```julia
using Plots
using LinearAlgebra
using JuMP
using Gurobi
using MLDatasets
import Statistics
import Random

"""
    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.
"""
function solve_LR_coef(X,y,b; LIMIT = 1000)
    n = size(X,1)
    # Probability function
    prob(x, β) = 1 / (1 + exp(-(dot(x,β) )))
    # Gradient of Loss Function
    ∇l(β,X,y,n) = sum( X[i,:] * (y[i] - prob(X[i,:], β)) for i = 1:n)
    # Hessian of Loss Function (not used)
    # Hl(β,X,n) = -sum( X[i,:] * transpose(X[i,:]) * prob(X[i,:], β)*(1-prob(X[i,:], β)) for
i = 1:n )

    global counter = 0
    tol = 1.e-6
    # Gradient descent
    while norm(∇l(b,X,y,n)) > tol && counter < LIMIT
        W = zeros(n,n)
        P = zeros(n)
        for i = 1:n
            W[i,i] = prob(X[i,:], b)*(1 - prob(X[i,:], b))
            P[i] = prob(X[i,:], b)
        end
        approx = W \ (y-P)
        z = X*b + approx
        # z = X*b + inv(W)*(y-P)   # may not be invertible
        b = inv((X'*W*X)) * X'*W*z
        global counter += 1
    end

    println("Number of iterations: $counter.")
    if counter == LIMIT
        println("Max iterations reached.")
    end

    return b
end
```

```julia
"""
    kernel_ridge(X,y,λ,K)
Kernel Ridge Regression
Computes parameters α for the kernel ridge estimator with kernel function K and
hyperparameter λ.
"""
function kernel_ridge(X,y,λ,K)
    n = length(y)
    k = zeros(n,n)
    for i = 1:n
        for j = i:n
            k[i,j] = K(X[i,:],X[j,:])
            k[j,i] = k[i,j]
            # println("$(k[i,j])")
        end
    end
    # Solution α to min (norm(y-k*α))^2 + λ α'*k*α
    α = (k + λ*I) \ y

    # f̂(x*) = sum( K(x*,X[i,:]) * α[i] )
    return α
end


"""
    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.
"""
function prox_grad_desc(X,y,β,λ; LIMIT=1000)
    # Soft-thresholding function
    S(y,l) = begin
        if abs(y) <= l
            return 0
        elseif y > l
            return y - l
        else
            return y + l
        end
    end

    # Solving min 1/2*norm(Y-Xβ,2)^2 + λ*norm(β,1)
    h = 1 / max(eigvals(X'X)...)    # learning rate
    ∇L(β) = -X'*(y-X*β)
    tol = 1.e-6
    counter = 0
    while norm(∇L(β)) > tol && counter < LIMIT
        β = S.(β - h*∇L(β), λ*h)
        counter += 1
```

```julia
        # if counter % 10 == 0
        #     println("Iteration $counter, gradient norm $(norm(∇L(β)))")
        # end
    end

    println("Number of iterations: $counter.")
    if counter == LIMIT
        println("Max iterations reached.")
    end

    return β
end


"""
    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.
"""
function elastic_net(X,y,β,λ,α; LIMIT=1000)
    # Soft-thresholding function
    S(y,l) = begin
        if abs(y) <= l
            return 0
        elseif y > l
            return y - l
        else
            return y + l
        end
    end

    # Solving min 1/2*norm(Y-Xβ,2)^2 + λ*(α*norm(β,1) + (1-α)*norm(β,2)^2)
    η = 1 / max(eigvals(X'X)...)   # learning rate
    ∇L(β) = -X'*(y-X*β) + 2λ*(1-α)*β
    tol = 1.e-6
    counter = 0
    while norm(∇L(β)) > tol && counter < LIMIT
        β = S.(β - η*∇L(β), λ*η*α)
        counter += 1
        # if counter % 10 == 0
        #     println("Iteration $counter, gradient norm $(norm(∇L(β)))")
        # end
    end

    println("Number of iterations: $counter.")
    if counter == LIMIT
        println("Max iterations reached.")
    end
```

```
    return β
end


"""
    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.
