Function Tutorial

Julia is a high-level, high-performance programming language for scientific computing. Its machine learning libraries are not as expansive or developed as Python yet, hence for my final project I implemented various machine learning methods in Julia and demonstrate their usage in this notebook. Below each function, I describe the method it performs and the details of its input arguments. Most methods solve for parameters of a model, thus prediction functions are generated afterwards corresponding to the learned parameters. Lastly, the hyperparameters I use here are mostly random and can be tuned for optimal performance.

In [38]:
```julia
# Load the source code for the functions
include("./RG_functions.jl");
```

Logistic Regression

solve_LR_coef(X,y,b; LIMIT) Logistic Regression Computes parameters b for logistic regression using the Newton-Rhapson Algorithm. X must have 1's appended as first column. y must have 1,0 encoding.

Create data that is approximately linearly separable.

In [17]:
```julia
X = rand(500,2)
y = zeros(500)
for i=1:size(X,1)
    if X[i,2] > 1.5*X[i,1] - .25
        y[i] = 1.0
    else
        y[i] = 0.0
    end
end

f(x) = 1.5*x - .25

x = LinRange(1/5,4/5,10)
x_noise_class1 = f.(x) + 1/5 * (rand(10) .- 0.5)
x_noise_class2 = f.(x) + 1/5 * (rand(10) .- 0.5)
Xx = vcat(X,hcat(x,x_noise_class1), hcat(x,x_noise_class2))
X = hcat(ones(size(Xx,1),1),Xx)    # append columns of 1's
y = vcat(y,ones(10),zeros(10));
```

Solve for parameters and create prediction function.

In [20]:
```julia
n,p = size(X)
b = zeros(p)
b = solve_LR_coef(X,y,b)
prob(x, β) = 1 / (1 + exp(-(dot(x,β) )))

# Prediction function
pred(x,b) = round(prob(x,b));
```

```
Number of iterations: 7.
```

Report misclassification error.

In [21]:
```julia
misclass = 0
for i = 1:n
    if pred(X[i,:],b) != y[i]
        misclass += 1
    end
end
println("$misclass out of $n training observations misclassified.")
```

```
520 out of 520 training observations misclassified.
```

Kernel Ridge

kernel_ridge(X,y,λ,K) Kernel Ridge Regression Computes parameters α for the kernel ridge estimator with kernel function K and hyperparameter λ.

Generate data for regression.

In [22]:
```julia
X = transpose(BostonHousing.features())
y = transpose(BostonHousing.targets())
n,p = size(X);
```

Compute parameters for two different basis functions and create prediction function.

In [23]:
```julia
# Radial Basis Function
s = 10
K_rbf(x,z) = exp(-(norm(x-z)^2) / (2*(s^2)))
# Polynomial Basis Function
c = 1; d = 2
K_poly(x,z) = (c + dot(x,z))^d

λ = 1
α_rbf = kernel_ridge(X,y,λ,K_rbf)
α_poly = kernel_ridge(X,y,λ,K_poly)

# Prediction function
pred_kernel_ridge(x,K,X,α) = sum( K(x,X[i,:]) * α[i] for i = 1:size(X,1));
```

Report mean squared error for each basis function.

In [10]:
```julia
# Mean Squared Error
y_preds_rbf = zeros(n)
y_preds_poly = zeros(n)
for i = 1:n
    y_preds_rbf[i] = pred_kernel_ridge(X[i,:],K_rbf,X,α_rbf)
    y_preds_poly[i] = pred_kernel_ridge(X[i,:],K_poly,X,α_poly)
end
println("Mean square error with RBF kernel: $(Statistics.mean( (y_preds_rbf .- y).^2))")
println("Mean square error with polynomial kernel: $(Statistics.mean( (y_preds_poly .- y).^2))")
```

```
Mean square error with RBF kernel: 48.58030241248382
Mean square error with polynomial kernel: 6.363275739611623
```

Proximal Gradient Descent

prox_grad_desc(X,y,β,λ; LIMIT) Proximal Gradient Descent Computes parameters β given initial guess for least squares regression with ℓ-1 penalty. Has hyperparameter λ.

