Student: Radu Madalin-Cristian

Group: 407

Bayesian Neural Networks

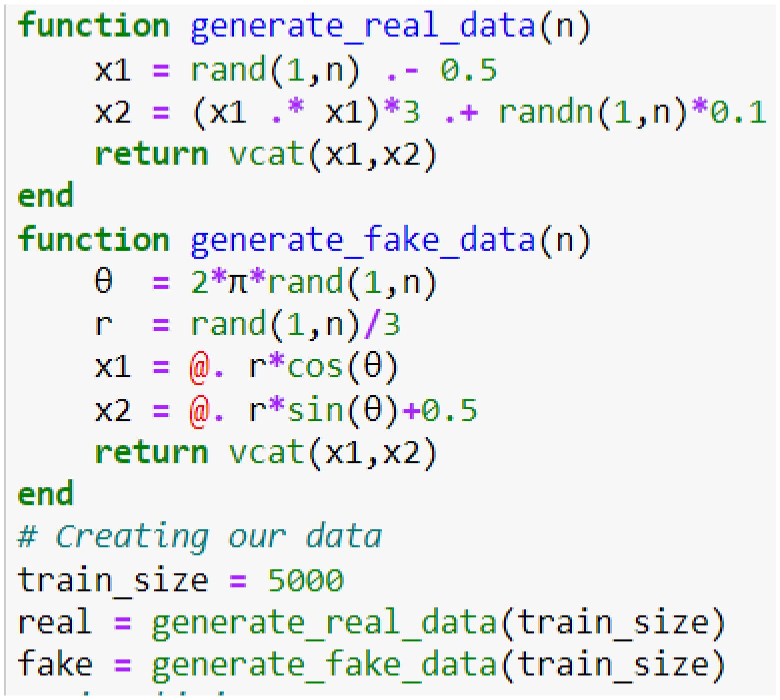
Probabilistic Programming – Project 2

1) The task

We are asked to build a Bayesian Neural Network and train it using MCMC to make categorical and binary classifications.

2) The implementation

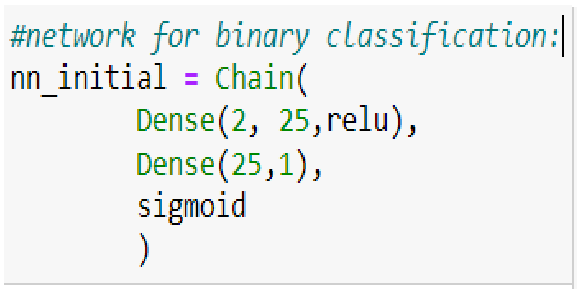
I used the iris dataset for the multi-label classification and some randomly generated points for the binary classification. This is how I created the data for the binary classification:



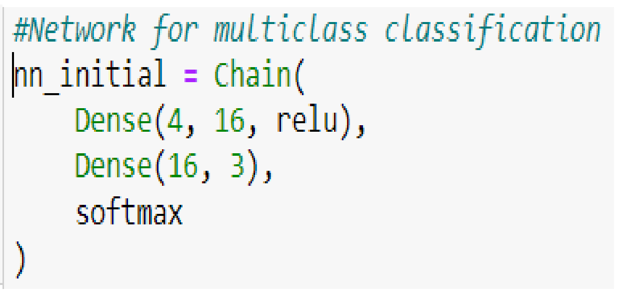
```julia
function generate_real_data(n)
    x1 = rand(1,n) .- 0.5
    x2 = (x1 .* x1)*3 .+ randn(1,n)*0.1
    return vcat(x1,x2)
end
function generate_fake_data(n)
    θ  = 2*π*rand(1,n)
    r  = rand(1,n)/3
    x1 = @. r*cos(θ)
    x2 = @. r*sin(θ)+0.5
    return vcat(x1,x2)
end
# Creating our data
train_size = 5000
real = generate_real_data(train_size)
fake = generate_fake_data(train_size)
```

The goal will be to differentiate between “real” and “fake” points.