"""
function SVM(X,y,C)
    n,p = size(X)
    model = JuMP.Model(JuMP.optimizer_with_attributes(Gurobi.Optimizer, "MIPGap" => .01,
"TimeLimit" => 180))
    JuMP.@variable(model, β_0)
    JuMP.@variable(model, β[1:p])
    JuMP.@variable(model, ξ[1:n] >= 0)
    JuMP.@objective(model, Min, β'*β)
    for i = 1:n
        JuMP.@constraint(model, y[i]*(X[i,:]'*β + β_0) >= 1 - ξ[i])
    end
    JuMP.@constraint(model, sum(ξ[i] for i = 1:n) <= C)

    JuMP.optimize!(model)

    return value(β_0), value.(β), value.(ξ)
end


"""
    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.
"""
function kernel_SVM(X,y,C,K)
    n = size(X,1)
    k = zeros(n,n)
    for i = 1:n
        for j = i:n
            k[i,j] = K(X[i,:],X[j,:])
            k[j,i] = k[i,j]
            # println("$(k[i,j])")
        end
    end

    model = JuMP.Model(JuMP.optimizer_with_attributes(Gurobi.Optimizer, "NonConvex" => 2,
"MIPGap" => .01, "TimeLimit" => 180))
```

```
    JuMP.@variable(model, 0 <= α[1:n] <= C)
    JuMP.@objective(model,
                    Max,
                    sum(α[i] for i = 1:n) - 1/2 * sum( sum( α[i]*α[j]*y[i]*y[j]*k[i,j] for j
= 1:n ) for i = 1:n)
                    )
    JuMP.@constraint(model, sum(α[i]*y[i] for i = 1:n) == 0)

    JuMP.optimize!(model)

    return value.(α)
end


"""
    bootstrapper(X,y,B)
Bootstrapping 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.
"""
function bootstrapper(X,y,B)
    n,p = size(X)
    b_samples = []
    ind_used = []
    ind_not_used = []
    for _ = 1:B
        X_b = zeros(n,p)
        y_b = zeros(n)
        ind_used_b = Set([])
        for i=1:n
            ind = rand(1:n)
            X_b[i,:] = X[ind,:]
            y_b[i] = y[ind]
            push!(ind_used_b, ind)
        end
        ind_all = Set(1:n)
        ind_not_used_b = setdiff(ind_all,ind_used_b)
        push!(b_samples, (X_b,y_b))
        push!(ind_used, ind_used_b)
        push!(ind_not_used, ind_not_used_b)
    end

    return b_samples, ind_used, ind_not_used
end


"""
    train_neural_network(X,y,num_hidden_layers,size_hidden_layers;h,activation,problem_type,n
um_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.
"""
function train_neural_network(X,y,num_hidden_layers,size_hidden_layers;h=0.1,
activation="sigmoid",problem_type="classification",num_classes=1,num_epochs=1)
    # Create activation functions
    if activation == "sigmoid"
        σ = (z) -> 1 / (1+exp(-z))
        Dσ = (z) -> σ(z)^2 * exp(-z)    # gradient with respect to z
        println("Sigmoid used.")
    elseif activation == "ReLU"
        σ = (z) -> max(z,0)
        Dσ = (z) -> begin
            if z >= 0.0
                return 1.0
            else
                return 0.0
            end
        end
        println("ReLU used.")
