Bob Ghosh, 42039157, j0k0b

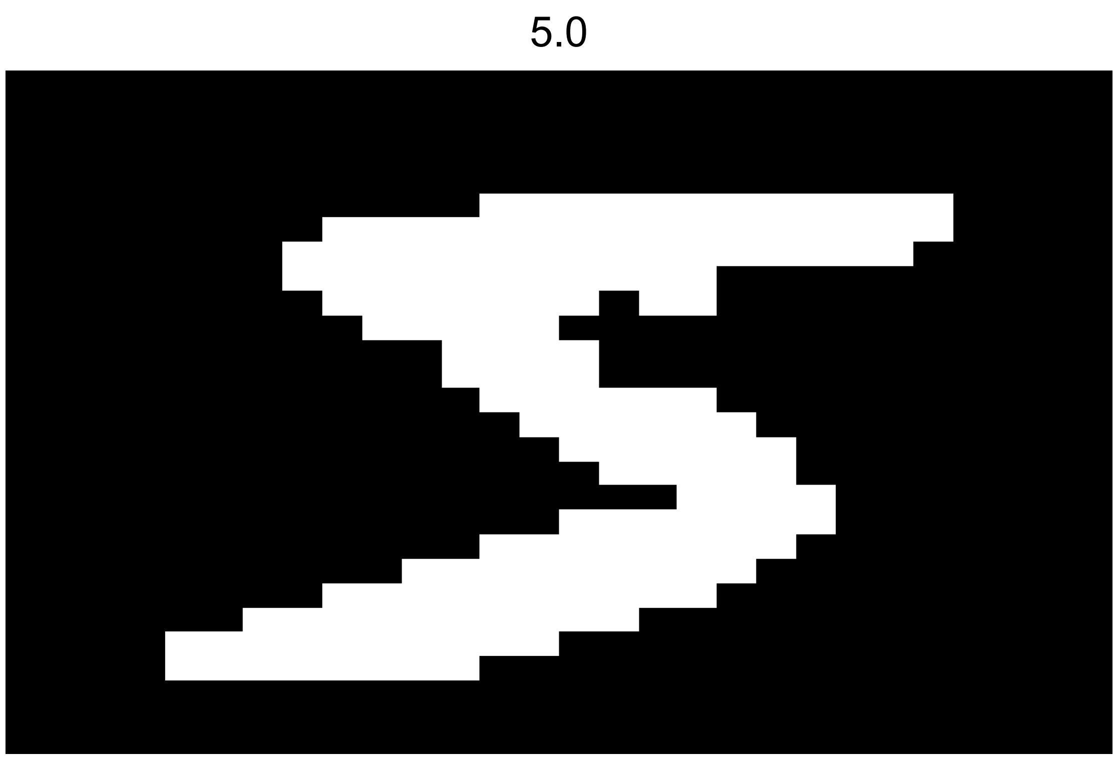
CPSC 406, In-Class Activity, 26th Feb

1. Retrieving Data

In [41]:
```julia
using MAT, Plots, Images, LinearAlgebra, ImageView, Colors, Statistics
# open file
file = matopen("mnist.mat")
trainX = float64.(read(file, "trainX"))
trainY = float64.(read(file, "trainY"))
testX = float64.(read(file, "testX"))
testY = float64.(read(file, "testY"));
```

In [2]:
```julia
i = 1
img = reshape(trainX[i,:], 28, 28)';
img = heatmap(Gray.(img))
plot(img, title = "$(trainY[i])", axis = false)
```

Out[2]:



2. Preprocessing

a

In [3]:
```
# pull out 4s and 9s from train set
idx4 = trainY .== 4
idx9 = trainY .== 9
idx = idx4 + idx9
idx = findall(x->x == 1, idx[1,:])

A = trainX[idx,:]
b = trainY[idx]
```

Out[3]:
```
11791-element Array{Float64,1}:
 4.0
 9.0
 4.0
 9.0
 4.0
 9.0
 4.0
 9.0
 9.0
 9.0
 9.0
 4.0
 9.0
 ⋮
 4.0
 9.0
 4.0
 9.0
 4.0
 9.0
 4.0
 9.0
 9.0
 4.0
 9.0
 9.0
```