I used the Flux library to create the networks. These are the networks for each of the classification tasks:



```julia
#network for binary classification:
nn_initial = Chain(
          Dense(2, 25,relu),
          Dense(25,1),
          sigmoid
          )
```



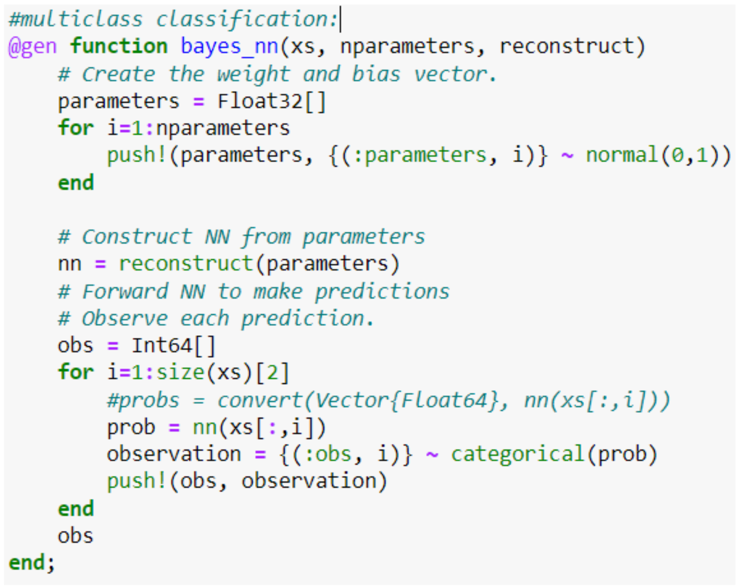
```julia
#Network for multiclass classification
nn_initial = Chain(
    Dense(4, 16, relu),
    Dense(16, 3),
    softmax
)
```

The iris entries have 4 values so there are 4 input neurons. There are 3 possible output targets, so the last layer has 3 neurons. The binary classification data consists of points which have an x value and a y value, meaning the input layer has 2 neurons. The output is the probability that the input is a “real” point.

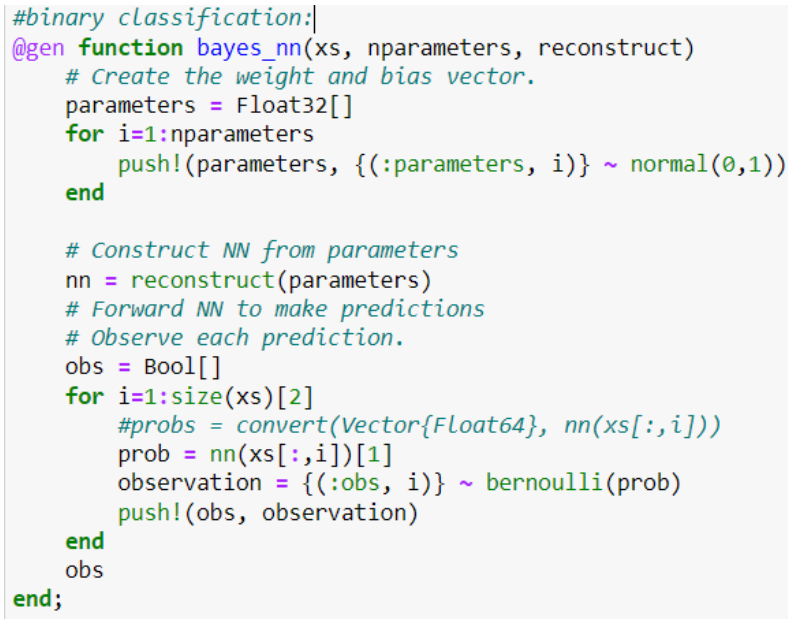
The Flux library also facilitates rebuilding the neural network after recalculating the weights and biases:

```julia
# Extract weights and a helper function to reconstruct NN from weights
parameters_initial, reconstruct = Flux.destructure(nn_initial)
```

This makes building the generative functions much easier:

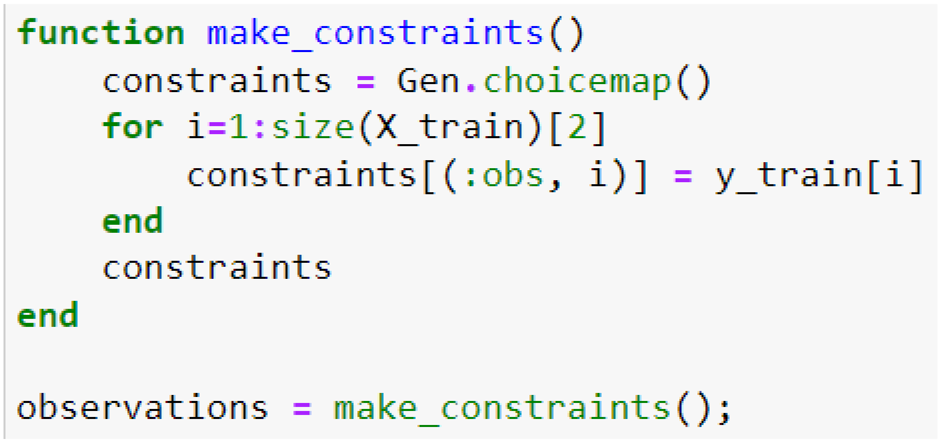


```julia
#multiclass classification:
@gen function bayes_nn(xs, nparameters, reconstruct)
    # Create the weight and bias vector.
    parameters = Float32[]
    for i=1:nparameters
        push!(parameters, {(:parameters, i)} ~ normal(0,1))
    end

    # Construct NN from parameters
    nn = reconstruct(parameters)
    # Forward NN to make predictions
    # Observe each prediction.
    obs = Int64[]
    for i=1:size(xs)[2]
        #probs = convert(Vector{Float64}, nn(xs[:,i]))
        prob = nn(xs[:,i])
        observation = {(:obs, i)} ~ categorical(prob)
        push!(obs, observation)
    end
    obs
end;
```