Generate data for regression and center it.

In [25]:
```julia
# Load data
X = transpose(BostonHousing.features())
y = transpose(BostonHousing.targets())
n,p = size(X)
# Center y and estimate β_0
y_centered = y .- Statistics.mean(y)
β_0 = Statistics.mean(y)
# Create matrix of centered X columns
X_centered = zeros(n,p)
for j in 1:p
    X_centered[:,j] = X[:,j] .- Statistics.mean(X[:,j])
end
```

Compute parameters and create prediction function.

In [26]:
```julia
# Initialize β
β_init = zeros(size(X_centered,2))
λ = 10000

β = prox_grad_desc(X_centered, y_centered, β_init, λ)
println("$β")

# Prediction function
pred(X,β,β_0) = β_0 .+ X * β;
```

```
Number of iterations: 1000.
Max iterations reached.
Real[0; 0.0381765577924749; 0; 0; 0; 0; 0; 0; 0; -0.016327374889328842; 0; 0.010874369828524572; -0.1857394401823319]
```

Report mean squared error.

In [13]:
```julia
y_pred = pred(X,β,β_0)

MSE = Statistics.mean((y - y_pred).^2)
println("Mean squared error: $MSE")
```

```
Mean squared error: 74.18408330593503
```

Elastic Net

elastic_net(X,y,β,λ,α; LIMIT) Elastic Net Computes parameters β given initial guess for least squares regression with elastic net penalty. Uses soft-thresholding function update derived in Homework 2. Hyperparameters are λ and α.

Generate data for regression and center it.

In [27]:
```julia
# Load data
X = transpose(BostonHousing.features())
y = transpose(BostonHousing.targets())
n,p = size(X)
# Center y and estimate β_0
y_centered = y .- Statistics.mean(y)
β_0 = Statistics.mean(y)
# Create matrix of centered X columns
X_centered = zeros(n,p)
for j in 1:p
    X_centered[:,j] = X[:,j] .- Statistics.mean(X[:,j])
end
```

Compute parameters and create prediction function.

In [28]:
```julia
# Initialize β
β_init = zeros(size(X_centered,2))
λ = 10000
α = 0.1   # 1 means lasso, 0 means ridge

β = elastic_net(X_centered, y_centered, β_init, λ, α)
println("$β")

# Prediction function
pred(X,β,β_0) = β_0 .+ X * β;
```

```
Number of iterations: 1000.
Max iterations reached.
Real[-0.041753603120333466; 0.05502475975765802; -0.019075219745038403; 0; 0; 0.015521427968780603; 0.0007620910771608188; -0.009899700036271004; 0.03733115888685481; -0.012845543597233947; -0.04954615426492346; 0.010518427530508552; -0.32525422243992397]
```

Report mean squared error.

In [18]:
```julia
y_pred = pred(X,β,β_0)

MSE = Statistics.mean((y - y_pred).^2)
println("Mean squared error: $MSE")
```

```
Mean squared error: 78.37089939829339
```

(Linear) Support Vector Machines

SVM(X,y,C) Support Vector Machines Computes parameters β_0, β and the margins ξ for linear SVMs. Uses Gurobi to solve the optimization problem. Has hyperparameter C. y must have -1,1 encoding.

Create data that is approximately linearly separable.

In [29]:
```julia
# Create linear boundary data
X = rand(500,2)
y = zeros(500)
for i=1:size(X,1)
    if X[i,2] > 1.5*X[i,1] - .25
        y[i] = 1
    else
        y[i] = -1
    end
end

f(x) = 1.5*x - .25

x = LinRange(1/5,4/5,10)
x_noise_class1 = f.(x) + 1/5 * (rand(10) .- 0.5)
x_noise_class2 = f.(x) + 1/5 * (rand(10) .- 0.5)
Xx = vcat(X,hcat(x,x_noise_class1), hcat(x,x_noise_class2))
X = hcat(ones(size(Xx,1),1),Xx)    # append columns of 1's
y = vcat(y,ones(10),-ones(10));
```

Compute parameters and create prediction function.

