

Julia SVM Helper Functions

```
#####
## by Qin Yu, Feb 2019
## using Julia 1.0.3
#####

gaussian_kernel_matrix(X, y) = # X's rows are input tensors.
    exp.(- y * pairwise(SqEuclidean(), Matrix(X), dims=1))
gaussian_kernel_matrix(X, X_test, y) = # X's rows are input tensors.
    exp.(- y * pairwise(SqEuclidean(), Matrix(X_test), Matrix(X), dims=1))

function extract_each_classes(X_ALL, y_ALL)
    CLASSES = sort(unique(y_ALL))
    NO_OF_CLASSES = length(CLASSES)
    L = [sum(y_ALL .== class) for class in CLASSES]
    X_EACH_train = [X_ALL[:, :, y_ALL .== class] for class in CLASSES]
    Xs = [hcat(transpose(reshape(X_EACH_train[i], 28*28, L[i])), ones(L[i]))) for i = 1:NO_OF_CLASSES]
    ys = ones(Int, L) .* CLASSES
    return L, Xs, ys
end

function extract_all_test(X_ALL, y_ALL)
    Xs = hcat(transpose(reshape(X_ALL, 28*28, :)), ones(size(X_ALL, 3)))
    return Xs, y_ALL
end

function prepare_one_vs_rest(X_ALL, y_ALL)
    X = hcat(transpose(reshape(X_ALL, 28*28, :)), ones(size(X_ALL, 3)))
    ys = [(y -> y == pos_label ? 1 : -1).(y_ALL) for pos_label = 0:9]
    return X, ys
end

function prepare_MNIST_data(class_pos, class_neg, y)
    # Load train data
    X_ALL_train, y_ALL_train = MNIST.traindata(Float32)

    NO_OF_NEGATIVE = sum(y_ALL_train .== class_neg)
    NO_OF_POSITIVE = sum(y_ALL_train .== class_pos)
    l = NO_OF_NEGATIVE + NO_OF_POSITIVE

    X_neg_train = X_ALL_train[:, :, y_ALL_train .== class_neg]
    X_pos_train = X_ALL_train[:, :, y_ALL_train .== class_pos]
    y_neg_train = - ones(Int, NO_OF_NEGATIVE)
    y_pos_train = ones(Int, NO_OF_POSITIVE)

    randomisation = randperm(l)
    X = hcat(transpose(reshape(cat(X_neg_train, X_pos_train; dims=3), 28*28, l)), ones(l))[randomisation, :]
    y = vcat(y_neg_train, y_pos_train)[randomisation, :]

    gaussian_kernel_matrix(X, y) = exp.(- y * pairwise(SqEuclidean(), Matrix(X), dims=1)) # X's rows are input tensors.
    K = gaussian_kernel_matrix(X, y)

    # Load test data
    X_ALL_test, y_ALL_test = MNIST.testdata(Float32)

    NO_OF_NEGATIVE_test = sum(y_ALL_test .== class_neg)
    NO_OF_POSITIVE_test = sum(y_ALL_test .== class_pos)
    l_test = NO_OF_NEGATIVE_test + NO_OF_POSITIVE_test

    X_neg_test = X_ALL_test[:, :, y_ALL_test .== class_neg]
    X_pos_test = X_ALL_test[:, :, y_ALL_test .== class_pos]
    y_neg_test = - ones(Int, NO_OF_NEGATIVE_test)
    y_pos_test = ones(Int, NO_OF_POSITIVE_test)

    X_test = hcat(transpose(reshape(cat(X_neg_test, X_pos_test; dims=3), 28*28, l_test)), ones(l_test))
    y_test = vcat(y_neg_test, y_pos_test)

    gaussian_kernel_matrix(X, X_test, y) = exp.(- y * pairwise(SqEuclidean(), Matrix(X_test), Matrix(X), dims=1))
    K_test = gaussian_kernel_matrix(X, X_test, y)

    return (X, y, K, l), (X_test, y_test, K_test, l_test)
end

# Only as termination criterion for the whole process.
function monitor_kkt_condition( $\alpha$ ,  $\sigma$ , y; accuracy=1e-2, C::Int32=Int32(1))
    yfx = y .*  $\sigma$ 
    for (idx,  $\alpha$ ) in enumerate( $\alpha$ )
        if  $\alpha$  == 0 && yfx[idx] < 1
            return false
        elseif  $\alpha$  == C && yfx[idx] > 1
            return false
        elseif 0 <  $\alpha$  < C && !isapprox(yfx[idx], 1; atol=accuracy)
            return false
        end
    end
    return true
end

function select_new_points( $\alpha$ ,  $\sigma$ , y, M, support_vector_idx; accuracy=1e-2, C::Int32=Int32(1))
    yfx = y .*  $\sigma$ 
    list_of_new_ids = Int[]
    list_of_new_min = Float64[]
    for (idx,  $\alpha$ ) in enumerate( $\alpha$ )
        if ( $\alpha$  == 0 && yfx[idx] < 1) || ( $\alpha$  == C && yfx[idx] > 1) || (0 <  $\alpha$  < C && !isapprox(yfx[idx], 1; atol=accuracy))