    end

    # Create appropriate loss function
    if problem_type == "classification"
        y_indicator = []
        classes = vcat(collect(1:9), 0)
        for i = 1:length(y)
            push!(y_indicator, map(k -> y[i] == k, classes))
        end

        L = (y_indicator,a_L_prob) -> -sum(y_indicator[k] * log(a_L_prob[k]) for
k=1:num_classes)
        DL = (y_indicator,a_L_prob) -> [-y_indicator[k] / a_L_prob[k] for k=1:num_classes]
        # These are gonna return vectors
        f = (z) -> [exp(z[j]) / sum(exp(z[k]) for k=1:num_classes) for j=1:num_classes]
        Df = (z) -> [(sum(exp(z[k]) for k=1:num_classes) - exp(z[j]))*exp(z[j]) /
(sum(exp(z[k]) for k=1:num_classes))^2 for j=1:num_classes]
    elseif problem_type == "regression"
        L = (y,a_L) -> norm(y - a_L)^2
        DL = (y,a_L) -> 2*(a_L - y)
        f = (z) -> z
        Df = (z) -> 1
    end

    # Total number of layers
```

```julia
    LL = num_hidden_layers + 2
    # Stepsize
    η = h

    # To store weighted inputs
    z = map(_ -> zeros(size_hidden_layers), 1:num_hidden_layers)
    pushfirst!(z, zeros(size(X,2)))
    push!(z, zeros(num_classes))
    # To store activations
    a = map(_ -> zeros(size_hidden_layers), 1:num_hidden_layers)
    pushfirst!(a, zeros(size(X,2)))
    push!(a, zeros(num_classes))
    # To store errors
    δ = map(_ -> zeros(size_hidden_layers), 1:num_hidden_layers)
    pushfirst!(δ, zeros(size(X,2)))
    push!(δ, zeros(num_classes))

    # Create biases
    b = map(_ -> rand(size_hidden_layers), 1:num_hidden_layers)
    pushfirst!(b, rand(size(X,2)))
    push!(b, rand(num_classes))
    # Create weight parameters
    W = [Matrix{Float64}(I(size(X,2)))]   # store identity for sake
    push!(W, rand(size_hidden_layers, size(X,2)) )   # W[2] = weights from layer 1 to layer 2
    for _ = 1:(num_hidden_layers-1)
        push!(W, rand(size_hidden_layers, size_hidden_layers))
    end
    push!(W, rand(num_classes, size_hidden_layers))

    for _ = 1:num_epochs
        for i = 1:size(X,1)
            # Begin forward pass
            z[1] = X[i,:]   # identical transformation
            a[1] = Vector{Float64}(σ.(z[1]))
            for l = 2:(LL-1) # LL and LL-1
                z[l] = W[l]*a[l-1] + b[l]
                a[l] = Vector{Float64}(σ.(z[l]))
            end
            z[LL] = W[LL]*a[LL-1] + b[LL]
            # a[LL] = f(z[LL])   # Regression requires a broadcasting with a . while
classification does not

            # Begin backward pass
            if problem_type == "classification"
                # δ[LL] = DL(y_indicator[i], a[LL]) .* Dσ.(z[LL])
                a[LL] = f(z[LL])
                δ[LL] = DL(y_indicator[i], a[LL]) .* Df(z[LL])
            elseif problem_type == "regression"
                # δ[LL] = DL(y[i], a[LL][1]) .* Dσ.(z[LL])
                a[LL] = f.(z[LL])
```

```julia
            δ[LL] = DL(y[i], a[LL][1]) .* Df.(z[LL])
        end

        # Compute weighted errors
        for l = (LL-1):-1:2
            δ[l] = W[l+1]' * δ[l+1] .* Vector{Float64}(Dσ.(z[l]))
        end

        # Update weights and biases
        for l = 2:LL
            W[l] = W[l] - η * δ[l] * a[l-1]'
            b[l] = b[l] - η * δ[l]
        end
    end
end

# for l=2:LL
#     println("W_$l is:\n$(W[l])")
#     println("b_$l is:\n$(b[l])")
# end

function f_predictor(x)
    z[1] = x    # identical transformation
    a[1] = Vector{Float64}(σ.(z[1]))
    for l = 2:(LL-1)
        z[l] = W[l]*a[l-1] + b[l]
        a[l] = Vector{Float64}(σ.(z[l]))
    end
    z[LL] = W[LL]*a[LL-1] + b[LL]

    #println("$(a[L])")

    if problem_type == "classification"
        a[LL] = f(z[LL])
        y_pred = argmax(a[LL])
        if y_pred == 10
            y_pred = 0
        end
        return y_pred, a[LL]
    elseif problem_type == "regression"
        a[LL] = f.(z[LL])
        return a[LL][1]  # if regression, final layer has 1 output
    end
end

if problem_type == "classification"
    # loss = [L(y_indicator[i], f_predictor(X[i,:])) for i = 1:size(X,1)]
    y_pred_vec = zeros(size(X,1))
    a_L_vec = map(k -> zeros(num_classes), 1:size(X,1))
    for i = 1:size(X,1)
```

```julia
            y_pred_vec[i], a_L_vec[i] = f_predictor(X[i,:])
        end
        total_loss = sum([L(y_indicator[i], a_L_vec[i]) for i = 1:size(X,1)])
        println("Total loss is $total_loss.")
        ave_loss = Statistics.mean([L(y_indicator[i], a_L_vec[i]) for i = 1:size(X,1)])
        println("Average loss $ave_loss out of $(size(X,1)) training images.")

