```
-- Only requirements allowed
require("hdf5")
require("nn")
require("optim")
require("gnuplot")
require("math")
-- require("cutorch")

cmd = torch.CmdLine()

-- Cmd Args
cmd:option('-datafile', 'PTB.hdf5', 'data file')
cmd:option('-classifier', 'nb', 'classifier to use')
cmd:option('-alpha', 1.0, 'Laplace smoothing coefficient')
cmd:option('-eta', 0.01, 'learning rate')
cmd:option('-lambda', 0.05, 'l2 regularization coefficient')
cmd:option('-n_epochs', 15, 'number of training epochs')
cmd:option('-m', 32, 'minibatch size')
cmd:option('-embed', 'n', 'word embeddings')

-- Writing to file
function writeToFile(predictions)
  local f = torch.DiskFile('predictions.txt', 'w')
  f:writeString('ID,Category\n')
  local id = 1

  for i = 1, predictions:size(1) do
    local pred = predictions[i][1]
    f:writeString(id .. ',' .. pred .. '\n')
    id = id + 1
  end
  f:close()
end

-- Plotting
function plot(data, title, xlabel, ylabel, filename)
  gnuplot.pngfigure(filename .. '.png')
  gnuplot.plot(data)
  gnuplot.title(title)
  gnuplot.xlabel(xlabel)
  gnuplot.ylabel(ylabel)
  gnuplot.plotflush()
end

function naive_bayes()
  print('Building naive bayes model...')
```

```
-- Generate prior for each part of speech
local double_output = train_output:double()
local p_y = torch.histc(double_output, nclasses)
p_y:div(torch.sum(p_y))

-- Build multinomial distribution for each relative window position
-- ie: p(word at position 1|y)
--     p(capitalization at position 1|y)

-- Record number occurrences of each word at each position given a class
print('Building p(word at i|y), p(cap at i|y)...')
local word_occurrences = torch.zeros(nclasses, dwin, nwords)
local cap_occurrences = torch.zeros(nclasses, dwin, ncaps)

for i = 1, train_output:size(1) do
  local y = train_output[i]

  local word_window = train_input_word_windows[i]
  local cap_window = train_input_cap_windows[i]
  for j = 1, dwin do
    local w = word_window[j]
    local c = cap_window[j]

    word_occurrences[y][j][w] = word_occurrences[y][j][w] + 1
    cap_occurrences[y][j][c] = cap_occurrences[y][j][c] + 1
  end
end

-- Add smoothing to account for words/caps not appearing in a position/class
local nb_accuracy = {}

alphas = {
  0.01
}

for k, v in pairs(alphas) do
  word_alpha = word_occurrences:clone()
  cap_alpha = cap_occurrences:clone()
  word_alpha:add(v)
  cap_alpha:add(v)

  -- Normalize to 1
  for y = 1, nclasses do
    for p = 1, dwin do
      -- All word/cap occurrences at position p in class y
      local w_sum = word_alpha[y][p]:sum()
      local c_sum = cap_alpha[y][p]:sum()
```

```
    -- Divide by sum across nwords/ncaps
    word_alpha:select(1, y):select(1, p):div(w_sum)
    cap_alpha:select(1, y):select(1, p):div(c_sum)
  end
end

function predict(word_windows, cap_windows)
  local pred = torch.IntTensor(word_windows:size(1))
  for i = 1, word_windows:size(1) do
    local word_window = word_windows[i]
    local cap_window = cap_windows[i]

    local p_y_hat = torch.zeros(nclasses)
    for y = 1, nclasses do
      p_y_hat[y] = p_y[y]

      -- Multiply p_y_hat by p(word at jly) and p(cap at jly)
      for j = 1, dwin do
        w = word_window[j]
        c = cap_window[j]

        p_y_hat[y] = p_y_hat[y] * word_alpha[y][j][w]
        p_y_hat[y] = p_y_hat[y] * cap_alpha[y][j][c]
      end
    end

    p_y_hat:div(p_y_hat:sum())
    val, prediction = torch.max(p_y_hat, 1)

    pred[i] = prediction
  end
  return pred
end

print('Running naive bayes on validation set...')

-- Generate predictions on validation
local pred = predict(valid_input_word_windows, valid_input_cap_windows)

pred = pred:eq(valid_output):double()
local accuracy = torch.mean(pred) * 100
nb_accuracy[v] = accuracy
print('Alpha ' .. v .. ' Validation accuracy: ' .. accuracy .. '.')