```

```

        push!(list_of_new_ids, idx)
        push!(list_of_new_min, yx[idx])
    end
end
list_of_new_ids = list_of_new_ids[sortperm(list_of_new_min)]
setdiff!(list_of_new_ids, support_vector_idx)
return list_of_new_ids
end

```

```
select_new_points (generic function with 1 method)
```

Julia SVM

```

#####
## by Qin Yu, Apr 2019
## using Julia 1.1.0
#####

using Revise, BenchmarkTools                # Development
using JLD2, FileIO, MLDatasets              # Data & IO
using LinearAlgebra, Distances, Random, Distributions # Maths
using CUDAdrv, CUDAnative, CuArrays        # GPU

MAX_MINI_BATCH_ID = 50

##### Compute Adatron SGD in GPU: #####
@inline sync_threads_and(predicate::Int32) = ccall("llvm.nvvm.barrier0.and", llvmcall, Int32, (Int32,), predicate)
@inline sync_threads_and(predicate::Bool) = ifelse(sync_threads_and(Int32(predicate)) != Int32(0), true, false)

function kernel_soft_SGD_SVM( $\alpha$ ,  $\sigma$ , K, y, l::Int32, C::Int32)
    j = (blockIdx().x-1) * blockDim().x + threadIdx().x
    can_stop = false
     $\delta_i$  = @cuStaticSharedMem(Float32, 1)
    while !sync_threads_and(can_stop)
        last_ $\alpha_j$  =  $\alpha[j]$ 
        # Adatron:
        for i = 1:l
            if j == i # Online
                last_ $\alpha$  =  $\alpha[i]$ 
                 $\mu_i$  = 1 / K[i,i]
                 $\delta_i[1]$  =  $\mu_i$  * (1 - y[i] *  $\sigma[i]$ )
                 $\alpha[i]$  =  $\alpha[i]$  +  $\delta_i[1]$ 
                 $\alpha[i] < 0$  && ( $\alpha[i] = 0$ ;  $\delta_i[1] = 0 - \text{last\_}\alpha$ )
                 $\alpha[i] > C$  && ( $\alpha[i] = C$ ;  $\delta_i[1] = C - \text{last\_}\alpha$ )
            end
            sync_threads()
             $\sigma[j] += \delta_i[1] * y[i] * K[i,j]$  # Parallel update
        end
        # Stopping criterion:
        can_stop = false
        isapprox( $\alpha[j]$ , last_ $\alpha_j$ ; atol=1e-4) && (can_stop = true)
    end
    return nothing
end

function optimise_working_set( $\alpha$ ,  $\sigma$ , K, y; C::Int32=Int32(1))
    l = Int32(length( $\alpha$ ))
    cu_ $\alpha$  = CuArray{Float32}( $\alpha$ )
    cu_ $\sigma$  = CuArray{Float32}( $\sigma$ )
    cu_K = CuArray{Float32}(K)
    cu_y = CuArray{Float32}(y)

    @cuda threads=l kernel_soft_SGD_SVM(cu_ $\alpha$ , cu_ $\sigma$ , cu_K, cu_y, l, C)

     $\alpha$  = Array{Float32}(cu_ $\alpha$ )
    return  $\alpha$ 
end

function optimise_working_set_CPU( $\alpha$ ,  $\sigma$ , K, y; C=1)
    l = length( $\alpha$ )
    while true
        last_ $\alpha$  = copy( $\alpha$ )
        for i = 1:l
             $\mu_i$  = 1 / K[i,i]
             $\delta_i$  =  $\mu_i$  * (1 - y[i] *  $\sigma[i]$ )
             $\alpha[i]$  =  $\alpha[i]$  +  $\delta_i$ 
             $\alpha[i] < 0$  && ( $\alpha[i] = 0$ ;  $\delta_i = 0 - \text{last\_}\alpha[i]$ )
             $\alpha[i] > C$  && ( $\alpha[i] = C$ ;  $\delta_i = C - \text{last\_}\alpha[i]$ )
             $\sigma$  .+=  $\delta_i * y[i] * K[i,:]$ 
        end
        all(isapprox.(last_ $\alpha$ ,  $\alpha$ ; atol=1e-4)) && break
    end
    return  $\alpha$ 
end

function stochastic_decomposition_test(C, M, ACCURACY, l, y, K, l_test, y_test, K_test; usecpu=false)
    M_safe = div(M, 4) * 3
    M_rand = M - M_safe

     $\alpha$  = zeros(Float32, l)
     $\sigma$  = zeros(Float32, l)

    selected_new_idx = []
    support_vector_idx, alpha0_vector_idx = collect(1:M), collect(M+1:l)