In [4]:
```
idx4 = testY .== 4;
idx9 = testY .== 9;
idx = idx4 + idx9;
idx = findall(x->x == 1, idx[1,:]);

Atest = testX[idx,:]
btest = testY[idx]
```

Out[4]:
```
1991-element Array{Float64,1}:
 4.0
 4.0
 9.0
 9.0
 9.0
 9.0
 4.0
 9.0
 4.0
 4.0
 4.0
 4.0
 4.0
 ⋮
 4.0
 9.0
 9.0
 4.0
 4.0
 4.0
 9.0
 4.0
 4.0
 4.0
 9.0
 4.0
```

b

In [5]:
```
b = map(x->x==4 ? 1 : -1, b);
btest = map(x->x==4.0 ? 1 : -1, btest)
```

Out[5]:
```
1991-element Array{Int64,1}:
  1
  1
 -1
 -1
 -1
 -1
  1
 -1
  1
  1
  1
  1
  1
  ⋮
  1
 -1
 -1
  1
  1
  1
 -1
  1
  1
  1
 -1
  1
```

c

In [6]:
```
m,n = size(A);
Amean = mean(A, dims = 1);
A = A - ones(m,1)*Amean;
```

In [7]:
```
Astd = std(A, dims = 1);
A = A ./ (ones(m,1)*max.(Astd,1));
```

The way the variance is calculated and defined is to quantify the statistical differences of the values from the mean. It will be a meaningless quantity if we remove variance before we remove the mean. Secondly, removing variance before the mean would lead to large inaccuracies in the data and the statistics of the data.

3

a

In [8]:
```
xls = A \ b;
loss = norm(A*xls-b)^2
```

Out[8]:
```
2136.2898686264043
```

The calculated loss comes out to be 2136.2899. $\therefore \text{loss}(x_{LS}) = ||Ax - b||_2^2$ = 2136.2898686264043.

b

In [9]:
```
A*xls
```

Out[9]:
```
11791-element Array{Float64,1}:
  0.7416235476982628
 -0.9094714343553417
  1.3500678663929255
 -0.8640153436259789
  1.274839474488299
 -0.47048246744351147
  0.6316810350135675
 -1.059392955348037
 -1.0643557378788249
 -1.21288614615639
 -0.592557546811024
  0.41019840230238996
 -0.8210531323703021
  ⋮
  0.8904141739613514
 -0.748092533006772
  0.8045809988984414
 -0.7567025353569596
  0.8006752688246408
 -0.9219972272668431
  0.7934894578446463
 -0.9909252819947398
 -1.278302213235008
  0.7989575699256131
 -1.0323987884547383
 -0.8712251520442909
```

We calculated the value of x_{LS} in part **a**, that satisfies the linear equation $Ax = b$. b contains values +1 and -1: +1 when 4, and -1 otherwise. Looking at the values of Ax_{LS} above, we can define $C_{x_{LS}}(a)$ as:
$C_{x_{LS}}(a) = +1$; if $Ax_{LS} \geq 0$
$C_{x_{LS}}(a) = -1$; if $Ax_{LS} < 0$

In [10]:
```
ib = A*xls;
ib = map(x-> x<0 ? -1 : 1, ib)
```

Out[10]:
```
11791-element Array{Int64,1}:
  1
 -1
  1
 -1
  1
 -1
  1
 -1
 -1
 -1
 -1
  1
 -1
  ⋮
  1
 -1
  1
 -1
  1
 -1
  1
 -1
 -1
  1
 -1
 -1
```

In [11]:
```
tmr = 0; #tmr = Train Misclass Rate
for i in 1:length(ib)
    if b[i] != ib[i] #comparing it with the train dataset
        tmr+=1;
    end
end
tmr = tmr/m
```

Out[11]:
```
0.030786192858960223
```

We find the train misclass rate (x_{LS}) = $\frac{1}{m}\sum_{i=1}^{m} I(C_{x_{LS}}(A_i), b_i) = 0.030786192858960223.$

c

In [12]:
```
mtest, ntest = size(Atest);
Atest = Atest - ones(mtest,1)*Amean;
Atest = Atest./(ones(mtest,1).*max.(Astd,1));
```

Calculating the mean and the standard deviation for the test data would mean that we are trying to normalize the dataset. When in fact we will be using the test dataset as a predictive set for the train set, and thus it should not be normalized.