In [32]:
```julia
# Set parameter
C = 100
# Solve
(β_0, β, ξ) = SVM(X,y,C)

# Prediction function
SVM_classifier(x,β_0,β) = sign(x'*β + β_0);
```

```
Academic license - for non-commercial use only - expires 2022-08-24
Gurobi Optimizer version 9.1.2 build v9.1.2rc0 (win64)
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads
Optimize a model with 521 rows, 524 columns and 3120 nonzeros
Model fingerprint: 0xda89d0f5
Model has 3 quadratic objective terms
Coefficient statistics:
  Matrix range     [2e-03, 1e+00]
  Objective range  [0e+00, 0e+00]
  QObjective range [2e+00, 2e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+00, 1e+02]
Presolve time: 0.00s
Presolved: 521 rows, 524 columns, 3120 nonzeros
Presolved model has 3 quadratic objective terms
Ordering time: 0.00s

Barrier statistics:
 Dense cols : 4
 Free vars  : 4
 AA' NZ     : 2.600e+03
 Factor NZ  : 3.513e+03
 Factor Ops : 3.024e+04 (less than 1 second per iteration)
 Threads    : 1

                  Objective                Residual
Iter       Primal          Dual         Primal    Dual     Compl     Time
   0   6.75733361e-10 -6.75733361e-10  5.20e+05 2.40e-05  9.99e+05     0s
   1   4.76560919e+06 -4.76338184e+06  1.63e+05 3.10e+02  3.25e+05     0s
   2   8.37673916e+06 -8.41762118e+06  7.11e+03 9.50e+00  2.92e+04     0s
   3   2.13291506e+06 -2.22586471e+06  6.73e+02 3.11e-08  5.37e+03     0s
   4   7.00034555e+05 -8.95564821e+05  1.98e+02 9.14e-09  2.07e+03     0s
   5   1.69505516e+05 -4.30578807e+05  4.62e+01 2.67e-09  7.27e+02     0s
   6   5.77120568e+04 -1.16600339e+05  5.67e+00 8.85e-10  1.72e+02     0s
   7   4.50695345e+04 -1.06904893e+05  4.48e+00 7.20e-10  1.50e+02     0s
   8   1.94813853e+04 -4.92865212e+04  1.18e+00 1.91e-10  6.66e+01     0s
   9   8.16442481e+03 -1.92590056e+04  2.18e-01 5.94e-11  2.64e+01     0s
  10   5.00106634e+03 -1.15563921e+04  1.10e-01 3.11e-11  1.59e+01     0s
  11   1.84614673e+03 -4.44910821e+03  2.18e-02 2.34e-11  6.05e+00     0s
  12   7.29009692e+02 -1.50157176e+03  4.10e-03 4.32e-12  2.14e+00     0s
  13   3.63375202e+02 -5.41608733e+02  1.16e-03 3.39e-12  8.70e-01     0s
  14   1.97020352e+02 -2.26549880e+02  3.10e-04 1.80e-12  4.07e-01     0s
  15   1.17345653e+02 -6.66984945e+01  7.05e-05 1.88e-13  1.77e-01     0s
  16   8.79248816e+01 -1.20280772e+01  3.29e-05 2.16e-13  9.61e-02     0s
  17   7.22677323e+01  1.33509420e+01  1.72e-05 1.59e-13  5.67e-02     0s
  18   6.30054125e+01  2.73602390e+01  1.01e-05 8.99e-14  3.43e-02     0s
  19   5.53136514e+01  3.62735447e+01  4.02e-06 1.29e-14  1.83e-02     0s
  20   5.14597277e+01  4.03112094e+01  1.98e-06 1.93e-14  1.07e-02     0s
  21   4.91054838e+01  4.30179629e+01  8.59e-07 5.33e-15  5.85e-03     0s
  22   4.74162772e+01  4.49210812e+01  2.64e-07 1.38e-14  2.40e-03     0s
  23   4.66637331e+01  4.57205427e+01  5.71e-08 5.88e-15  9.07e-04     0s
  24   4.62947062e+01  4.61020599e+01  5.73e-09 8.88e-15  1.85e-04     0s
  25   4.61992544e+01  4.61973003e+01  4.03e-11 5.88e-15  1.88e-06     0s
  26   4.61983023e+01  4.61983003e+01  7.39e-13 4.44e-15  1.90e-09     0s
  27   4.61983013e+01  4.61983013e+01  1.42e-13 2.66e-15  1.90e-12     0s

Barrier solved model in 27 iterations and 0.01 seconds
Optimal objective 4.61983013e+01


User-callback calls 97, time in user-callback 0.00 sec
```