        # Compute misclassication
        misclass = 0
        for i = 1:size(X,1)
            if y_pred_vec[i] != y[i]
                misclass += 1
            end
        end
        println("$misclass of $(size(X,1)) training images misclassified.")
        return W,b
    elseif problem_type == "regression"
        ave_loss = Statistics.mean([L(y[i], f_predictor(X[i,:])) for i = 1:size(X,1)])
        println("Average loss $ave_loss out of $(size(X,1)) training images.")
        return W,b
    end

end

"""
    update_neural_network(X,y,W,b,num_hidden_layers,size_hidden_layers;h,activation,problem_t
ype,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.
"""
function
update_neural_network(X,y,W,b,num_hidden_layers,size_hidden_layers;h=0.1,activation="sigmoid"
,problem_type="classification",num_classes=1,num_epochs=1)
    # Create activation functions
    if activation == "sigmoid"
        σ = (z) -> 1 / (1+exp(-z))
        Dσ = (z) -> σ(z)^2 * exp(-z)    # gradient with respect to z
        println("Sigmoid used.")
    elseif activation == "ReLU"
        σ = (z) -> max(z,0)
        Dσ = (z) -> begin
            if z >= 0.0
                return 1.0
            else
                return 0.0
            end
        end
        println("ReLU used.")
```

```
    end

    # Create appropriate loss function
    if problem_type == "classification"
        y_indicator = []
        classes = vcat(collect(1:9), 0)
        for i = 1:length(y)
            push!(y_indicator, map(k -> y[i] == k, classes))
        end

        L = (y_indicator,a_L_prob) -> -sum(y_indicator[k] * log(a_L_prob[k]) for
k=1:num_classes)
        DL = (y_indicator,a_L_prob) -> [-y_indicator[k] / a_L_prob[k] for k=1:num_classes]
        # These are gonna return vectors
        f = (z) -> [exp(z[j]) / sum(exp(z[k]) for k=1:num_classes) for j=1:num_classes]
        Df = (z) -> [(sum(exp(z[k]) for k=1:num_classes) - exp(z[j]))*exp(z[j]) /
(sum(exp(z[k]) for k=1:num_classes))^2 for j=1:num_classes]
    elseif problem_type == "regression"
        L = (y,a_L) -> norm(y - a_L)^2
        DL = (y,a_L) -> 2*(a_L - y)
        f = (z) -> z
        Df = (z) -> 1
    end

    # Total number of layers
    LL = num_hidden_layers + 2
    # Stepsize
    η = h

    # To store weighted inputs
    z = map(_ -> zeros(size_hidden_layers), 1:num_hidden_layers)
    pushfirst!(z, zeros(size(X,2)))
    push!(z, zeros(num_classes))
    # To store activations
    a = map(_ -> zeros(size_hidden_layers), 1:num_hidden_layers)
    pushfirst!(a, zeros(size(X,2)))
    push!(a, zeros(num_classes))
    # To store errors
    δ = map(_ -> zeros(size_hidden_layers), 1:num_hidden_layers)
    pushfirst!(δ, zeros(size(X,2)))
    push!(δ, zeros(num_classes))

    for _ = 1:num_epochs
        for i = 1:size(X,1)
            # Begin forward pass
            z[1] = X[i,:]   # identical transformation
            a[1] = Vector{Float64}(σ.(z[1]))
            for l = 2:(LL-1) # LL and LL-1
                z[l] = W[l]*a[l-1] + b[l]
                a[l] = Vector{Float64}(σ.(z[l]))
```

```
            end
            z[LL] = W[LL]*a[LL-1] + b[LL]
            # a[LL] = f(z[LL])   # Regression requires a broadcasting with a . while
classification does not

            # Begin backward pass
            if problem_type == "classification"