In [13]:
```
loss_t = norm(Atest*xls - btest)^2
```

Out[13]:
```
862.6339833737284
```

The calculated loss comes out to be 862.634. $\therefore \text{loss}_{test}(x_{LS}) = ||A_{test}x - b_{test}||_2^2$ = 862.6339833737284.

In [14]:
```
ib = Atest*xls
```

Out[14]:
```
1991-element Array{Float64,1}:
  0.6810587685929981
  0.7998472742834444
 -0.44732571159356055
 -0.521621253585898
 -0.8907119128899657
 -0.7471972983155702
  0.9317886397208117
 -0.5233203849989578
  0.7130146649626213
  0.9206736322169555
  0.8633577140512902
  0.38089015484401145
  0.6763181577077987
  ⋮
  0.22088947606294487
 -0.9149659858797987
 -0.4964357269199139
  1.229956736401316
  0.6832835155329782
  0.7822493998440501
 -0.5360441018773913
  1.0162177262338417
  1.094418717993196
  0.9895271154894223
 -0.22844663584838526
  0.6573036486711071
```

In [15]:
```
ib = map(x-> x<0 ? -1 : 1, ib) # applying the same rule
```

Out[15]:
```
1991-element Array{Int64,1}:
  1
  1
 -1
 -1
 -1
 -1
  1
 -1
  1
  1
  1
  1
  1
  ⋮
  1
 -1
 -1
  1
  1
  1
 -1
  1
  1
  1
 -1
  1
```

In [16]:
```
tsmr = 0 # tsmr = Test Misclass Rate
for i in 1:length(ib)
    if ib[i] != btest[i]
        tsmr += 1;
    end
end
tsmr = tsmr/mtest # similar to tmr
```

Out[16]:
```
0.03566047212456052
```

We find the test misclass rate (x_{LS}) = $\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} I(C_{x_{LS}}(A_{test_i}), b_{test_i}) = 0.03566047212456052$, which is very close the train misclass rate.

4. Logistic regression

a. Conceptual questions.

i.

Since log-likelihood is a one-to-one transformation of the general likelihood function (which returns values between 0 and 1), and log is basically an increasing function, mathematically, maximizing the log-likelihood is equivalent to likelihood.

ii.

We are given the loss function as
$f(x) := -log(\Pi_{i=1}^{m}\sigma(a_i^T x)^{b_i}(1-\sigma(a_i^T x))^{1-b_i})$.

To reduce the complexity, let us define a static variable.
Let $k_i = \sigma(a_i^T x)$. We are also given that $\sigma(s) = \frac{1}{1+e^{-s}}$
$. \therefore f(x) = -log(\Pi_{i=1}^{m} k_i^{b_i}(1-k_i)^{1-b_i})$.

$=> f(x) = -log(k_1^{b_1}(1-k_1)^{1-b_1} * k_2^{b_2}(1-k_2)^{1-b_2} * \cdots * k_m^{b_m}(1-k_m)^{1-b_m})$

$=> f(x) = -((b_1)log(k_1) + (1-b_1)log(1-k_1) + (b_2)log(k_2) + (1-b_2)log(1-k_2) + \cdots$
$+ (b_m)log(k_m) + (1-b_m)log(1-k_m))$

$=> f(x) = -\sum_{i=1}^{m}(b_i log(k_i) + (1-b_i)log(1-k_i))$.

Gradient

Let us define the matrix A as:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

Now
$log(k_i) = log(\sigma(a_i^T x)) = log(\frac{1}{1+e^{(-a_i^T x)}}) = -log(1+e^{(-a_i^T x)})$.
$\frac{\partial}{\partial x_j} log(k_i) = \frac{a_{ij}^T e^{-a_{ij}^T x}}{1+e^{-a_{ij}^T x}} = a_{ij}^T(1-k_i)$.