Report misclassification error.

In [33]:
```julia
# Report error
misclass = 0
for i = 1:n
    if SVM_classifier(X[i,:], β_0, β) != y[i]
        misclass += 1
    end
end
println("$misclass out of $n training observations misclassified.")
```

```
16 out of 506 training observations misclassified.
```

Kernel Support Vector Machines

kernel_SVM(X,y,C,K) Kernel Support Vector Machines Computers paramters α for Kernel SVMs with kernel function K and hyperparameter C. Uses Gurobi to solve the optimization problem. y must have -1,1 encoding.

Generate data that is approximately separable with a quadratic boundary.

In [34]:
```julia
# Quadratic Boundary Data
X = rand(500,2)
y = zeros(500)
for i=1:size(X,1)
    if X[i,2] > -(X[i,1]+.5)*(X[i,1]-1)
        y[i] = 1
    else
        y[i] = -1
    end
end

f(x) = -(x+0.5)*(x-1)

x = LinRange(0,1,10)
x_noise_class1 = f.(x) + 1/5 * (rand(10) .- 0.5)
x_noise_class2 = f.(x) + 1/5 * (rand(10) .- 0.5)
X = vcat(X,hcat(x,x_noise_class1), hcat(x,x_noise_class2))
y = vcat(y,ones(10),-ones(10));
```

Compute parameters for two different basis functions and create prediction function.

