_0 to 0.1 for Robbins-Monro and set α_0 to 0.001 for Adam. Use a batch size[2] of

[2]Newton's method will take a few minutes to train for 50 epochs. Use a much bigger batch size to first make sure your Hessian calculation is correct.

200 and 50 epochs (passes through the full training data) for each method. Report the same plot and accuracies you did for Problem #2: one training curve and validation log likelihood for each method, test log likelihood for best method according to validation performance. Discuss what you observe in a few sentences. **Include all code for your implementation.**

Part 2 (10 points): Answer the following three questions.

1. What's the intuition behind Newton's Method? In other words, what does the Hessian tell us about a given location on the function?
2. What's the intuition behind the AdaM update? Think about what happens to the step size when the variance of the gradients ($\hat{\mathbf{v}}_t$) is large and when it is small. Is the underlying mechanism similar to Newton's method's use of the Hessian?
3. State the runtime and memory complexity (in big $\mathcal{O}$ notation) in terms of the feature dimensionality d and the batch size K for each update method.

APPENDIX

STARTER IMPLEMENTION

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import copy

### Helper Functions ###

def logistic_fn(z):
    ### TO DO ###

def load_data_pairs(type_str):
    return pd.read_csv("mnist_2s_and_6s/"+type_str+"_x.csv").values, pd.read_csv("mnist_2s_and_6s/"+type_str+"_y.csv").values

def run_log_reg_model(x, beta):
    ### TO DO ###

def calc_log_likelihood(x, y, beta):
    theta_hats = run_log_reg_model(x, beta)
    ### Return an average, not a sum!
    ### TO DO ###

def calc_accuracy(x, y, beta):
    theta_hats = run_log_reg_model(x, beta)
    ### TO DO ###


### Model Training ###

def train_logistic_regression_model(x, y, beta, learning_rate, batch_size, max_epoch):
    beta = copy.deepcopy(beta)
    n_batches = x.shape[0]/batch_size
    train_progress = []

    for epoch_idx in xrange(max_epoch):
        for batch_idx in xrange(n_batches):

            ### TO DO ###

            # perform updates
            beta += learning_rate * beta_grad

```

```
        train_progress.append(calc_log_likelihood(x, y, beta))
        print "Epoch %d.  Train Log Likelihood: %f" %(epoch_idx,
    train_progress[-1])

    return beta, train_progress


if __name__ == "__main__":

    ### Load the data
    train_x, train_y = load_data_pairs("train")
    valid_x, valid_y = load_data_pairs("valid")
    test_x, test_y = load_data_pairs("test")

    # add a one for the bias term
    train_x = np.hstack([train_x, np.ones((train_x.shape[0],1))])
    valid_x = np.hstack([valid_x, np.ones((valid_x.shape[0],1))])
    test_x = np.hstack([test_x, np.ones((test_x.shape[0],1))])

    ### Initialize model parameters
    beta = np.random.normal(scale=.001, size=(train_x.shape[1],1))

    ### Set training parameters
    learning_rates = [1e-3, 1e-2, 1e-1]
    batch_sizes = [train_x.shape[0]]
    max_epochs = 250

    ### Iterate over training parameters, testing all combinations
    valid_ll = []
    valid_acc = []
    all_params = []
    all_train_logs = []

    for lr in learning_rates:
        for bs in batch_sizes:
            ### train model
            final_params, train_progress =
    train_logistic_regression_model(train_x, train_y, beta, lr, bs,
    max_epochs)
            all_params.append(final_params)
            all_train_logs.append((train_progress, "Learning rate: %f,
    Batch size: %d" %(lr, bs)))

            ### evaluate model on validation data
            valid_ll.append( calc_log_likelihood(valid_x, valid_y,
    final_params) )
            valid_acc.append( calc_accuracy(valid_x, valid_y,
    final_params) )

    ### Get best model
    best_model_idx = np.argmax(valid_acc)
    best_params = all_params[best_model_idx]
    test_ll = calc_log_likelihood(test_x, test_y, best_params)
    test_acc = calc_accuracy(test_x, test_y, best_params)
    print "Validation Accuracies: "+str(valid_acc)
    print "Test Accuracy: %f" %(test_acc)

    ### Plot
    plt.figure()
    epochs = range(max_epochs)
    for idx, log in enumerate(all_train_logs):
        plt.plot(epochs, log[0], '--', linewidth=3, label="Training, "+
    log[1])
        plt.plot(epochs, max_epochs*[valid_ll[idx]], '-', linewidth=5,
    label="Validation, "+log[1])
```

```
    plt.plot(epochs,  max_epochs*[test_ll], '*', ms=8, label="Testing, "+
    all_train_logs[best_model_idx][1])

    plt.xlabel(r"Epoch ($t$)")
    plt.ylabel("Log Likelihood")
    plt.ylim([-.8, 0.])
    plt.title("MNIST Results for Various Logistic Regression Models")
    plt.legend(loc=4)
    plt.show()
```

ADAM UPDATE

```

def get_AdaM_update(alpha_0, grad, adam_values, b1=.95, b2=.999, e=1e-8):
    adam_values['t'] += 1

    # update mean
    adam_values['mean'] = b1 * adam_values['mean'] + (1-b1) * grad
    m_hat = adam_values['mean'] / (1-b1**adam_values['t'])

    # update variance
    adam_values['var'] = b2 * adam_values['var'] + (1-b2) * grad**2
    v_hat = adam_values['var'] / (1-b2**adam_values['t'])

    return alpha_0 * m_hat/(np.sqrt(v_hat) + e)

# Initialize a dictionary that keeps track of the mean, variance, and
    update counter
alpha_0 = 1e-3
adam_values = \
    {'mean': np.zeros(beta.shape), 'var': np.zeros(beta.shape), 't': 0}

### Inside the training loop do ###
beta_grad = # compute gradient w.r.t. the weight vector (beta) as usual
beta_update = get_AdaM_update(alpha_0, beta_grad, adam_values)
beta += beta_update
```