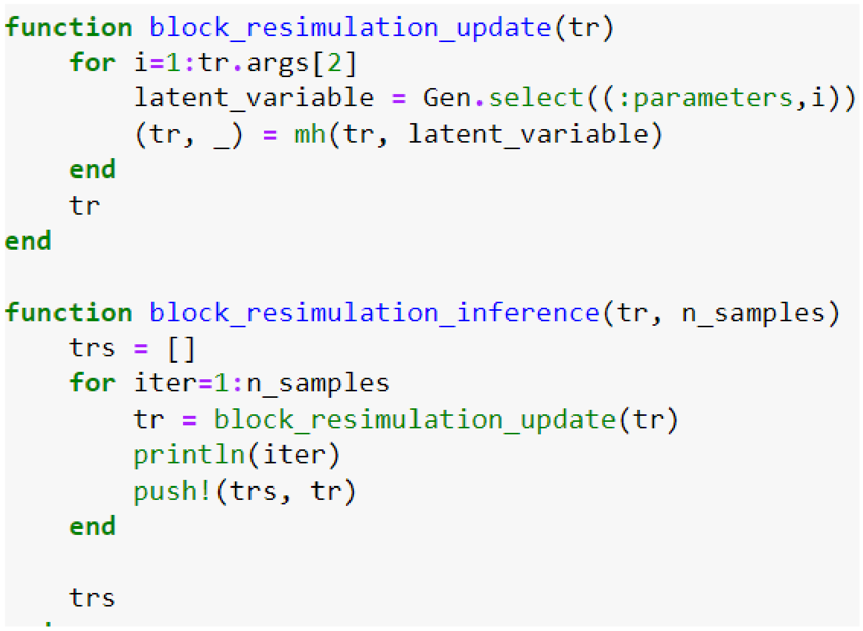


```julia
#binary classification:
@gen function bayes_nn(xs, nparameters, reconstruct)
    # Create the weight and bias vector.
    parameters = Float32[]
    for i=1:nparameters
        push!(parameters, {(:parameters, i)} ~ normal(0,1))
    end

    # Construct NN from parameters
    nn = reconstruct(parameters)
    # Forward NN to make predictions
    # Observe each prediction.
    obs = Bool[]
    for i=1:size(xs)[2]
        #probs = convert(Vector{Float64}, nn(xs[:,i]))
        prob = nn(xs[:,i])[1]
        observation = {(:obs, i)} ~ bernoulli(prob)
        push!(obs, observation)
    end
    obs
end;
```

All that is left to do is make the constraints for the models and define the inference function.

