**Homework #5: Logistic Regression**

**Probabilistic Learning: Theory and Algorithms**
CS 274A, Winter 2020  
Due: Noon Wednesday, March 4th 2020  
Gradescope: plots and written responses; submit code via EEE Dropbox

---

**Introduction**

**Model Definition.** Logistic regression is a simple yet powerful and widely used binary classifier. Given a $(1 \times d)$-dimensional feature vector $x$, a $d$-dimensional vector of real-valued parameters $\beta$, a real-valued bias parameter $\beta_0$, and an output variable $y \in \{0, 1\}$, the logistic regression classifier is written as

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

$f(\cdot)$ is known as the **sigmoid** or **logistic** function, which is where logistic 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, which allows us to write $\theta_x = f(x\beta)$.

**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 $x$. Although logistic regression, at first glance, may seem different than the simple density and distribution models (such as Gaussians, Betas, Gammas, Multinomials, etc), we still learn the parameters $\beta$ via the maximum likelihood framework. 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).

Given a dataset $D = \{D_x, D_y\} = \{(x_1, \ldots, x_i, \ldots, x_N), (y_1, \ldots, y_i, \ldots, y_N)\}$, and assuming that the observations are conditionally independent given the parameters $\beta$, we can define the normalized log-likelihood as

$$\log p(D_y|D_x, \beta) = \frac{1}{N} \sum_{i=1}^{N} y_i \log f(x_i\beta) + (1 - y_i) \log(1 - f(x_i\beta)).$$ \hspace{1cm} (1)

We need to solve Equation 1 for the maximum likelihood estimate $\hat{\beta}^{\text{MLE}}$. This is where the process is different from what we typically did with estimating maximum likelihood parameters for simple densities/distributions such as Gaussians or multinomials. If we try to set $\nabla_\beta \log p(D_y|D_x, \beta) = 0$ and solve for $\beta$, we will quickly find ourselves stuck, unable to find a closed-form solution (we end up with a coupled set of non-linear equations in the unknowns $\beta$). One way to solve such problems is to optimize via iterative gradient ascent\(^1\) method. Let $\nabla_\beta \log p(D_y|D_x, \beta)$ be the gradient of the log-likelihood evaluated at the our current guess $\beta$ for the parameters—this is a vector of partial derivatives, one partial derivative for each component (each parameter) in the vector $\beta$. The gradient points in the steepest uphill direction (locally) of the log-likelihood—so we can take a small step in this direction to move uphill (the step needs to be small since “steepest” is only true at the current location). In other words, we need to iterate the following equation to gradually send $\nabla_\beta \log p(D_y|D_x, \beta) \to 0$, which will indicate we’re at a local maximum:

$$\beta_{t+1} = \beta_t + \alpha \nabla_\beta \log p(D_y|D_x, \beta_t)$$ \hspace{1cm} (2)

where $\beta_t$ is the current estimate of the parameters at iteration (or epoch) $t$, and $\alpha$ is known as the learning rate, the size of the steps we take when ascending the function. $\alpha$ can be fixed, but it is often more effective to use a gradually decreasing learning rate (i.e. $\alpha_t \to 0$ as $t$ increases).

\(^1\)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).
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:

\[ \nabla_{\beta} \log p(D_y|D_x, \beta) \approx \frac{1}{M} \sum_{m=1}^{M} \nabla_{\beta} \log p(y_m|x_m, \beta) \text{ such that } M << N, \]

(3)

where \( M \) 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.

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 holdout (test) dataset of features \( D_{x_{\text{test}}} \) as follows. For a feature vector \( x_j \in D_{x_{\text{test}}} \), we predict it’s label according to

\[ \hat{y}_j = \begin{cases} 1, & \text{if } \hat{\theta}_j \geq 0.5 \\ 0, & \text{if } \hat{\theta}_j < 0.5 \end{cases} \text{ where } \hat{\theta}_j = f(x_j \hat{\beta}_T). \]

(4)

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 using a point estimate \( \hat{\beta}_T \) of the parameter vector (an alternative Bayesian approach would be to average over the posterior density of the \( \beta \) given the data, but even for a relatively simple model like the logistic regression model, computing this posterior density is in general tractable—there is no conjugate prior and no closed form solution for the posterior).

Assuming that we have labels for the test dataset—call them \( D_{y_{\text{test}}} \)—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:

\[ \log p(D_{y_{\text{test}}}|D_{x_{\text{test}}}, \hat{\beta}_T) = \frac{1}{N_{\text{test}}} \sum_{j=1}^{N_{\text{test}}} y_{j_{\text{test}}} \log \hat{\theta}_j + (1 - y_{j_{\text{test}}}) \log(1 - \hat{\theta}_j), \]

(5)

where, as above, \( \hat{\theta}_j = f(x_{j_{\text{test}}} \hat{\beta}_T) \) and \( N_{\text{test}} \) is the number of examples in our test set.

The second metric calculates classification performance directly. Define the classification error metric as follows:

\[ \mathcal{E} = \frac{1}{N_{\text{test}}} \sum_{j=1}^{N_{\text{test}}} I[y_{j_{\text{test}}} \neq \hat{y}_j] \]

(6)

where \( I \) takes value 1 if the true test label does not match the predicted label and 0 if there is a match. \( \mathcal{E} \), then, simply counts the number of incorrect predictions and divides by the size of the test set, \( N_{\text{test}} \).

