```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
```

```julia
        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)
```

```julia
    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
```