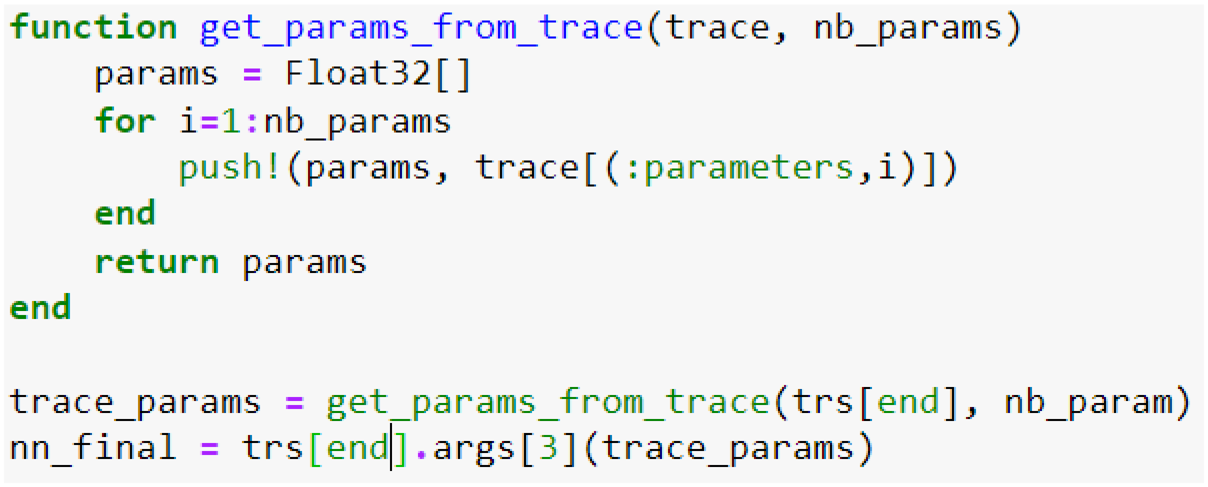


```julia
function make_constraints()
    constraints = Gen.choicemap()
    for i=1:size(X_train)[2]
        constraints[(:obs, i)] = y_train[i]
    end
    constraints
end

observations = make_constraints();
```



```
function block_resimulation_update(tr)
    for i=1:tr.args[2]
        latent_variable = Gen.select((:parameters,i))
        (tr, _) = mh(tr, latent_variable)
    end
    tr
end

function block_resimulation_inference(tr, n_samples)
    trs = []
    for iter=1:n_samples
        tr = block_resimulation_update(tr)
        println(iter)
        push!(trs, tr)
    end

    trs
```

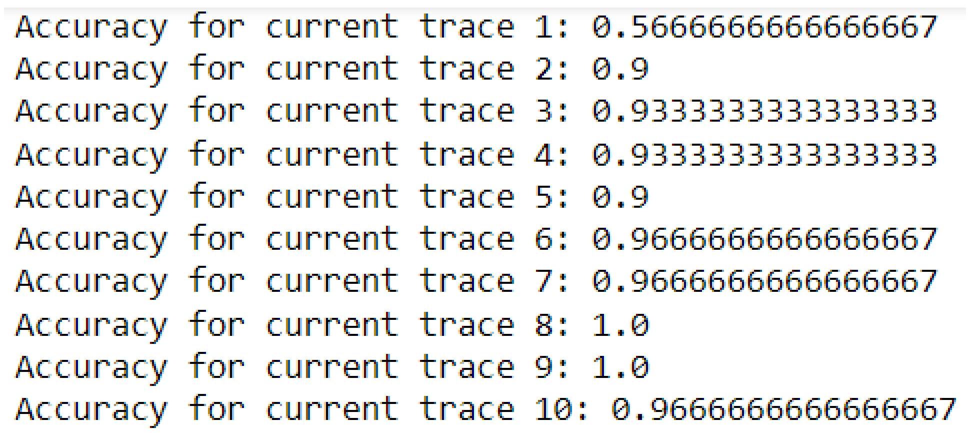
Building the final Neural Network:



```
function get_params_from_trace(trace, nb_params)
    params = Float32[]
    for i=1:nb_params
        push!(params, trace[(:parameters,i)])
    end
    return params
end

trace_params = get_params_from_trace(trs[end], nb_param)
nn_final = trs[end].args[3](trace_params)
```

The results

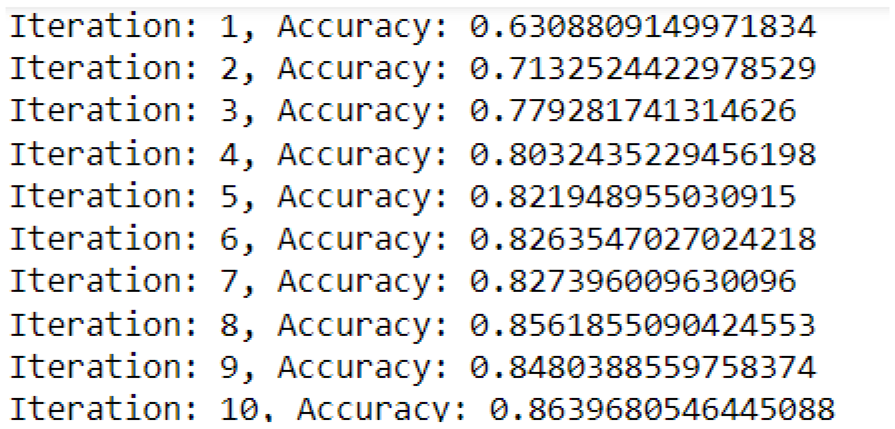
The Bayes Neural Network reached an almost perfect accuracy in less than 10 epochs, for multi-class classification:



```
Accuracy for current trace 1: 0.5666666666666667
Accuracy for current trace 2: 0.9
Accuracy for current trace 3: 0.9333333333333333
Accuracy for current trace 4: 0.9333333333333333
Accuracy for current trace 5: 0.9
Accuracy for current trace 6: 0.9666666666666667
Accuracy for current trace 7: 0.9666666666666667
Accuracy for current trace 8: 1.0
Accuracy for current trace 9: 1.0
Accuracy for current trace 10: 0.9666666666666667
```