                # δ[LL] = DL(y_indicator[i], a[LL]) .* Dσ.(z[LL])
                a[LL] = f(z[LL])
                δ[LL] = DL(y_indicator[i], a[LL]) .* Df(z[LL])
            elseif problem_type == "regression"
                # δ[LL] = DL(y[i], a[LL][1]) .* Dσ.(z[LL])
                a[LL] = f.(z[LL])
                δ[LL] = DL(y[i], a[LL][1]) .* Df.(z[LL])
            end

            # Compute weighted errors
            for l = (LL-1):-1:2
                δ[l] = W[l+1]' * δ[l+1] .* Vector{Float64}(Dσ.(z[l]))
            end

            # Update weights and biases
            for l = 2:LL
                W[l] = W[l] - η * δ[l] * a[l-1]'
                b[l] = b[l] - η * δ[l]
            end
        end
    end

    # for l=2:LL
    #     println("W_$l is:\n$(W[l])")
    #     println("b_$l is:\n$(b[l])")
    # end

    function f_predictor(x)
        z[1] = x   # identical transformation
        a[1] = Vector{Float64}(σ.(z[1]))
        for l = 2:(LL-1)
            z[l] = W[l]*a[l-1] + b[l]
            a[l] = Vector{Float64}(σ.(z[l]))
        end
        z[LL] = W[LL]*a[LL-1] + b[LL]

        #println("$(a[L])")

        if problem_type == "classification"
            a[LL] = f(z[LL])
            y_pred = argmax(a[LL])
            if y_pred == 10
                y_pred = 0
```

```
            end
            return y_pred, a[LL]
        elseif problem_type == "regression"
            a[LL] = f.(z[LL])
            return a[LL][1]  # if regression, final layer has 1 output
        end
    end

    if problem_type == "classification"
        # loss = [L(y_indicator[i], f_predictor(X[i,:])) for i = 1:size(X,1)]
        y_pred_vec = zeros(size(X,1))
        a_L_vec = map(k -> zeros(num_classes), 1:size(X,1))
        for i = 1:size(X,1)
            y_pred_vec[i], a_L_vec[i] = f_predictor(X[i,:])
        end
        total_loss = sum([L(y_indicator[i], a_L_vec[i]) for i = 1:size(X,1)])
        println("Total loss is $total_loss.")
        ave_loss = Statistics.mean([L(y_indicator[i], a_L_vec[i]) for i = 1:size(X,1)])
        println("Average loss $ave_loss out of $(size(X,1)) training images.")

        # Compute misclassication
        misclass = 0
        for i = 1:size(X,1)
            if y_pred_vec[i] != y[i]
                misclass += 1
            end
        end
        println("$misclass of $(size(X,1)) training images misclassified.")
        return W,b
    elseif problem_type == "regression"
        ave_loss = Statistics.mean([L(y[i], f_predictor(X[i,:])) for i = 1:size(X,1)])
        println("Average loss $ave_loss out of $(size(X,1)) training images.")
        return W,b
    end

end


"""
    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.
"""
function
predict_neural_network(X,W,b,num_hidden_layers,size_hidden_layers;activation="sigmoid",proble
m_type="classification",num_classes=1)
    # Create activation functions
```

```
    if activation == "sigmoid"
        σ = (z) -> 1 / (1+exp(-z))
        Dσ = (z) -> σ(z)^2 * exp(-z)    # gradient with respect to z
        println("Sigmoid used.")
    elseif activation == "ReLU"
        σ = (z) -> max(z,0)
        Dσ = (z) -> begin
            if z >= 0.0
                return 1.0
            else
                return 0.0
            end
        end
        println("ReLU used.")