$log(1-k_i) = log(1-\sigma(a_i^T x)) = log(1 - \frac{1}{1+e^{(-a_i^T x)}}) = log(\frac{1+e^{(-a_i^T x)} -1)-1}{1+e^{(-a_i^T x)}}) = -a_i^T x$
$- log(1+e^{(-a_i^T x)})$.
$\frac{\partial}{\partial x_j} log(1-k_i) = -a_{ij}^T + a_{ij}^T(1-k_i) = -a_{ij}^T + a_{ij}^T - a_{ij}^T(k_i) = -a_{ij}^T(k_i)$.

$\therefore \frac{\partial}{\partial x_j} f(x) = -\sum_{i=1}^{m}(b_i a_{ij}^T(1-k_i) - (1-b_i)a_{ij}^T(k_i))$

$= -\sum_{i=1}^{m}(b_i a_{ij}^T - b_i a_{ij}^T k_i - a_{ij}^T k_i + a_{ij}^T k_i b_i)$

$= -\sum_{i=1}^{m}(b_i a_{ij}^T - a_{ij}^T k_i) = -\sum_{i=1}^{m}(a_{ij}^T(b_i - k_i)) = \sum_{i=1}^{m}(a_{ij}^T(k_i - b_i))$.

$$\nabla f(x) = \begin{bmatrix} a_{1,1}^T(k_1 - b_1) & a_{1,2}^T(k_1 - b_1) & \dots & a_{1,n}^T(k_1 - b_1) \\ a_{2,1}^T(k_2 - b_2) & a_{2,2}^T(k_2 - b_2) & \dots & a_{2,n}^T(k_1 - b_1) \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1}^T(k_m - b_m) & a_{m,2}^T(k_m - b_m) & \dots & a_{m,n}^T(k_m - b_m) \end{bmatrix}$$

$\therefore \nabla f(x) = A^T(k - b)$

Hessian

$\frac{\partial}{\partial x_l} k_i = a_{il}^T(1 - k_i)$

$\therefore \frac{\partial^2}{\partial^2 x_l} f(x) = \sum_{i=1}^{m} a_{ij}^T a_{il}^T k_i(1 - k_i)$

$$\nabla^2 f(x) = \begin{bmatrix} a_{1,1}^T a_{1,1} k_1(1 - k_1) & \dots & \dots \\ \dots & \dots & \dots \\ \vdots & \vdots & \vdots \\ \dots & \dots & a_{m,n}^T a_{m,n} k_m(1 - k_m) \end{bmatrix}$$

For the sake of simplicity, let us define a diagonal matrix Z as:

$$Z = \begin{bmatrix} k_1(1 - k_1) & \dots & \dots & 0 \\ 0 & k_2(1 - k_2) & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & \dots & \dots & k_m(1 - k_m) \end{bmatrix}$$

$\therefore \nabla^2 f(x) = A^T Z A.$

The function is convex. We come to this particular conclusion because the Hessian is a positive definite matrix. The values in Z (diagonals) are always positive, since $k_i = \frac{1}{1+e^{-a_i^T x}} \Rightarrow 0 \leq k_i \leq 1.$

b. Coding questions.

i.

In [17]:
```
trainX = float64.(read(file, "trainX"))
trainY = float64.(read(file, "trainY"))
testX = float64.(read(file, "testX"))
testY = float64.(read(file, "testY"));
idx4 = trainY .== 4
idx9 = trainY .== 9
idx = idx4 + idx9
idx = findall(x->x == 1, idx[1,:])

A = trainX[idx,:]
b = trainY[idx]

teidx4 = testY .== 4;
teidx9 = testY .== 9;
teidx = teidx4 + teidx9;
teidx = findall(x->x == 1, teidx[1,:]);

Atest = testX[teidx,:];
btest = testY[teidx];

b = map(x->(x==4 ? 1 : -1), b);
btest = map(x->(x==4 ? 1 : -1), btest);

Amean = mean(A, dims=1);
A = A - ones(m,1)*Amean;

Astd = std(A, dims=1);
A = A ./ (ones(m,1)*max.(Astd,1));
```

In [18]:
```julia
b = (b.+1)/2
```

Out[18]:
```
11791-element Array{Float64,1}:
 1.0
 0.0
 1.0
 0.0
 1.0
 0.0
 1.0
 0.0
 0.0
 0.0
 0.0
 1.0
 0.0
 ⋮
 1.0
 0.0
 1.0
 0.0
 1.0
 0.0
 1.0
 0.0
 0.0
 1.0
 0.0
 0.0
```

ii.