```

```

working_set_counter = zeros(Int32, 1)

batch_id = 0
while !monitor_kkt_condition( $\alpha$ ,  $\sigma$ , y; C=C) && batch_id < MAX_MINI_BATCH_ID
    batch_id += 1

    current_working_set_counter = working_set_counter[support_vector_idx]
    sorted_support_vector_idx = support_vector_idx[sortperm(current_working_set_counter)]

    if length(support_vector_idx) >= M_safe
        no_of_new_idx = min(length(selected_new_idx), M_rand)
    else
        no_of_new_idx = min(length(selected_new_idx), M-length(support_vector_idx))
    end

    no_of_old_idx = length(support_vector_idx)
    no_of_exceeds = no_of_new_idx + no_of_old_idx - M
    if no_of_exceeds > 0
        abandoned_support_vector_idx = sorted_support_vector_idx[end+1-no_of_exceeds:end]
        setdiff!(support_vector_idx, abandoned_support_vector_idx)
        union!(alpha0_vector_idx, abandoned_support_vector_idx)
    end
    selected_new_idx = selected_new_idx[1:no_of_new_idx]
    union!(support_vector_idx, selected_new_idx)
    setdiff!(alpha0_vector_idx, selected_new_idx)

    working_set_counter[support_vector_idx] .+= 1
     $\alpha$ _subset,  $\sigma$ _subset =  $\alpha$ [support_vector_idx],  $\sigma$ [support_vector_idx]
    y_subset, K_subset = y[support_vector_idx], K[support_vector_idx, support_vector_idx]
    if usecpu
         $\alpha$ [support_vector_idx] = optimise_working_set_CPU( $\alpha$ _subset,  $\sigma$ _subset, K_subset, y_subset; C=C)
    else
         $\alpha$ [support_vector_idx] = optimise_working_set( $\alpha$ _subset,  $\sigma$ _subset, K_subset, y_subset; C=C)
    end

     $\sigma$  = K * ( $\alpha$  .* y)

    for (i_index, i) in enumerate(support_vector_idx)
        if  $\alpha$ [i] == 0
            deleteat!(support_vector_idx, i_index)
            push!(alpha0_vector_idx, i)
        end
    end

    selected_new_idx = select_new_points( $\alpha$ ,  $\sigma$ , y, M, support_vector_idx; accuracy=ACCURACY, C=C)
end

 $\hat{y}$  = sign( $\sigma$ )
 $\hat{y}$ _test = sign(K_test * ( $\alpha$  .* y))
error_rate = count( $\hat{y}$  .!= y) / 1 * 100
error_rate_test = count( $\hat{y}$ _test .!= y_test) / 1_test * 100
return  $\alpha$ , error_rate, error_rate_test, monitor_kkt_condition( $\alpha$ ,  $\sigma$ , y; C=C)
end

function stochastic_decomposition(C, M, ACCURACY, 1, y, K; usecpu=false)
    M_safe = div(M, 4) * 3
    M_rand = M - M_safe

     $\alpha$  = zeros(Float32, 1)
     $\sigma$  = zeros(Float32, 1)

    selected_new_idx = []
    support_vector_idx, alpha0_vector_idx = collect(1:M), collect(M+1:1)
    working_set_counter = zeros(Int32, 1)

    batch_id = 0
    while !monitor_kkt_condition( $\alpha$ ,  $\sigma$ , y; accuracy=ACCURACY, C=C) && batch_id < MAX_MINI_BATCH_ID
        batch_id += 1

        current_working_set_counter = working_set_counter[support_vector_idx]
        sorted_support_vector_idx = support_vector_idx[sortperm(current_working_set_counter)]

        if length(support_vector_idx) >= M_safe
            no_of_new_idx = min(length(selected_new_idx), M_rand)
        else
            no_of_new_idx = min(length(selected_new_idx), M-length(support_vector_idx))
        end

        no_of_old_idx = length(support_vector_idx)
        no_of_exceeds = no_of_new_idx + no_of_old_idx - M
        if no_of_exceeds > 0
            abandoned_support_vector_idx = sorted_support_vector_idx[end+1-no_of_exceeds:end]
            setdiff!(support_vector_idx, abandoned_support_vector_idx)
            union!(alpha0_vector_idx, abandoned_support_vector_idx)
        end
        selected_new_idx = selected_new_idx[1:no_of_new_idx]
        union!(support_vector_idx, selected_new_idx)
        setdiff!(alpha0_vector_idx, selected_new_idx)

        working_set_counter[support_vector_idx] .+= 1
         $\alpha$ _subset,  $\sigma$ _subset =  $\alpha$ [support_vector_idx],  $\sigma$ [support_vector_idx]
        y_subset, K_subset = y[support_vector_idx], K[support_vector_idx, support_vector_idx]
        if usecpu
            println("using CPU")
             $\alpha$ [support_vector_idx] = optimise_working_set_CPU( $\alpha$ _subset,  $\sigma$ _subset, K_subset, y_subset; C=C)
        else