    end

    # Create appropriate loss function
    if problem_type == "classification"
        y_indicator = []
        classes = vcat(collect(1:9), 0)
        for i = 1:length(y)
            push!(y_indicator, map(k -> y[i] == k, classes))
        end

        L = (y_indicator,a_L_prob) -> -sum(y_indicator[k] * log(a_L_prob[k]) for
k=1:num_classes)
        DL = (y_indicator,a_L_prob) -> [-y_indicator[k] / a_L_prob[k] for k=1:num_classes]
        # These are gonna return vectors
        f = (z) -> [exp(z[j]) / sum(exp(z[k]) for k=1:num_classes) for j=1:num_classes]
        Df = (z) -> [(sum(exp(z[k]) for k=1:num_classes) - exp(z[j]))*exp(z[j]) /
(sum(exp(z[k]) for k=1:num_classes))^2 for j=1:num_classes]
    elseif problem_type == "regression"
        L = (y,a_L) -> norm(y - a_L)^2
        DL = (y,a_L) -> 2*(a_L - y)
        f = (z) -> z
        Df = (z) -> 1
    end

    # Total number of layers
    LL = num_hidden_layers + 2

    # To store weighted inputs
    z = map(_ -> zeros(size_hidden_layers), 1:num_hidden_layers)
    pushfirst!(z, zeros(size(X,2)))
    push!(z, zeros(num_classes))
    # To store activations
    a = map(_ -> zeros(size_hidden_layers), 1:num_hidden_layers)
    pushfirst!(a, zeros(size(X,2)))
    push!(a, zeros(num_classes))
```

```julia
    # for l=2:LL
    #     println("W_$l is:\n$(W[l])")
    #     println("b_$l is:\n$(b[l])")
    # end

    function f_predictor(x)
        z[1] = x    # identical transformation
        a[1] = Vector{Float64}(σ.(z[1]))
        for l = 2:(LL-1)
            z[l] = W[l]*a[l-1] + b[l]
            a[l] = Vector{Float64}(σ.(z[l]))
        end
        z[LL] = W[LL]*a[LL-1] + b[LL]

        #println("$(a[L])")

        if problem_type == "classification"
            a[LL] = f(z[LL])
            y_pred = argmax(a[LL])
            if y_pred == 10
                y_pred = 0
            end
            return y_pred
        elseif problem_type == "regression"
            a[LL] = f.(z[LL])
            return a[LL][1]  # if regression, final layer has 1 output
        end
    end

    predictions = zeros(size(X,1))
    for i = 1:size(X,1)
        predictions[i] = f_predictor(X[i,:])
    end

    return predictions
end
```

```julia
using LinearAlgebra

"""
    gradient_descent(f, ∇f, x0; α, tol, LIMIT)
Gradient descent algorithm for a function f, gradient ∇f, initial guess x0, learning rate α,
and tolerance tol. LIMIT is max iterations.
"""
function gradient_descent(f, ∇f, x0; α = 1.e-3, tol = 1.e-7, LIMIT = 5.e6)
    global counter = 0
    x = x0

    while norm(∇f(x)) > tol && counter < LIMIT
       x = x .- α .* ∇f(x)
       global counter += 1
    end

    println("Number of iterations: $counter.")
    if counter == LIMIT
        println("Max iterations reached.")
    end

    return x
end

"""
    newton_method(f, ∇f, ∇2f, x0; tol, LIMIT)
Newton method for a function f, gradient ∇f, Hessian ∇2f, initial guess x0, and tolerance
tol. LIMIT is max iterations.
"""
function newton_method(f, ∇f, ∇2f, x0; tol = 1.e-7, LIMIT = 500000)
    global counter = 0
    x = x0

    while norm(∇f(x)) > tol && counter < LIMIT
       x = x .- ∇2f(x) \ ∇f(x)
       global counter += 1
    end

    println("Number of iterations: $counter.")
    if counter == LIMIT
        println("Max iterations reached.")
    end

    return x
end


"""
    nonnegative_matrix_factorization(X, k; tol, LIMIT)
```

```julia
Performs nonnegative matrix factorization X = W * H where X is n x p, W is n x k, and H is k
x p. The inner dimension k is a hyperparameter.