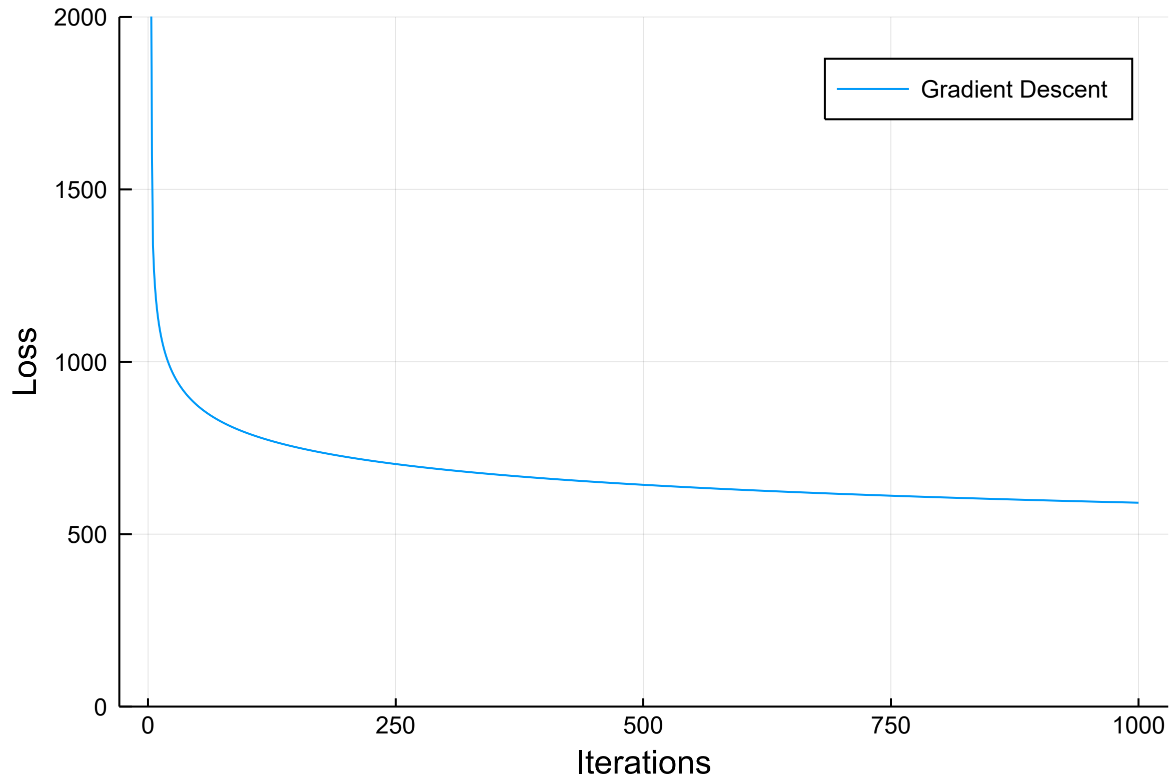
In [19]:
```
sigmoid(s) = 1/(1+exp(-s));
x_0 = zeros(n,1);
alpha = 1/m;
loss = [];

# Gradient.
function gradient(Theta, X, Y)
    m = length(Y)
    H = map(s -> sigmoid(s), (Theta' * X')')
    return (X'*H - X'*Y)
end


# Cost.
function cost(Theta, X, Y)
    m = length(Y)
    H = map(s -> sigmoid(s), (Theta' * X')')
    val = sum((-Y)'*log.(H) - (1 .- Y)'*log.(1 .- H))
    val = 0
    for i in 1:length(Y)
        alp = Theta'*X[i,:]
        s = 1/(1+exp(-alp[1]))
        x = max(1-s,0.000000000001)
        s = max(s, 0.000000000001)

        val += (-Y[i]*log(s)-(1-Y[i])*log(x))
    end
    display(val);
    push!(loss,val)
end

# Gradient Descent.
function gradientDescent(X, Y, Theta, alpha, n)
    m = length(Y)
    i = 1;
    while i <= n
        cost(Theta, X,Y)
        g = gradient(Theta, X, Y)
        Theta = Theta - alpha * g
        i += 1
    end
    return Theta
end


Theta = gradientDescent(A, b, x_0, alpha, 1000)
```

```
8172.898405983718

5316.934265104132

2176.0586239680365

1600.7522556873666

1338.8431246149414
```

```
1266.9168408756098

1218.556281963525

1182.198226390548

1153.3193566168038

1129.4764388563835

1109.2211807053409
```