```

```

         $\alpha$ [support_vector_idx] = optimise_working_set( $\alpha$ _subset,  $\sigma$ _subset, K_subset, y_subset; C=C)
    end
     $\sigma$  = K * ( $\alpha$  .* y)

    for (i_index,i) in enumerate(support_vector_idx)
        if  $\alpha$ [i] == 0
            deleteat!(support_vector_idx, i_index)
            push!(alpha0_vector_idx, i)
        end
    end

    selected_new_idx = select_new_points( $\alpha$ ,  $\sigma$ , y, M, support_vector_idx; accuracy=ACCURACY, C=C)
end

 $\hat{y}$  = sign( $\sigma$ )
error_rate = count( $\hat{y}$  .!= y) / 1 * 100
return  $\alpha$ , error_rate, monitor_kkt_condition( $\alpha$ ,  $\sigma$ , y; accuracy=ACCURACY, C=C)
end

```

`stochastic_decomposition` (generic function with 1 method)

MNIST Training and Testing

```

#####
## by Qin Yu, Apr 2019
## using Julia 1.1.0
#####

##### MNIST One-vs-One Multi-class #####
function MNIST_train(L, L_test, Xs, ys, C, M, ACCURACY, y; usecpu=false)
    time_K_spent = time_GPU_spent = time_total_spent = 0
    time_total_start = time()

    randomisation_list = Array{Int32,1}[]
     $\alpha$ _list = Array{Float32,1}[]
    error_list = Float64[]
    error_list_test = Float64[]
    finished_list = Bool[]
    time_ivj_list = Float64[]

    for ci = 1:10, cj = 1:10
        ci >= cj && continue
        print("$ci-1-vs-$(cj-1) - ")

        time_K_start = time()
        l = L[ci] + L[cj]
        randomisation = randperm(l)
        y = vcat(ones(Int32, size(ys[ci])), -ones(Int32, size(ys[cj])))[randomisation,:]
        X = vcat(Xs[ci], Xs[cj])[randomisation,:]
        K = gaussian_kernel_matrix(X, y)
        time_K_spent += time() - time_K_start

        # l_test = L_test[ci] + L_test[cj]
        # randomisation_test = randperm(l_test)
        # y_test = vcat(ones(Int32, size(ys_test[ci])), -ones(Int32, size(ys_test[cj])))[randomisation_test,:]
        # X_test = vcat(Xs_test[ci], Xs_test[cj])[randomisation_test,:]
        # K_test = gaussian_kernel_matrix(X, X_test, y)

        time_GPU_start = time()
         $\alpha$ , error_rate, finished = stochastic_decomposition(C, M, ACCURACY, l, y, K; usecpu=usecpu)
        #  $\alpha$ , error_rate, error_rate_test, finished =
        #     stochastic_decomposition(C, M, l, y, K, l_test, y_test, K_test)
        time_GPU_spent_this = time() - time_GPU_start
        push!(time_ivj_list, time_GPU_spent_this)
        time_GPU_spent += time_GPU_spent_this

        println("error rate = $error_rate% ; finished = $finished ; time = $time_GPU_spent_this")
        # println("error rate = $error_rate% ; test error rate = $error_rate_test% ;