In [35]:
```julia
# Radial Basis Function
s = 100
K_rbf(x,z) = exp(-(norm(x-z)^2) / (2*(s^2)))
# Polynomial Basis Function
c = 0; d = 2
K_poly(x,z) = (c + dot(x,z))^d

C = 10
α_rbf = kernel_SVM(X,y,C,K_rbf)
α_poly = kernel_SVM(X,y,C,K_poly)

# find index of max α component, let that be k
k_rbf = argmax(α_rbf)
k_poly = argmax(α_poly)

b_rbf = y[k_rbf] - sum( α_rbf[i]*y[i]*K_rbf(X[k_rbf,:],X[i,:]) for i = 1:n)
b_poly = y[k_poly] - sum( α_poly[i]*y[i]*K_poly(X[k_poly,:],X[i,:]) for i = 1:n)

# Prediction function
kernel_SVM_classifier(x,K,X,α,b) = sign(sum( α[i]*y[i]*K(x,X[i,:]) for i = 1:size(X,1)) + b);
```

```
Academic license - for non-commercial use only - expires 2022-08-24
Gurobi Optimizer version 9.1.2 build v9.1.2rc0 (win64)
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads
Optimize a model with 1 rows, 520 columns and 520 nonzeros
Model fingerprint: 0xb086e786
Model has 135460 quadratic objective terms
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 1e+00]
  QObjective range [1e+00, 2e+00]
  Bounds range     [1e+01, 1e+01]
  RHS range        [0e+00, 0e+00]
Presolve time: 0.01s
Presolved: 1 rows, 520 columns, 520 nonzeros
Presolved model has 135460 quadratic objective terms
Ordering time: 0.00s

Barrier statistics:
 Free vars  : 6
 AA' NZ     : 2.100e+01
 Factor NZ  : 2.800e+01
 Factor Ops : 1.400e+02 (less than 1 second per iteration)
 Threads    : 1

                  Objective                Residual
Iter       Primal          Dual         Primal    Dual     Compl     Time
   0   2.08088975e+06  1.30260000e+06  8.40e+04 1.54e-03  1.00e+06     0s
   1   2.51912508e+03  1.29925501e+06  8.40e-02 1.54e-09  1.25e+03     0s
   2   2.53022010e+03  6.24872767e+03  1.21e-04 2.22e-12  3.58e+00     0s
   3   4.30701455e+03  4.77310601e+03  4.51e-06 8.24e-14  4.48e-01     0s
   4   4.31212450e+03  4.35412185e+03  3.36e-07 6.15e-15  4.04e-02     0s
   5   4.31476546e+03  4.35025443e+03  2.57e-07 4.70e-15  3.41e-02     0s
   6   4.31884948e+03  4.34238613e+03  1.40e-07 2.66e-15  2.26e-02     0s
   7   4.32297946e+03  4.33499722e+03  6.53e-08 1.78e-15  1.16e-02     0s
   8   4.32578871e+03  4.33140802e+03  2.91e-08 1.78e-15  5.40e-03     0s
   9   4.32693901e+03  4.33038325e+03  1.71e-08 1.78e-15  3.31e-03     0s
  10   4.32761481e+03  4.32987080e+03  1.04e-08 1.78e-15  2.17e-03     0s
  11   4.32803919e+03  4.32959075e+03  6.23e-09 1.78e-15  1.49e-03     0s
  12   4.32834741e+03  4.32939462e+03  3.75e-09 1.78e-15  1.01e-03     0s
  13   4.32854243e+03  4.32926475e+03  2.25e-09 1.78e-15  6.95e-04     0s
  14   4.32875741e+03  4.32913701e+03  5.96e-10 1.78e-15  3.65e-04     0s
  15   4.32895452e+03  4.32897084e+03  3.38e-14 3.55e-15  1.57e-05     0s
  16   4.32896267e+03  4.32896269e+03  3.38e-14 2.66e-15  1.83e-08     0s
  17   4.32896268e+03  4.32896268e+03  7.99e-15 2.66e-15  1.84e-11     0s

Barrier solved model in 17 iterations and 0.02 seconds
Optimal objective 4.32896268e+03


User-callback calls 78, time in user-callback 0.00 sec
Academic license - for non-commercial use only - expires 2022-08-24
Gurobi Optimizer version 9.1.2 build v9.1.2rc0 (win64)
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads
Optimize a model with 1 rows, 520 columns and 520 nonzeros
Model fingerprint: 0xb37c290e
Model has 135460 quadratic objective terms
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 1e+00]
  QObjective range [1e-08, 7e+00]
  Bounds range     [1e+01, 1e+01]
  RHS range        [0e+00, 0e+00]
Presolve time: 0.01s
Presolved: 1 rows, 520 columns, 520 nonzeros
Presolved model has 135460 quadratic objective terms
Ordering time: 0.00s

Barrier statistics:
 Free vars  : 3
 AA' NZ     : 6.000e+00
 Factor NZ  : 1.000e+01
 Factor Ops : 3.000e+01 (less than 1 second per iteration)
 Threads    : 1

                  Objective                Residual
Iter       Primal          Dual         Primal    Dual     Compl     Time
   0   1.22465487e+06  3.22760000e+06  2.04e+05 2.32e-06  1.00e+06     0s
   1   4.21729873e+03  3.18545957e+06  6.47e+02 7.36e-09  6.21e+03     0s
   2  -4.92356958e+04  7.21571408e+05  6.47e-04 5.12e-13  7.41e+02     0s
   3  -1.25698537e+04  4.09278690e+04  2.47e-05 2.84e-13  5.14e+01     0s
   4  -1.61231612e+03  5.08726391e+03  5.76e-07 1.03e-13  6.44e+00     0s
   5  -3.56010790e+02  2.69745265e+03  2.07e-07 4.90e-14  2.94e+00     0s
   6   4.88821466e-01  2.06782307e+03  1.27e-07 2.85e-14  1.99e+00     0s
   7   2.42133289e+02  1.61680909e+03  7.71e-08 2.44e-14  1.32e+00     0s
   8   4.16756576e+02  1.28057572e+03  3.74e-08 1.40e-14  8.31e-01     0s
   9   5.50481981e+02  1.06929974e+03  1.85e-08 1.06e-14  4.99e-01     0s
  10   6.12918865e+02  9.85585266e+02  1.17e-08 7.31e-15  3.58e-01     0s
  11   6.81287750e+02  9.02028660e+02  5.61e-09 7.05e-15  2.12e-01     0s
  12   7.17165438e+02  8.59328807e+02  3.02e-09 5.93e-15  1.37e-01     0s
  13   7.46290454e+02  8.27772349e+02  1.43e-09 8.44e-15  7.83e-02     0s
  14   7.61274468e+02  8.12639382e+02  7.40e-10 4.78e-15  4.94e-02     0s
  15   7.71828053e+02  8.01780517e+02  3.03e-10 7.61e-15  2.88e-02     0s
  16   7.79937448e+02  7.93953193e+02  3.55e-14 8.88e-15  1.35e-02     0s
  17   7.86103811e+02  7.87224496e+02  4.35e-14 6.57e-15  1.08e-03     0s
  18   7.86665431e+02  7.86670598e+02  3.51e-14 9.45e-15  4.97e-06     0s
  19   7.86668074e+02  7.86668116e+02  2.49e-12 5.35e-15  4.02e-08     0s
  20   7.86668087e+02  7.86668088e+02  1.72e-10 7.68e-15  1.40e-09     0s

Barrier solved model in 20 iterations and 0.02 seconds
Optimal objective 7.86668087e+02


User-callback calls 84, time in user-callback 0.00 sec
```

