ML 1

2. • The reason we subtract the mean before dividing is that we are trying to get a "standard score" or "z-score" of the data. This transformation makes the mean = 0 and the std = 1

→ This wouldn't be achieved if we divided by the standard deviation first

→ Consider:

$a = [105, 101, 95]$

$$\frac{a - \text{mean}(a)}{\text{std}(a)} = [0.7259 \quad 0.4148 \quad -1.106]$$

$$\frac{a}{\text{std}(a)} - \text{mean}(a) = [-67.2470, -67.5581, -69.1135]$$

Not centered around zero

2. We use the pre-computed mean of the training set, to normalize the test set for 2 reasons:
   1. The training set is much larger and thus the sample mean and sample standard deviation are more representative of the population
   2. It is generally best to not derive anything of your model from the test set, so that bench marking to other algorithms is much easier

3 ai) The derivative of a function and the derivative of the log of a function are zero at the same place → Thus minimizing one is like minimizing the other

aii) $$f(x) = \log\left[\prod_{i=1}^{m} \sigma(a_i^T x)^{b_i} (1-\sigma(a_i^T x))^{1-b_i}\right]$$

Log rules: 1) $\log(a \cdot b) = \log(a) + \log(b)$

2) $\log(a^b) = b\log(a)$

$$f(x) = \sum_{i=1}^{m} \log\left(\sigma(a_i^T x)^{b_i} (1-\sigma(a_i^T x))^{1-b_i}\right)$$

$$\sum_{i=1}^{m} \left[\log\left(\sigma(a_i^T x)^{b_i}\right) + \log\left(1-\sigma(a_i^T x)\right)^{1-b_i}\right]$$

$$f(x) = \sum_{i=1}^{m} \left[b_i \log(\sigma(a_i^T x)) + (1-b_i)\log\left(1-\sigma(a_i^T x)\right)\right]$$

$$\frac{\partial}{\partial x_k}\left(a_i^T x\right) = a_i^T \cdot \partial$$

$$f(x) = \sum_{i=1}^{m}\left[b_i \log\left(\sigma\left(a_i^T x\right)\right) + (1-b_i)\log\left(1-\sigma\left(a_i^T x\right)\right)\right]$$

Note: same as homework 1, except missing $\frac{1}{m}$ d -

$$g(h) = b\log(h) + (1-b)\log(1-h)$$

$$h(z) = \frac{1}{1+e^{-z}} \qquad z(x) = a^T x$$

$$\frac{dg}{dx} = \frac{dg}{dh} \cdot \frac{dh}{dz} \cdot \frac{dz}{dx}$$

$$\frac{dg}{dh} = \frac{b}{h} - \frac{1-b}{1-h} = \frac{b(1-h)}{h(1-h)} - \frac{h(1-b)}{h(1-h)} = \frac{b-bh-h+hb}{h(1-h)} = \frac{b-h}{h(1-h)}$$

$$\frac{dh}{dz} = h(z)\left(1-h(z)\right)$$

$$\frac{dz}{dx} = a^T \qquad \frac{dg}{dx} = \frac{b-h}{\cancel{h(1-h)}} \cdot \cancel{h(1-h)} \cdot a^T$$

$$= \left(b_i - \sigma\left(a_i^T x\right)\right) a_i^T \qquad m \times n$$

$$\sum_{i=1}^{m}\left(b_i - \sigma\left(a_i^T x\right)\right) a_i^T$$

$$\nabla f(x) = A^T\left(b - \sigma(Ax)\right)$$

$$\nabla f(x) = A^T(b - \sigma(Ax))$$

$$= A^T b - A^T \sigma(Ax)$$

$$\nabla^2 f(x) = A A^T \sigma(Ax)(1 - \sigma(Ax))$$

- Since Hessian for HW1 is convex then this should be concave

4b iv) Overall, the BTLS version is preferred even though it didn't perform much better.

- I do not have to worry about picking a stepsize as much
- It converges in less iterations
- Although each iteration of BTLS took a longer time

4 b v). Suprisingly, the linear regression model performed much better than the logistic regression by a large margin.

- This is suprising since logistic regression was much harder to implement.
- I would definitely choose linear regression model for this scenario, since I got much better results and it was easier. However, I would potentially use logistic regression if I had to deal with more classes (All numbers 1-9). Assuming I fixed the bugs...