        #     finished = $finished ; time = $time_GPU_spent_this")

        push!(randomisation_list, randomisation)
        push!( $\alpha$ _list,  $\alpha$ )
        push!(error_list, error_rate)
        # push!(error_list_test, error_rate_test)
        push!(finished_list, finished)
    end

    time_total_spent = time() - time_total_start
    return ([time_K_spent, time_GPU_spent, time_total_spent],
            randomisation_list,  $\alpha$ _list, error_list, error_list_test, finished_list, time_ivj_list)
end

function MNIST_test_slave(X_ALL, y_ALL, Xs, ys,  $\alpha$ _list, randomisation_list, y)
    i = 0
     $\hat{y}$ _test_list = Array{Int}[]
    for ci = 1:10, cj = 1:10
        ci >= cj && continue
        print("$ci-1-vs-$(cj-1) - ")
        i += 1

        X_test, y_test = extract_all_test(X_ALL, y_ALL)
        K_test = gaussian_kernel_matrix(vcat(Xs[ci], Xs[cj])[randomisation_list[i],:], X_test, y)
        y_true = vcat(ones(Int32, size(ys[ci])), -ones(Int32, size(ys[cj])))[randomisation_list[i],:]
         $\sigma$ _test = K_test * ( $\alpha$ _list[i] .* y_true)
    end
end

```

```

        ŷ_test = (y -> y > 0 ? ci-1 : cj-1).(σ_test)

        push!(ŷ_test_list, ŷ_test)
    end
    return ŷ_test_list
end

function MNIST_test(y_test, ŷ_test_list)
    ŷ_test_matrix = hcat(ŷ_test_list...)
    ŷ_test = mapslices(mode, ŷ_test_matrix; dims=2)
    error_rate_test = count(ŷ_test .!= y_test) / length(y_test) * 100
    correct_rate_test = 100 - error_rate_test # First run: 99.1%
    return ŷ_test, error_rate_test, correct_rate_test
end

function run(C::Int32=Int32(1), # Penalty
            M::Int=512, # Max size of minibatch
            ACCURACY=1e-2, # GPU accuracy = 0.01 * ACCURACY
            γ=0.015; # Gaussian kernel parameter
            usecpu=false)
    X_ALL, y_ALL = MNIST.traindata(Float32)
    X_ALL_test, y_ALL_test = MNIST.testdata(Float32)
    L, Xs, ys = extract_each_classes(X_ALL, y_ALL)
    L_test, Xs_test, ys_test = extract_each_classes(X_ALL_test, y_ALL_test)

    @time time_list, randomisation_list, α_list, error_list, error_list_test, finished_list, time_ivj_list =
        MNIST_train(L, L_test, Xs, ys, C, M, ACCURACY, γ; usecpu=usecpu)
    @save "iv1.jld2" time_list, randomisation_list, α_list, error_list, error_list_test, finished_list, time_ivj_list
    @time ŷ_test_list = MNIST_test_slave(X_ALL, y_ALL, Xs, ys, α_list, randomisation_list, γ)
    @time ŷ_test, error_rate_test = MNIST_test(y_ALL, ŷ_test_list)
    println(error_rate_test)
end

```

run (generic function with 5 methods)