Report misclassification error.

In [36]:
```julia
misclass_rbf = 0
misclass_poly = 0
for i = 1:n
    if kernel_SVM_classifier(X[i,:], K_rbf,X, α_rbf, b_rbf) != y[i]
        misclass_rbf += 1
    end
    if kernel_SVM_classifier(X[i,:], K_poly,X, α_poly, b_poly) != y[i]
        misclass_poly += 1
    end
end
println("RBF kernel: $misclass_rbf out of $n training observations misclassified.")
println("Polynomial kernel: $misclass_poly out of $n training observations misclassified.")
```

```
RBF kernel: 298 out of 506 training observations misclassified.
Polynomial kernel: 284 out of 506 training observations misclassified.
```

Bootstrapping Procedure

bootstrapper(X,y,B) Boostrapping Procedure Returns B bootstrapped samples of input data X and y. Samples are stored in an array. The indices used and not used in each sample are also returned for out-of-bag error calculation.

In [40]:
```julia
# Sample data
X = [ones(5)'; 2*ones(5)'; 3*ones(5)'; 4*ones(5)']
y = [1.0; 2.0; 3.0; 4.0]
# Number of boostrapped samples
B = 10

b_samples, ind_used, ind_not_used = bootstrapper(X,y,B)

# Observations and indices for the i-th bootstrapped sample
i = 1
X_b, y_b = b_samples[i]
println("X_$i = $X_b\ny_$i = $y_b")
println("Indices used: $(ind_used[i])")
println("Indices not used: $(ind_not_used[i])")
```

```
X_1 = [3.0 3.0 3.0 3.0 3.0; 4.0 4.0 4.0 4.0 4.0; 3.0 3.0 3.0 3.0 3.0; 2.0 2.0 2.0 2.0 2.0]
y_1 = [3.0, 4.0, 3.0, 2.0]
Indices used: Set(Any[4, 2, 3])
Indices not used: Set([1])
```