In [20]:
```
plot(loss, label = "Gradient Descent",
    xlabel = "Iterations", ylabel = "Loss", ylims = (0,2000))
```

Out[20]:



In [21]:
```julia
ib = A*Theta
```

Out[21]:
```
11791×1 Array{Float64,2}:
  10.240508502454796
  -8.047382293737916
  15.606606123635723
 -13.832936000963542
  11.746530604836012
  -4.460402928410925
   4.53562590797375
 -15.605059509699345
  -9.319669079036744
 -14.027355235277746
  -9.313491625461511
   1.106672098988787
 -11.76847322308295
   ⋮
  12.569498949843869
 -11.359625528717723
   7.527746250367656
 -11.207472886524446
   9.90637372873736
 -10.180415639600186
  13.04457571880799
 -14.534714815899124
 -17.518431171086256
   9.39515540029565
  -9.107280485258013
  -7.918018882232677
```

We see the values fluctuating between positive and negative. So we may use the condition to converge.

In [22]:
```
ib = map(x-> x >= 0 ? 1 : 0, ib) # condition: if A*Theta < 0, then 0.
```

Out[22]:
```
11791×1 Array{Int64,2}:
 1
 0
 1
 0
 1
 0
 1
 0
 0
 0
 0
 1
 0
 ⋮
 1
 0
 1
 0
 1
 0
 1
 0
 0
 1
 0
 0
```

In [23]:
```
tmr2 = 0
for i in 1:m
    if(ib[i] != b[i])
       tmr2 += 1;
    end
end

tmr2 = tmr2/m
```

Out[23]:
```
0.018403867356458315
```

The training missclass rate using this particular condition was found to be 0.018403867356458315.

In [24]:
```
mtest, ntest = size(Atest);
Atest = Atest - ones(mtest,1)*Amean;
Atest = Atest./(ones(mtest,1).*max.(Astd,1));
```

In [25]:
```julia
ib = Atest*Theta
```

Out[25]:
```
1991×1 Array{Float64,2}:
  4.808259906019611
  6.6148038753964205
 -9.161800646313825
 -6.04935761416325
 -9.49544424257222
 -6.5825743647066215
  7.948716762263889
 -2.8344299212414206
  5.542890786971411
  8.278967064532027
 14.953463992325057
  3.53957092915113
  4.71953603107517
  ⋮
  8.433135482874954
 -8.314743384365903
 -3.6423887825312344
 13.877723915694768
  6.137954182262305
  6.254392175325377
 -4.405427764305195
 11.369529482777347
  9.094713716893294
  9.83686685197192
 -2.2702289490006744
  5.7317837103721825
```

In [26]:
```
ib = map(x-> x >= 0 ? 1 : 0, ib) # condition: if Atest*Theta < 0, then 0.
```

Out[26]:
```
1991×1 Array{Int64,2}:
 1
 1
 0
 0
 0
 0
 1
 0
 1
 1
 1
 1
 1
 ⋮
 1
 0
 0
 1
 1
 1
 0
 1
 1
 1
 0
 1
```

In [27]:
```
tsmr2 = 0
for i in 1:mtest
    if(ib[i] != btest[i])
       tsmr2 += 1;
    end
end

tsmr2 = tsmr2/mtest
```

Out[27]:
```
0.5238573581115018
```

iii.

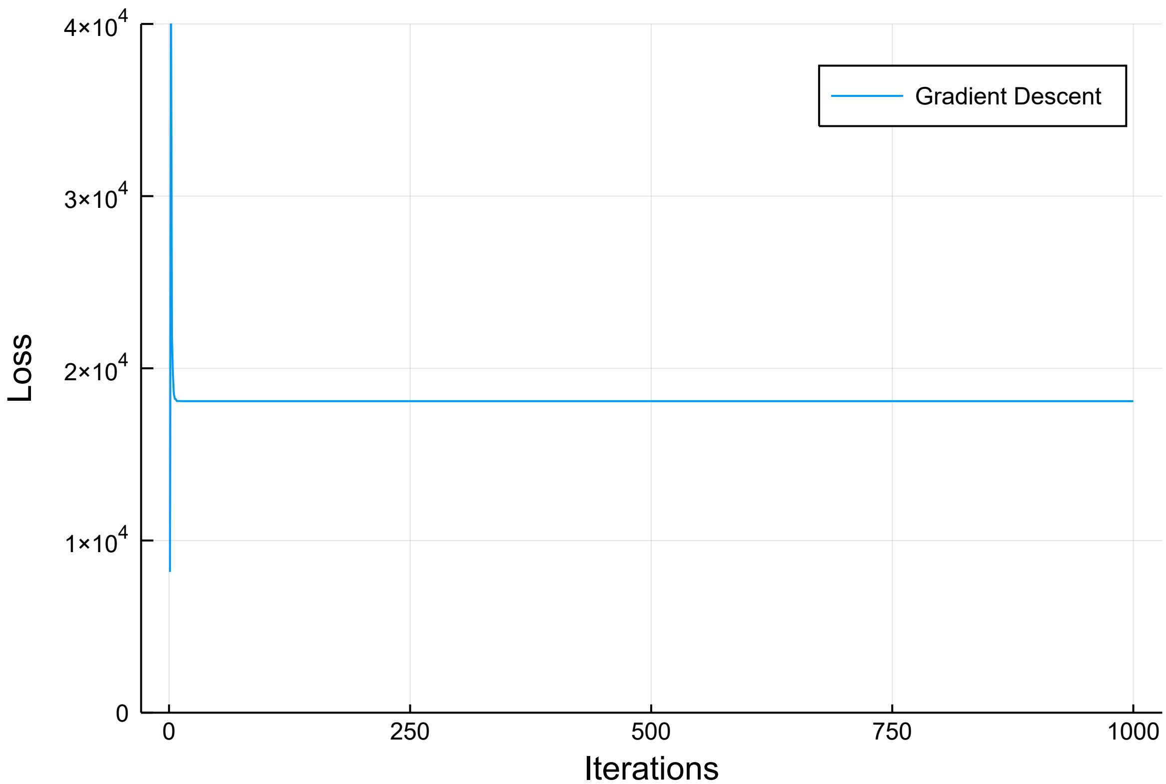
In [28]:
```
x_0 = zeros(n,1);
alpha = 1/2;
loss = [];
t = 1;

# Gradient.
function gradient(Theta, X, Y)
    m = length(Y)
    H = map(s -> sigmoid(s), (Theta' * X')')
    return (X'*H - X'*Y)
end


# Cost.
function cost(Theta, X, Y)
    m = length(Y)
    H = map(s -> sigmoid(s), (Theta' * X')')
    val = sum((-Y)'*log.(H) - (1 .- Y)'*log.(1 .- H))
    val = 0
    for i in 1:length(Y)
        alp = Theta'*X[i,:]
        s = 1/(1+exp(-alp[1]))
        x = max(1-s,0.000000000001)
        s = max(s, 0.000000000001)

        val += (-Y[i]*log(s)-(1-Y[i])*log(x))
    end
    display(val);
    push!(loss,val)
end


# Gradient Descent.
function gradientDescent(X, Y, Theta, alpha, t, n)
    m = length(Y)
    i = 1;
    while i <= n
        cost(Theta, X,Y)
        g = gradient(Theta, X, Y)
        t = alpha * t
        Theta -= (t/2) * g
        i += 1
    end
    return Theta
end


Theta = gradientDescent(A, b, x_0, alpha, t, 1000)
```

```
8172.898405983718

43476.431546787535

21902.279678716674

19672.857197643705
```

```
18512.994119378964

18234.435391918152

18205.907219680645

18105.994801616038

18095.675122190827

18098.318831781733

18098.318831086843
```