Run SVM

run() # 1256.897736s ≈ 20.95min

```

0-vs-1 - error rate = 0.007895775759968417% ; finished = true ; time = 11.731913089752197
0-vs-2 - error rate = 0.042083999663328% ; finished = true ; time = 12.422368049621582
0-vs-3 - error rate = 0.024888003982080638% ; finished = true ; time = 10.846972942352295
0-vs-4 - error rate = 0.06799830004249893% ; finished = true ; time = 7.246622085571289
0-vs-5 - error rate = 0.07052186177715092% ; finished = true ; time = 18.025906801223755
0-vs-6 - error rate = 0.10978802466007939% ; finished = true ; time = 10.650660037994385
0-vs-7 - error rate = 0.016409583196586804% ; finished = true ; time = 7.1162559861145
0-vs-8 - error rate = 0.08493290300662477% ; finished = true ; time = 11.790964126586914
0-vs-9 - error rate = 0.07580862533692723% ; finished = true ; time = 7.729828834533691
1-vs-2 - error rate = 0.13385826771653542% ; finished = true ; time = 16.983326196670532
1-vs-3 - error rate = 0.06991377301328362% ; finished = true ; time = 11.7693190574646
1-vs-4 - error rate = 0.14303877940241577% ; finished = true ; time = 8.491616010665894
1-vs-5 - error rate = 0.04932993504891885% ; finished = true ; time = 11.43572211265564
1-vs-6 - error rate = 0.023696682464454978% ; finished = true ; time = 22.854277849197388
1-vs-7 - error rate = 0.261397708925963% ; finished = true ; time = 12.761433839797974
1-vs-8 - error rate = 0.1508774716112126% ; finished = true ; time = 25.24915885925293
1-vs-9 - error rate = 0.11031439602868175% ; finished = true ; time = 7.722901821136475
2-vs-3 - error rate = 0.223435354454463% ; finished = true ; time = 51.78821110725403
2-vs-4 - error rate = 0.11864406779661016% ; finished = true ; time = 24.21775197982788
2-vs-5 - error rate = 0.043940592319184464% ; finished = true ; time = 24.209162950515747
2-vs-6 - error rate = 0.0421017177500842% ; finished = true ; time = 26.605338096618652
2-vs-7 - error rate = 0.29452671193651314% ; finished = true ; time = 31.75123906135559
2-vs-8 - error rate = 0.20323482089931408% ; finished = true ; time = 57.46496510505676
2-vs-9 - error rate = 0.07558578987150416% ; finished = true ; time = 15.855940103530884
3-vs-4 - error rate = 0.05011275369581559% ; finished = true ; time = 15.090398073196411
3-vs-5 - error rate = 0.3808864265927978% ; finished = true ; time = 66.38876104354858
3-vs-6 - error rate = 0.008299443937256203% ; finished = true ; time = 14.185173988342285
3-vs-7 - error rate = 0.18554372378186512% ; finished = true ; time = 28.159248113632202
3-vs-8 - error rate = 0.35052578868302453% ; finished = true ; time = 55.56105613708496
3-vs-9 - error rate = 0.25662251655629137% ; finished = true ; time = 22.970326900482178
4-vs-5 - error rate = 0.03551451655864335% ; finished = true ; time = 19.871274948120117
4-vs-6 - error rate = 0.13605442176870747% ; finished = true ; time = 15.57249402999878
4-vs-7 - error rate = 0.21475179648137438% ; finished = true ; time = 42.69997715950012
4-vs-8 - error rate = 0.09407337723424271% ; finished = true ; time = 19.575289964675903
4-vs-9 - error rate = 0.5003816470189127% ; finished = true ; time = 73.25518202781677
5-vs-6 - error rate = 0.17638239703677575% ; finished = true ; time = 26.81020998954773
5-vs-7 - error rate = 0.06845798391237377% ; finished = true ; time = 20.46792197227478
5-vs-8 - error rate = 0.17743080198722497% ; finished = true ; time = 53.68216514587402
5-vs-9 - error rate = 0.12313104661389623% ; finished = true ; time = 23.383502960205078
6-vs-7 - error rate = 0.008208158909956496% ; finished = true ; time = 6.258177995681763
6-vs-8 - error rate = 0.05098139179199593% ; finished = true ; time = 16.659406185150146
6-vs-9 - error rate = 0.02528018875874273% ; finished = true ; time = 7.46078896522522
7-vs-8 - error rate = 0.115549686365137% ; finished = true ; time = 18.93709897994995
7-vs-9 - error rate = 0.4830522351400033% ; finished = true ; time = 76.98206496238708
8-vs-9 - error rate = 0.1440677966101695% ; finished = true ; time = 44.56295299530029
Total training time 1278.997176 seconds (19.93 M allocations: 159.586 GiB, 1.11% gc time)

```

Final error rate 0.895%