Table of Contents


```
clear all
close all
```

Part 1 - Retrieval

```
load("mnist.mat")
%
% figure(1)
% clf
% i = 1;
% imshow(reshape(trainX(i,:),28,28)')
% title(trainY(i))
```

Part 2 - Pre-processing

```
idx = trainY == 4 | trainY == 9;
Atr = double(trainX(idx,:));
btr = double(trainY(idx))';
ntr = size(Atr, 2);
mtr = size(Atr, 1);
btr(btr==4)=1;
btr(btr==9)=-1;

idx_test = testY == 4 | testY == 9;
Atest = double(testX(idx_test,:));
btest = double(testY(idx_test))';
mtest = size(Atest, 1);
% Turn labels into +1 -1
btest(btest==4)=1;
btest(btest==9)=-1;

% Normalization
[Atr, Amean, Astd] = normalize(Atr);
% NOTE: It is important to not re-compute the Amean and Astd for 2
 reasons:
```

```
% 1. The training set has 10x more samples and the mean and standard
% deviation are more representative of the population mean
% 2. A rule of thumb when evaluating the performance of your model is
 to
% avoid touching the test set to avoid overtrainning

Atest = Atest - ones(mtest,1)*Amean;
Atest = Atest ./ max(ones(mtest,1)*Astd,1);
% Validation Functions
C = @(z) (z > 0)*2 - 1;
I = @(x,y) x ~= y;
misclass_rate = @(A,y,x) sum(I(C(A*x), y))/length(y);
```

Part 3 - Linear Regression

```
btr_ls = btr;
btest_ls = btest;
x_lr = Atr \ btr;

train_loss = norm(Atr*x_lr - btr_ls, 2)
test_loss = norm(Atest*x_lr - btest_ls, 2)

train_misclass_rate_lr = misclass_rate(Atr, btr_ls, x_lr)
test_misclass_rate_lr = misclass_rate(Atest, btest_ls, x_lr)
```

```
Warning: Rank deficient, rank = 607, tol =  2.842930e-10.

train_loss =

   46.2200


test_loss =

   26.3921


train_misclass_rate_lr =

    0.0308


test_misclass_rate_lr =

    0.0362
```

Part 4 - Logistic Regression

Pre-process

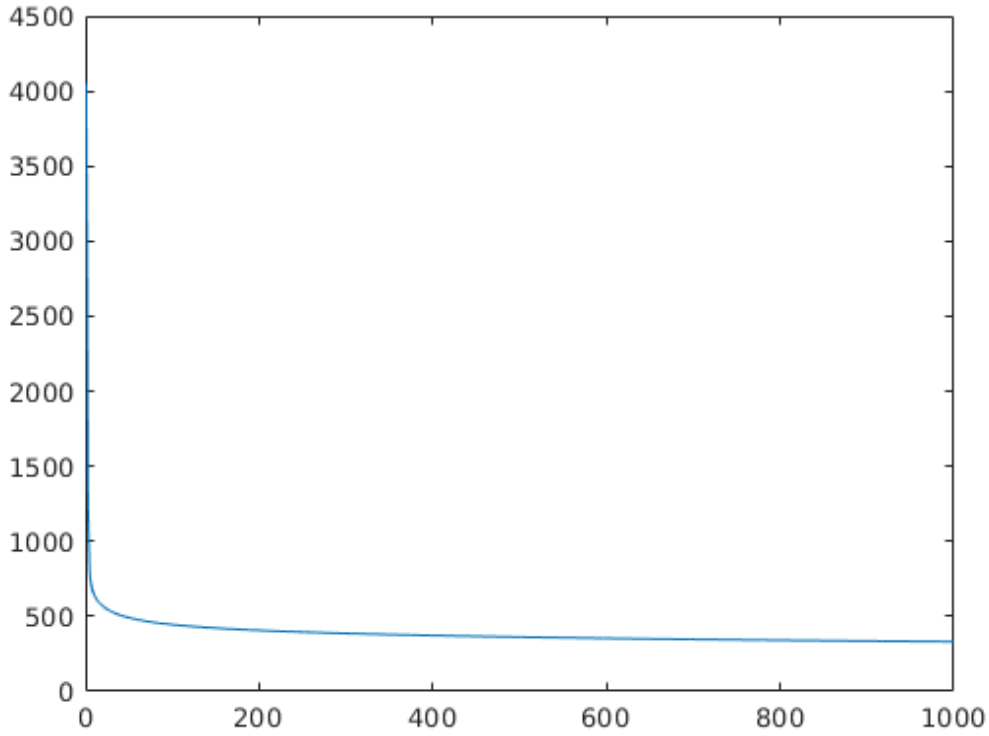
```
btr = (btr+1)/2;
```

```
btest = (btest+1)/2;

% Initialize functions
sig = @(x) 1.0 ./(1+exp(-x));
f = @(x) f_func(Atr, btr, x, sig);
g = @(x) Atr'*(sig(Atr*x) -btr);
x0 = zeros(ntr, 1);
epsilon = 1e-4;
max_iter = 1e3;
```

Gradient Descent

```
[x_gd, trace_gd, status] = gd(f, g, x0, 1/mtr ,max_iter, epsilon);
if status < 0
    disp("GD diverged")
end
train_misclass_rate_gd = misclass_rate(Atr, btr,  x_gd)
test_misclass_rate_gd = misclass_rate(Atest, btest, x_gd )
figure(1)
plot(trace_gd)
hold on
```

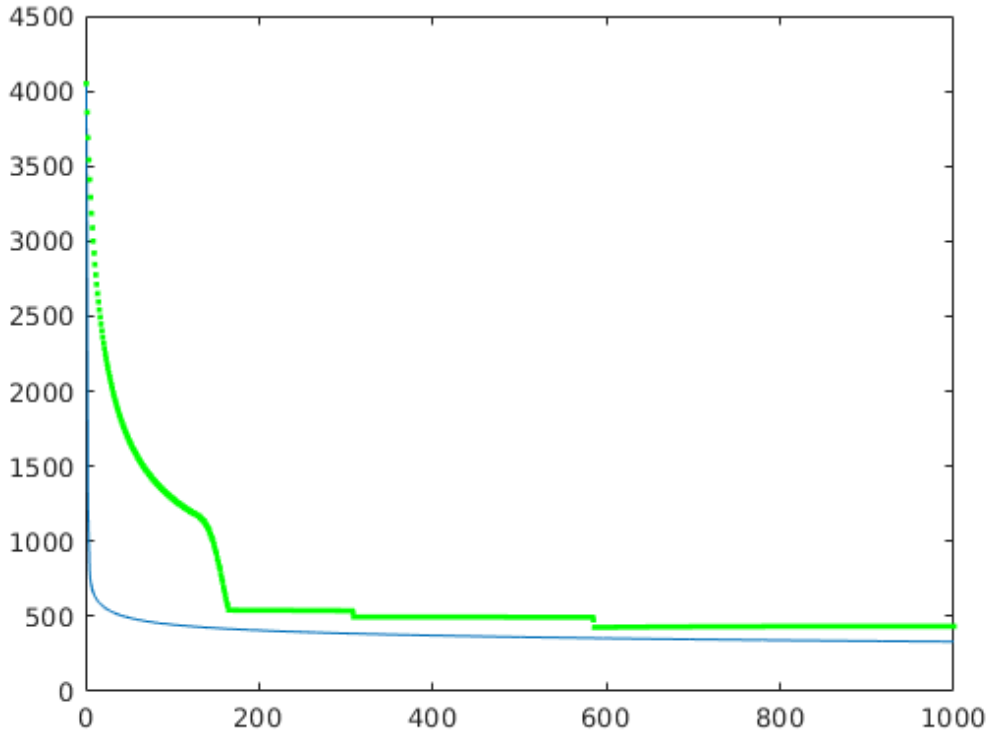
```
train_misclass_rate_gd =

    0.5146


test_misclass_rate_gd =

    0.5239
```



Gradient Descent With Backtracking Line Search

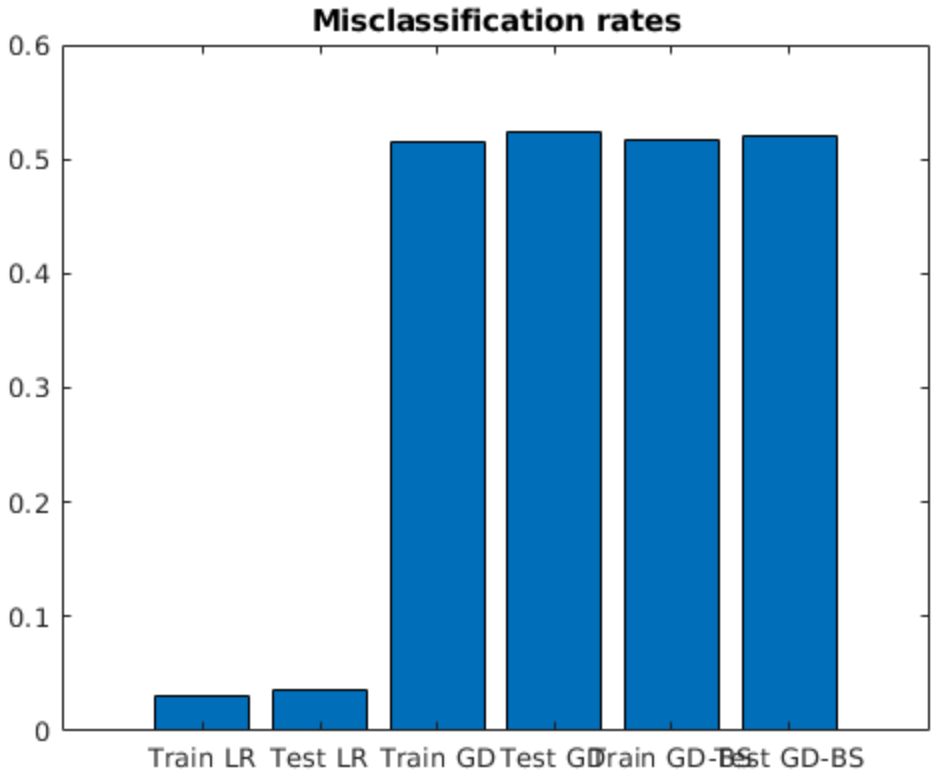
```
[x_gd_bt, trace_bt, status] = gd_bt(f, g, x0, 1, 0.5, 0.5, 1000,
 1e-1);
if status < 0
    disp("GD diverged")
end
train_misclass_rate_gd_btls = misclass_rate(Atr, btr, x_gd_bt)
test_misclass_rate_gd_btls = misclass_rate(Atest, btest, x_gd_bt)
plot(trace_bt, 'g.')
hold off
```

```
train_misclass_rate_gd_btls =

    0.5171


test_misclass_rate_gd_btls =

    0.5203
```



Comparisons

```
figure(2)
rates = [train_misclass_rate_lr, test_misclass_rate_lr,
 train_misclass_rate_gd,test_misclass_rate_gd,
 train_misclass_rate_gd_btls, test_misclass_rate_gd_btls];
bar(rates)
title('Misclassification rates')
set(gca,'xticklabel',{'Train LR', 'Test LR', 'Train GD','Test
 GD','Train GD-BS', 'Test GD-BS'});
```



Debugging

Linear regression worst samples

```
[M, ind] = maxk(abs(Atest*x_lr - btest_ls),3 );
figure(3)
imshow(reshape(Atest(ind(1),:),28,28)');
figure(4)
imshow(reshape(Atest(ind(2),:),28,28)');
figure(5)
imshow(reshape(Atest(ind(3),:),28,28)');
% Logistic regression worst samples
sig(Atest*x_gd);
[M, ind] = maxk(abs(sig(Atest*x_gd) - btest),3);
figure(6)
imshow(reshape(Atest(ind(1),:),28,28)');
figure(7)
imshow(reshape(Atest(ind(2),:),28,28)');
figure(8)
imshow(reshape(Atest(ind(3),:),28,28)');
```

Helper Functions

```
function [X, avg, Xstd] = normalize(X)
    [m, ~] = size(X);
    avg = mean(X,1);
    X = X - ones(m,1)*avg;
    Xstd = std(X,1);
    X = X ./ max(ones(m,1)*Xstd,1);
end

function cost = f_func(A, b, x, act_func)
    m = length(b);
    z = act_func(A*x);
    cost = sum(-log(z(b == 1))) + sum(-log(1 - z(b == 0)))/m;
end
```

Published with MATLAB® R2018a

```
function [xk,trace, status] = gd(f, g, x0, alpha, max_iters, epsilon)
trace = zeros(max_iters, 1);
xk = x0;
for k = 1:max_iters
    trace(k) = f(xk);
    dk = -g(xk);
    xk = xk + alpha*dk;
%     fprintf("k = %3d norm_grad = %2.6f f_val = %2.6f\n", k,
 norm(dk), f(xk));
    if norm(dk) < epsilon
        status = 1;
        trace = trace(1:k);
        return
    elseif k > 2 && trace(k) > trace(k-1)
        status = -1;
        trace = trace(1:k);
        return
    elseif isnan(norm(xk)) || ~isfinite(norm(xk))
        status = -1;
        trace = trace(1:k);
        return
    end
end
status = 0;
end
```

Published with MATLAB® R2018a

```
function [xk,trace, status] = gd_bt(f, g, x0, s, alpha, beta,
 max_iters, epsilon)
trace = zeros(max_iters, 1);
xk = x0;

for k = 1:max_iters
    trace(k) = f(xk);
    % Determine new step size
    dk = -g(xk); % Negative gradient is descent direction
%     fprintf("k = %3d norm_grad = %2.6f f_val = %2.6f\n", k,
 norm(dk), f(xk));

    tk = s;
    for i = 0:20
        if f(xk) - f(xk + tk*dk) >= -alpha*tk*g(xk)'*dk
            break
        end
        tk = s*beta^i;
    end

    xk = xk + tk*dk;
    % Early stopping conditions
    if norm(g(xk)) < epsilon
        trace = trace(1:k);
        status = 1;
        return
%     elseif k > 2 && trace(k) > trace(k-1)
%         trace = trace(1:k);
%         status = -1;
%         return
    elseif isnan(norm(xk)) || ~isfinite(norm(xk))
        trace = trace(1:k);
        status = -1;
        return
    end
end
status = 0;
end
```

Published with MATLAB® R2018a