Dataset. In this homework, we will be implementing a logistic regression classifier and therefore need a 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 3’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 \( \times \) 784. The data can be downloaded in CSV format here:

\n
www.ics.uci.edu/~smyth/courses/data/mnist_2s_and_6s.zip
PROBLEM #1: GRADIENT-BASED LEARNING FOR LOGISTIC REGRESSION

Given: You will need to begin by defining the equation for the gradient for Equation 1. The derivative of the logistic function is \( \frac{\partial}{\partial z} f(z) = f(z)(1 - f(z)) \frac{\partial}{\partial z} \).

With the gradient equation we can now implement a logistic regression classifier. Using the programming language of your choice, write a script that (a) trains a logistic regression model and (b) evaluates a model (with some setting of the parameters) by calculating the log likelihood (Equation 5) and classification error (Equation 6). Some skeleton Python code is provided in the Appendix. 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 \( T = 250 \times N/batch\_size \).

The dataset is divided into 3 parts: training, validation, and test. Typically the model is fit to the training data and log-likelihood values on a validation set are used for example to determine (i) when to stop training (to prevent overfitting), and (ii) for model selection (to select a single model if multiple models are being trained). And then the test set is used for final evaluation of the selected model.

Each subproblem below gives three “hyperparameter” settings for learning rate (part 1) or batch size (problem 2). For each subproblem, train three models, one for each setting. All reported log-likelihoods should be normalized, i.e., divided by the number of samples used to calculate the log-likelihood as in Equation 1.

Report the following information for each subproblem:

1. Plot the log-likelihood during training as a function of epochs, for each parameter setting. Also draw 3 flat lines indicating the validation log-likelihood (of each trained model). And finally draw a flat line for the test log-likelihood of the model that performed best on the
validation data. Your graph should look like that of Figure 1. Use colors to differentiate the plots.

2. Generate tables, of the training, validation, and test performance of the trained models, for each subproblem, and for each of classification error and log-likelihood. One $3 \times 3$ table for classification error and one $3 \times 3$ table for log-likelihood, per subproblem: the rows should be the hyperparameter settings and the columns are the train/validation/test performance.

3. Briefly describe the results that you see in your plots and tables, e.g., convergence rates, error and log-likelihood values vary across training and validation sets, what would have happened if you had selected the best-performing model (e.g., using log-likelihood) on the validation data, etc.

Part 1 (20 points): The first task is to train three logistic regression models, one for each of the following learning rates ($\alpha$ in Equation 2): 0.001, 0.01, 0.1. Report the results as described above. I

Part 2 (20 points): The second task is to train three more logistic regression models, one for each of the following batch sizes ($M$ in Equation 3): 1000, 100, 10. Report the results as described above. You can fix the learning rate to 0.01 for this part.

PROBLEM #2: 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

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

In other words, we set the learning rate at time $t$ by dividing some fixed, initial learning rate $\alpha_0$ 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

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

where $\alpha_0$ is some fixed constant, $\hat{m}_t$ is the geometric mean of the past gradients, $\hat{v}_t$ the geometric variance, and $\epsilon$ 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 hyperparameters to choose ourselves. We simply set the learning rate to the inverse Hessian matrix, which in the case of logistic regression is:

$$\alpha_t = (X^T A_t X)^{-1}$$

where $A_t$ is a diagonal matrix with $a_{i,i} = \hat{\theta}_i (1 - \hat{\theta}_i)$ (i.e. the predicted Bernoulli probability using the parameter estimates at time $t$) and $X$ 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 $O(d^3)$, which for MNIST is $O(784^3) = O(481,890,304)$.

Part 1 (30 points): Implement Robbins-Monro, AdaM, and Newton-Raphson updates for the logistic regression model you implemented in Problem #1. Python code for AdaM is provided in the Appendix. Set $\alpha_0$ to 0.1 for Robbins-Monro and set $\alpha_0$ to 0.001 for Adam. Use a batch size of 2

\[^2\]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 type of plot for log-likelihood and the same types of tables for log-likelihood and classification errors as you did for Problem #1 above. 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 (\(\hat{v}_t\)) 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 \(O\) notation) in terms of the feature dimensionality \(d\) and the batch size \(M\) for each update method.

What to Submit
- Plots and written descriptions, in one document, to Gradescope
- A Zip file with all your code, including a script that runs the code (with comments), to Canvas.

Appendix

Starter Implementation

```python
## CS 274 2020, Homework 5, Skeleton Python Code for Logistic Regression
## (Code originally written in Winter 2017 by Eric Nalisnick, UC Irvine)

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_3s_and_6s/"+type_str+"_x.csv").values, pd.read_csv("mnist_3s_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
```
```python
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 = numpy.hstack((train_x, numpy.ones((train_x.shape[0], 1))))
    valid_x = numpy.hstack((valid_x, numpy.ones((valid_x.shape[0], 1))))
    test_x = numpy.hstack((test_x, numpy.ones((test_x.shape[0], 1))))

    ### Initialize model parameters
    beta = numpy.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 = numpy.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: +\text{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], '-o', 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("Normalized 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