"""
function nonnegative_matrix_factorization(X, k; tol = 1.e-7, LIMIT = 1000)
    n, p = size(X)
    W = ones(n, k)
    H = ones(k, p)


    global counter = 0
    oldWH = X
    newWH = W * H

    while norm(newWH - oldWH) > tol && counter < LIMIT
        oldWH = W * H

        H_num = W' * X
        H_denom = W' * W * H

        H = H .* (H_num ./ H_denom)

        W_num = X * H'
        W_denom = W * H * H'

        W = W .* (W_num ./ W_denom)

        global counter +=1
        newWH = W * H
    end

    return W, H

end


"""
    adaboost(X, y)
Performs the AdaBoost algorithm on a data set with entries consisting of -1's and 1's.
The columns of X are weak classifiers, and the entries of y are the labels for the rows of X.
"""
function adaboost(X, y; tol = 1.e-5, LIMIT = 5)
    n, p = size(X)

    coeffs = zeros(p)

    f(R) = dot(coeffs, R)
    global counter = 1
    ϵ = 1

    while counter < LIMIT && norm(ϵ) > tol
```

```
        current_min = Inf
        current_min_index = 1
        total_error = 0
        polarity = 1

        for c = 1:p
            class_error = 0
            total_error = 0

            for r = 1:n
                if X[r, c] != y[r]
                    class_error += exp(-y[r] * f(X[r, :]))
                end

                total_error += exp(-y[r] * f(X[r, :]))
            end

            this_polarity = 1

            if class_error > total_error / 2
                class_error = total_error - class_error
                this_polarity = -1
            end

            if class_error < current_min
                current_min = class_error
                current_min_index = c
                polarity = this_polarity
            end
        end

        ϵ = current_min / total_error
        α = 1/2 * log((1-ϵ)/ϵ + 1e-10)

        coeffs[current_min_index] += polarity * α

        global counter += 1
    end

    println("Number of iterations: $counter.")

    return R -> dot(R, coeffs)
end


"""
    k_means(X, k; LIMIT)
Performs k-means clustering on the matrix X with k clusters.
"""
function k_means(X, k; LIMIT = 500)
```

```
n, p = size(X)
centroids = zeros(k, p)
used = []
nearest = zeros(n)

# generate initial centroids randomly
for i = 1:k
    rand_idx = rand(1:n)
    while rand_idx ∈ used
        rand_idx = rand(1:n)
    end

    centroids[i, :] = X[rand_idx, :]
    append!(used, [rand_idx])
end


#identify nearest centroids for each observation given initial centroids
for row = 1:n
    current_min = Inf
    current_min_idx = 1

    for ctrd in 1:k
        dist = norm(X[row, :] .- centroids[ctrd, :])
        if dist < current_min
            current_min = dist
            current_min_idx = ctrd
        end
    end

    nearest[row] = current_min_idx
end

global counter = 0
centroids_changed = true
while counter < LIMIT && centroids_changed
    centroids_changed = false

    # update centroids
    for ctrd in 1:k
        idxs = []
        for row = 1:n
            if nearest[row] == ctrd
                append!(idxs, [row])
            end
        end

        this_cluster = X[idxs, :]
        numrows, _ = size(this_cluster)
```

```
        cluster_mean = zeros(p)
        for col in 1:p
            for row in 1:numrows
                cluster_mean[col] += 1/numrows * this_cluster[row, col]
            end
        end

        if centroids[ctrd, :] != cluster_mean[:]
            centroids_changed = true
        end

        centroids[ctrd, :] = cluster_mean[:]
    end

    #identify new nearest centroids for each observation
    for row = 1:n
        current_min = Inf
        current_min_idx = 1

        for ctrd in 1:k
            dist = norm(X[row, :] .- centroids[ctrd, :])
            if dist < current_min
                current_min = dist
                current_min_idx = ctrd
            end
        end

        nearest[row] = current_min_idx
    end

    counter += 1
end

#final nearest centroids
for row = 1:n
    current_min = Inf
    current_min_idx = 1

    for ctrd in 1:k
        dist = norm(X[row, :] .- centroids[ctrd, :])
        if dist < current_min
            current_min = dist
            current_min_idx = ctrd
        end
    end

    nearest[row] = current_min_idx
end

return centroids, nearest
```

```
end
```