In [40]:
```
plot(loss, label = "Gradient Descent",
   xlabel = "Iterations", ylabel = "Loss", ylims = (0,40000))
```

Out[40]:



In [30]:
```julia
ib = A * Theta
```

Out[30]:
```
11791×1 Array{Float64,2}:
  17223.449364567325
 -14259.227377163084
  34565.362836322216
 -20169.127877582196
  30703.716090605627
  -1186.3786504056889
   3284.9505097418737
 -22942.992411173684
 -21441.625877501087
 -24725.636230586922
  -7718.651718016193
   6695.559151311971
 -12569.752208466998
      ⋮
  26741.482586520277
 -19146.700496326896
  20283.87797001591
 -13039.299662598798
  16009.752821981045
 -16629.00733913455
  24482.820883334694
 -18269.025331920235
 -26175.68264666567
  23587.890997368027
 -17272.843564152623
 -16407.786686269304
```

In [31]:
```
ib = map(x-> x >= 0 ? 1 : 0, ib) # condition: if A*Theta < 0, then 0.
```
Out[31]:
```
11791×1 Array{Int64,2}:
 1
 0
 1
 0
 1
 0
 1
 0
 0
 0
 0
 1
 0
 ⋮
 1
 0
 1
 0
 1
 0
 1
 0
 0
 1
 0
 0
```

In [32]:
```
tmr3 = 0
for i in 1:m
    if(ib[i] != b[i])
       tmr3 += 1;
    end
end

tmr3 = tmr3/m
```
Out[32]:
```
0.05555084386396404
```

Training Misclassification Rate = 0.05555084386396404

In [33]:
```
ib = Atest*Theta
```

Out[33]:
```
1991×1 Array{Float64,2}:
  12918.115298543991
  17036.353309298018
 -16376.35025759276
 -21869.989001514794
 -13901.368183981274
 -11801.741699841814
  14340.027866255223
 -19834.453424919684
   9702.332420176605
  13961.949872583024
  35530.65868077357
  10769.414347174994
   4315.676167380155
      ⋮
  14604.800087914897
 -20850.760807506198
  -2777.1575871628847
  35521.90985401843
   6749.445715260302
  17726.291393616975
  -3515.166937439796
  28144.770330900592
  18190.842272292204
  16439.507674885437
  -1121.18440556625
   3264.5104104910147
```

In [34]:
```
ib = map(x-> x >= 0 ? 1 : 0, ib) # condition: if Atest*Theta < 0, then 0.
```

Out[34]:
```
1991×1 Array{Int64,2}:
 1
 1
 0
 0
 0
 0
 1
 0
 1
 1
 1
 1
 1
 ⋮
 1
 0
 0
 1
 1
 1
 0
 1
 1
 1
 0
 1
```

In [35]:
```
tsmr3 = 0
for i in 1:mtest
    if(ib[i] != btest[i])
       tsmr3 += 1;
    end
end

tsmr3 = tsmr3/mtest
```

Out[35]:
```
0.5374183827222502
```

Testing Misclassification Rate = 0.5374183827222502.

iv.

Eventhough the testing misclassification rates are very similar in both of the methods, we see a significant difference in training misclassification rates. And considering the loss plot of the two, it is safe to say that the constant step size leads to a better result.

v.

The training misclassification rate on the constant sized logistic model is less than the linear model. Although, it is not better by much, and testing misclassification rate is worse. So, the gain was not worth it.