Accuracy for BNN, multi-class classification

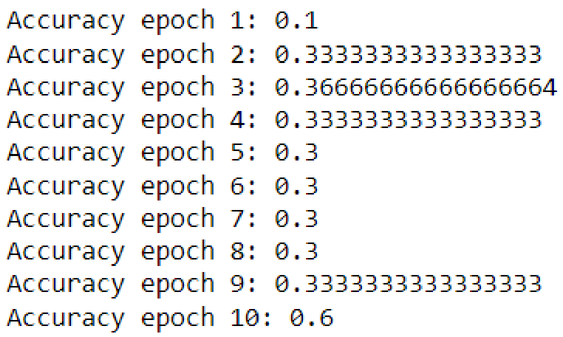
For binary classification, the predicted probability reached about 0.86 in 10 epochs:



```
Iteration: 1, Accuracy: 0.6308809149971834
Iteration: 2, Accuracy: 0.7132524422978529
Iteration: 3, Accuracy: 0.779281741314626
Iteration: 4, Accuracy: 0.8032435229456198
Iteration: 5, Accuracy: 0.821948955030915
Iteration: 6, Accuracy: 0.8263547027024218
Iteration: 7, Accuracy: 0.827396009630096
Iteration: 8, Accuracy: 0.8561855090424553
Iteration: 9, Accuracy: 0.8480388559758374
Iteration: 10, Accuracy: 0.8639680546445088
```

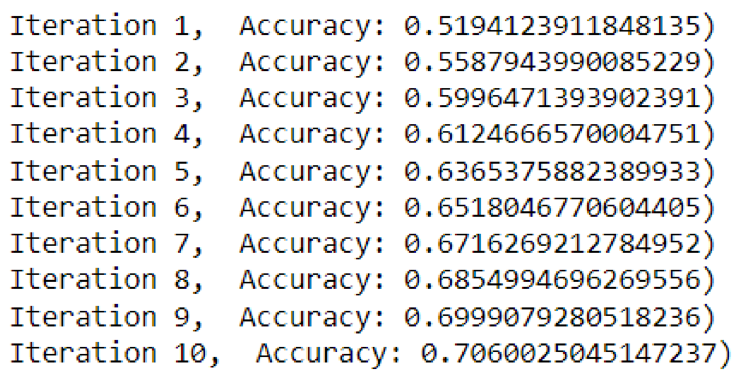
Accuracy for BNN, binary classification

For the traditional neural network, the numbers are a bit different:



```
Accuracy epoch 1: 0.1
Accuracy epoch 2: 0.3333333333333333
Accuracy epoch 3: 0.36666666666666664
Accuracy epoch 4: 0.3333333333333333
Accuracy epoch 5: 0.3
Accuracy epoch 6: 0.3
Accuracy epoch 7: 0.3
Accuracy epoch 8: 0.3
Accuracy epoch 9: 0.3333333333333333
Accuracy epoch 10: 0.6
```

Accuracy for traditional NN, Multi-class classification



```
Iteration 1,  Accuracy: 0.5194123911848135)
Iteration 2,  Accuracy: 0.5587943990085229)
Iteration 3,  Accuracy: 0.5996471393902391)
Iteration 4,  Accuracy: 0.6124666570004751)
Iteration 5,  Accuracy: 0.6365375882389933)
Iteration 6,  Accuracy: 0.6518046770604405)
Iteration 7,  Accuracy: 0.6716269212784952)
Iteration 8,  Accuracy: 0.6854994696269556)
Iteration 9,  Accuracy: 0.6999079280518236)
Iteration 10,  Accuracy: 0.7060025045147237)
```

Accuracy for traditional NN, binary classification

As it can be seen from the images, the Bayesian Neural Networks have vastly better performance metrics. Furthermore, training them took less time, so not only are they better, but they are also time efficient.

Bibliography

- https://medium.com/coffee-in-a-klein-bottle/deep-learning-with-julia-e7f15ad5080b
- https://towardsdatascience.com/how-to-build-an-artificial-neural-network-from-scratch-in-julia-c839219b3ef8
- https://towardsdatascience.com/build-your-first-neural-network-with-flux-jl-in-julia-10ebdfcf2fa3
- https://github.com/julia4ta/tutorials/tree/master/Series%2005/Tutorial%2005x08