pred = predict(train_input_word_windows, train_input_cap_windows)
pred = pred:eq(train_output):double()
```

```
    accuracy = torch.mean(pred) * 100
    print('Train accuracy: ' .. accuracy .. '.')
  end
  -- print(nb_accuracy)
  -- plot(nb_accuracy, 'Naive Bayes Validation Accuracy', 'Alpha', 'Accuracy', 'nb_alpha')

  print('Writing to file...\n')
  local f = torch.DiskFile('training_output/naivebayes.txt', 'w')
  f:seekEnd()
  for k, v in pairs(nb_accuracy) do
    f:writeString('alpha ' .. k .. ' accuracy : ' .. '\n')
  end

  f:close()
    -- print('Running naive bayes on test set...')
  -- test_preds = predict(test_input_word_windows, test_input_cap_windows)
  -- writeToFile(test_preds)
end

function model(structure)
  local embedding_size = 50
  local caps_size = 5
  local din = dwin * (embedding_size + caps_size)
  local dout = nclasses
  local dhid2 = 200

  local model = nn.Sequential()

  if structure == 'lr' then
    print('Building logistic regression model...')

    -- local x_ww = torch.zeros(2,5)
    -- x_ww[1] = train_input_word_windows[1]
    -- x_ww[2] = train_input_word_windows[2]
    -- local x_cw = torch.zeros(2,5)
    -- x_cw[1] = train_input_cap_windows[1]
    -- x_cw[2] = train_input_cap_windows[2]

    local sparseW_word = nn.LookupTable(nwords, nclasses)
    local W_word = nn.Sequential():add(sparseW_word):add(nn.Sum(2))

    local sparseW_cap = nn.LookupTable(ncaps, nclasses)
    local W_cap = nn.Sequential():add(sparseW_cap):add(nn.Sum(2))

    local par = nn.ParallelTable()
    par:add(W_word) -- first child
    par:add(W_cap) -- second child
```

```
  model:add(par)
  model:add(nn.CAddTable())
  model:add(nn.LogSoftMax())
  -- print(model:forward({x_ww, x_cw}))

  -- model:add(nn.LookupTable(nwords, nclasses))
  -- -- print(model:forward(x_ww))
  -- model:add(nn.Mean(2))
  -- model:add(nn.Add(nclasses))
  -- model:add(nn.LogSoftMax())

elseif structure == 'mlp' or structure == 'lb' then
  if embed == 'y' then
    print('Building multilayer perceptron model with pretrained embeddings...')
  else
    print('Building multilayer perceptron model...')
  end

  -- Use two parallel sequentials to support LookupTables with Reshape
  -- Word LookupTable
  local word_lookup = nn.Sequential()
  local w = nn.LookupTable(nwords, embedding_size)
  if embed == 'y' then -- Pretrained embed init
    for i = 1, nwords do
      w.weight[{i}] = embeddings[{i}]
    end
  end

  local w_reshape = nn.Reshape(dwin * embedding_size)
  word_lookup:add(w):add(w_reshape)

  -- Cap LookupTable
  local cap_lookup = nn.Sequential()
  local c = nn.LookupTable(ncaps, caps_size)
  local c_reshape = nn.Reshape(dwin * caps_size)
  cap_lookup:add(c):add(c_reshape)

  local par = nn.ParallelTable()
  par:add(word_lookup)
  par:add(cap_lookup)
  model:add(par)
  model:add(nn.JoinTable(2))

  if structure == 'mlp' then
    model:add(nn.Linear(din, dhid))
    model:add(nn.HardTanh())
```

```
    model:add(nn.Linear(dhid, dout))
    -- model:add(nn.ReLU())
    -- model:add(nn.Dropout(0.5))
    -- model:add(nn.HardTanh())
    -- model:add(nn.Linear(dhid2, dout))
  elseif structure == 'lb' then
    model:add(nn.Linear(din, dout))
  end

  model:add(nn.LogSoftMax())
else
  print('Classifier incorrectly specified, bailing out.')
  return
end

local nll = nn.ClassNLLCriterion()
-- nll.sizeAverage = false

local params, gradParams = model:getParameters()

local accuracy = {}
function train(e)
  -- Package selected dataset into minibatches
  local selected_x_ww = train_input_word_windows
  local selected_x_cw = train_input_cap_windows
  local selected_y = train_output
  local n_train_batches = math.floor(selected_x_ww:size(1) / batch_size) - 1

  order = torch.randperm(selected_x_ww:size(1))
  -- subsets = torch.range(1,word_order:size(1)-subset_size, subset_size)

  print('\nBeginning epoch ' .. e .. ' training: ' .. n_train_batches .. ' minibatches of size
' .. batch_size .. '.')