Neural Networks

train_neural_network(X,y,num_hidden_layers,size_hidden_layers;h,activation,problem_type,num_classes,num_epochs) Initiate Neural Network Computes weight matrices W and biases b and prints the training error. W and b are initialized in this function. Activation options are 'sigmoid' and 'ReLu'. Problem types are 'classification' and 'regression'. Other parameters settings are number of hidden layers, number of neurons per hidden layer, number of epochs, number of classes (1 if regression), and learning rate h.update_neural_network(X,y,W,b,num_hidden_layers,size_hidden_layers;h,activation,problem_type,num_classes,num_epochs) Update Neural Network Computes weight matrices W and biases b and prints the training error. W and b are passed as arguments in this function. The network settings should be the same as those passed when creating W and b.predict_neural_network(X,W,b,num_hidden_layers,size_hidden_layers;activation,problem_type,num_classes) Predictions via Trained Neural Network Computes predictions to input data X given weights W and biases b. Network settings should be the same as those which were used to learn W and b.

NN Classification

Load MNIST handwritten digit data.

In [2]:
```julia
tr_size = 5000
train_x, train_y = MNIST.traindata(1:tr_size)
te_size = 200
test_x,  test_y  = MNIST.testdata(1:te_size)

X = zeros(tr_size,784)
for i = 1:tr_size
    X[i,:] = reshape(train_x[:,:,i],1,784)
end
y = train_y[1:tr_size]

X_test = zeros(te_size,784)
for i = 1:te_size
    X_test[i,:] = reshape(test_x[:,:,i],1,784)
end
y_test = test_y;
```

Train neural network.

In [3]:
```julia
num_hidden_layers = 2
size_hidden_layers = 8
W,b = train_neural_network(X,y,num_hidden_layers,size_hidden_layers,activation="sigmoid",problem_type="classifi
```

```
Sigmoid used.
Total loss is 20236.307355480145.
Average loss 4.047261471096029 out of 5000 training images.
4507 of 5000 training images misclassified.
```

Update the weights.

In [4]:
```julia
W,b = update_neural_network(X,y,W,b,num_hidden_layers,size_hidden_layers,activation="sigmoid",problem_type="cla
```

```
Sigmoid used.
Total loss is 20236.307355480145.
Average loss 4.047261471096029 out of 5000 training images.
4507 of 5000 training images misclassified.
```

Predict test data.

In [5]:
```julia
y_pred = predict_neural_network(X_test,W,b,num_hidden_layers,size_hidden_layers;activation="sigmoid",problem_ty
```

```
Sigmoid used.
```

NN Regression

In [6]:
```julia
# Regression
X_all = transpose(BostonHousing.features())
X = X_all[1:450,:]
X_test = X_all[451:end,:]
y_all = transpose(BostonHousing.targets())
y = y_all[1:450]
y_test = y_all[451:end];
```

In [7]:
```julia
num_hidden_layers = 2
size_hidden_layers = 10
W,b = train_neural_network(X,y,num_hidden_layers,size_hidden_layers,activation="sigmoid",problem_type="regressi
```

```
Sigmoid used.
Average loss 185.63988444260593 out of 450 training images.
```

In [8]:
```julia
W,b = update_neural_network(X,y,W,b,num_hidden_layers,size_hidden_layers,activation="sigmoid",problem_type="reg
```

```
Sigmoid used.
Average loss 185.63959026676432 out of 450 training images.
```

Predict test data.

In [9]:
```julia
y_pred = predict_neural_network(X_test,W,b,num_hidden_layers,size_hidden_layers;activation="sigmoid",problem_ty
```

```
Sigmoid used.
```

In [ ]: