CS 274A Homework 4

Caleb Nelson

Due Date: 11am Monday February 25th

Problem 1: Gradient Derivation

Letting $\theta = f(x\beta)$ and $\theta_i = f(x_i\beta)$,

$$\nabla_\beta log(p(D_y|D_x, \beta)) = \frac{\delta}{\delta\beta}\sum_{i=1}^{N} y_i log(\theta_i) + (1-y_i)log(1-\theta_i) \tag{1}$$

Since $\frac{\delta}{\delta\beta}f(x\beta) = f(x\beta)(1-f(x\beta)) = \theta(1-\theta)$,

$$\begin{aligned}\frac{\delta}{\delta\beta}\sum_{i=1}^{N} y_i log(\theta_i) + (1-y_i)log(1-\theta_i) &= \sum_{i=1}^{N}(\frac{y_i}{\theta_i} - \frac{1-y_i}{1-\theta_i})\frac{d}{d\beta}f(x_i\beta) \\ &= \sum_{i=1}^{N}(\frac{y_i}{\theta_i} - \frac{1-y_i}{1-\theta_i})\theta_i(1-\theta_i)x_i = \sum_{i=1}^{N}(\frac{y_i-\theta_i}{\theta_i(1-\theta_i)})\theta_i(1-\theta_i)x_i \\ &= \sum_{i=1}^{N}(y_i-\theta_i)x_i \end{aligned} \tag{2}$$

Problem 2: Implementation

Note that in the table of results on the next page, a batch size of 9467 means that all of the training data was treated as a single batch.

Learning Rate	Batch Size	Validation Acc	Test Acc
0.001	9467	0.98268	0.98381
0.001	1000	0.98479	0.98225
0.001	100	0.98479	0.98277
0.001	10	0.98479	0.98277
0.01	9467	0.97930	0.98486
0.01	1000	0.98141	0.98120
0.01	100	0.98183	0.98277
0.01	10	0.98226	0.98277
0.1	9467	0.97930	0.98486
0.1	1000	0.98183	0.97963
0.1	100	0.97381	0.97911
0.1	10	0.97296	0.97963

A couple trends stand out of these results. First looking at learning rate, it appears that lower learning rates lead to increased performance, but this effect all but goes away when the data is treated as a single batch. I belive this is because a higher rate, while making the model converge faster, leads to a slight loss in precision in margins as thin as the ones we see in the table, and this effect is enhanced when the batch size is decreased which leads to further impercision.

Looking at the batch size specifically, the two batch sizes that appeared to perform the best are batch sizes of 10 and and treating the data as a single batch. This seems like a strange result, as it means that we see the best performance on both ends of the spectrum of batch sizes instead of the middle, but I think it might be because mathematically, treating the data as a single batch is the same as using a batch size of 1. This would lead to the logical conclusion that breaking the data into smaller chunks and learning on those individually is more effective than trying to make broad conclusions on large swaths of data at a time.

Problem 3: Adaptive Learning Rates

1. In the table of results, we can see the AdaM performed the best for both batch sizes, followed by Newton's method, with R-M performing the worst. A batch size of 200 helped AdaM, but made R-M and Newton both perform worse. The sole execption is that a batch size of 200 actually helped R-M's performance on validation data, putting it ahead of Newton, but this might be a result of chance or a quirk of the data used as opposed to a result of actual statistical significance.

ALR Method	Batch Size	Validation Acc	Test Acc
R-M	200	0.97592	0.97389
AdaM	200	0.98690	0.98538
Newton	200	0.97254	0.97650
R-M	50	0.97381	0.97441
AdaM	50	0.98395	0.98381
Newton	50	0.97761	0.97963

2. (a) Every element of the Hessian matrix is the second partial derivative of the function we are trying to minimize in relation to the two features corresponding to its location in the matrix. This, multiplying it by our calculated gradiant will give us the direction we need to move each feature in order to bring our betas closer to a local extrema where the first derivative is 0.

   (b) The gradients represent roughly how close we are to our local extrema, with lower values for gradients corresponding to proximity to the local extrema. If we take the distribution of past gradients, the mean of this distribution will represent our average distance from the extrema and our variance will reprsent how consistently we stay a certain distance from it. As we move closer to the extrema, the gradients will decrease and we should expect the mean to fall and the variance to rise. Looking at the AdaM formula, both of these factors will lead to a shrinking learning rate, which is what we want. The underlying method is similar to Newton's in that we want to decreate the learning rate as we get closer to the extrema so that we don't overshoot it.

   (c) Note that I'm not taking the difficulty of calculating the gradient into account, as it is a neccesary step for every method and an equally difficult part for each.
   R-M: $O(1)$, scales only with the number of epochs
   AdaM: $O(1)$, also scales only with the number of epochs
   Newton: Multiplying the three matricies together requires $d * K * K + d * K * d$ operations. Inverting the result adds an additional d^3 operations. This leaves us with a big O of $O(dKK + dKd + d^3) = O(dMM)$ where $M = max(K, d)$.

Appendix: Source Code

```
## CS 274 2019, Homework 4, Skeleton Python Code for Logistic Regression
## Written by Caleb Nelson
## (Adapted from code originally written in Winter 2017 by Eric Nalisnick, UC Irvin

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

### Helper Functions ###

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def logistic_fn(z):
    return np.log(sigmoid(z))

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

def run_log_reg_model(x, beta):
    return sigmoid(np.dot(x, beta))

def calc_log_likelihood(x, y, beta):
    theta_hats = run_log_reg_model(x, beta)
    return np.sum(np.multiply(y, np.log(theta_hats+1e-100)) + np.multiply((1 - y),

def calc_accuracy(x, y, beta):
    theta_hats = run_log_reg_model(x, beta)
    correct = 0
    for i in range(y.shape[0]):
        if ((theta_hats[i] >= 0.5) == y[i]):
            correct += 1
    return correct / y.shape[0]

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)


### Model Training ###

def train_logistic_regression_model(x, y, beta, learning_rate, batch_size, max_epoc
    beta = copy.deepcopy(beta)
    n_batches = math.ceil(x.shape[0]/batch_size)
    train_progress = []
    adam_values = {'mean': np.zeros(beta.shape), 'var': np.zeros(beta.shape), 't':

    for epoch_idx in range(max_epoch):
        for batch_idx in range(n_batches):
            sob = batch_idx * batch_size
            eob = sob + batch_size
            x_batch = x[sob:eob]
            y_batch = y[sob:eob]
            theta_hats = run_log_reg_model(x_batch, beta)
            beta_grad = np.dot(x_batch.T, y_batch - theta_hats)

            if (alr == "R-M"):
                adam_values['t'] += 1
                beta += (learning_rate/adam_values['t']) * beta_grad
            elif (alr == "AdaM"):
                beta_update = get_AdaM_update(learning_rate, beta_grad, adam_values
                beta += beta_update
            elif (alr == "Newton"):
                theta_matrix = np.diag(np.multiply(theta_hats, 1-theta_hats).flatte
                hessian = np.linalg.multi_dot([x_batch.T, theta_matrix, x_batch])
                antisingular = np.zeros(hessian.shape)
                np.fill_diagonal(antisingular, 1e-10)
                hessian += antisingular
                beta += np.dot(np.linalg.inv(hessian), beta_grad)
```

```
            else:
                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

def problem_two():
    ### Set training parameters
    learning_rates = [1e-3, 1e-2, 1e-1]
    batch_sizes = [train_x.shape[0], 1000, 100, 10]
    max_epochs = 250

    ### Iterate over training parameters, testing all combinations
    valid_ll = []
    valid_acc = []
    test_acc = []
    results = []

    for lr in learning_rates:
        for bs in batch_sizes:
            ### train model
            final_params = train_logistic_regression_model(train_x, train_y, beta,

            ### evaluate model on validation and test data
            valid_ll.append( calc_log_likelihood(valid_x, valid_y, final_params) )
            acc = calc_accuracy(valid_x, valid_y, final_params)
            tacc = calc_accuracy(test_x, test_y, final_params)
            valid_acc.append(acc)
            test_acc.append(tacc)
            results.append((str(lr), str(bs), str(acc), str(tacc)))

    for result in results:
        print("Learning rate: "+result[0]+" Batch Size: "+result[1]+" Validation Ac

def problem_three():
    ### Set training parameters
    batch_sizes = [200, 50]
    max_epochs = 250
```

```
    ### Iterate over training parameters, testing all combinations
    valid_ll = []
    valid_acc = []
    test_acc = []
    results = []

    for bs in batch_sizes:
        for alr in ["R-M", "AdaM", "Newton"]:
            ### train model
            lr = 0.1 if alr == "R-M" else 0.001
            final_params = train_logistic_regression_model(train_x, train_y, beta,

            ### evaluate model on validation and test data
            valid_ll.append( calc_log_likelihood(valid_x, valid_y, final_params) )
            acc = calc_accuracy(valid_x, valid_y, final_params)
            tacc = calc_accuracy(test_x, test_y, final_params)
            valid_acc.append(acc)
            test_acc.append(tacc)
            results.append((alr, str(bs), str(acc), str(tacc)))

    for result in results:
        print("ALR: "+result[0]+" Batch Size: "+result[1]+" Validation Acc: "+resul

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))

    print("Running code for problem 2")
    problem_two()
    print("Running code for problem 3")
    problem_three()
```