  for i = 1, n_train_batches do
    -- n_train_batches do

    local batch_start = torch.random(i*batch_size+1, (i+1)*batch_size)
    local batch_end = math.min((batch_start + batch_size - 1), order:size(1))
    -- --     -- print(batch_start .." - " .. batch_end)

    curr_batch = order[{{batch_start,batch_end}}]:long()
    local x_ww = selected_x_ww:index(1, curr_batch)
    local x_cw = selected_x_cw:index(1, curr_batch)

    -- local batch_start = torch.random(1, selected_x_ww:size(1) - batch_size)
    -- -- local batch_start = (i - 1) * batch_size + 1
    -- local batch_end = batch_start + batch_size - 1
```

```
    -- local range = torch.range(batch_start, batch_end):long()
    -- local x_ww = selected_x_ww:index(1, range)

    -- local x_cw = selected_x_cw:index(1, range)
    local y = selected_y:index(1, curr_batch)

    -- Compute forward and backward pass (predictions + gradient updates)
    function run_minibatch(p)
      if params ~= p then
        params:copy(p)
      end

      -- Accumulate gradients from scratch each minibatch
      gradParams:zero()

      -- Forward pass
      local preds = model:forward({x_ww, x_cw})
      local loss = nll:forward(preds, y)

      if i % 2000 == 0 then
        print('Completed ' .. i .. ' minibatches.')
        -- print('Loss after ' .. batch_end .. ' examples: ' .. loss)
      end

      -- Backward pass
      local dLdpreds = nll:backward(preds, y)
      model:backward(preds, dLdpreds)

      return loss, gradParams
    end

    options = {
      learningRate = eta,
      learningRateDecay = 0.0001
      -- alpha = 0.95 -- For rmsprop
      -- momentum = 0.5
    }

    -- Use optim package for minibatch sgd
    optim.sgd(run_minibatch, params, options)
    -- optim.adagrad(run_minibatch, params, options)
    -- optim.rmsprop(run_minibatch, params, options) -- Slower
  end
end

function test(x_ww, x_cw, y)
```

```
  local preds = model:forward({x_ww, x_cw})
  local loss = nll:forward(preds, y)
  local max, yhat = preds:max(2)

  local correct = yhat:int():eq(y):double():mean() * 100
  return correct, loss
end

function valid_acc()
  return test(valid_input_word_windows, valid_input_cap_windows, valid_output)
end

function train_acc()
  return test(train_input_word_windows, train_input_cap_windows, train_output)
end

print('Validation accuracy before training: ' .. valid_acc() .. ' %.')
print('Beginning training...')
local vloss = torch.DoubleTensor(n_epochs) -- valid/train loss/accuracy/time
local tloss = torch.DoubleTensor(n_epochs)
local vacc = torch.DoubleTensor(n_epochs)
local tacc = torch.DoubleTensor(n_epochs)
local etime = torch.DoubleTensor(n_epochs)

for i = 1, n_epochs do
  if i >= 10 then
    eta = 0.01
  end

  local timer = torch.Timer()
  train(i)

  local va, vl = valid_acc()
  local ta, tl = train_acc()

  vloss[i] = vl
  tloss[i] = tl
  vacc[i] = va
  tacc[i] = ta
  etime[i] = timer:time().real

  print('Epoch ' .. i .. ' training completed in ' .. timer:time().real .. ' seconds.')
  print('Validation accuracy after epoch ' .. i .. ': ' .. va .. ' %.')

  local flr = torch.DiskFile('training_output/lrepoch=' .. i .. '.txt', 'w')
  for j = 1, i do
    flr:writeString(vacc[j] .. ',' .. tacc[j] .. ',' .. vloss[j] .. ','  .. tloss[j] .. ','  .. etime[j] .. '\n')
```

```
    end
    flr:close()
  end

  print('Writing to file...\n')

  local f = torch.DiskFile('training_output/mlp_anneal_test_eta=' .. eta .. '.txt', 'w')
  f:seekEnd()
  f:writeString('\nMLP hyperparams: eta=' .. eta .. ', dhid1=' .. dhid .. ', dhid2=' .. dhid2 ..',
dwin=' .. dwin .. ', pretrainedembed=' .. embed)

  f:writeString('\nValid Acc, Train Acc, Valid Loss, Train Loss, Time\n')

  for i = 1, n_epochs do
    f:writeString(vacc[i] .. ',' .. tacc[i] .. ',' .. vloss[i] .. ','  .. tloss[i] .. ','  .. etime[i] .. '\n')
  end
  f:close()

  -- Write test results to file
  local test_preds = model:forward({test_input_word_windows,
test_input_cap_windows})
  local pred_val, pred_idx = torch.max(test_preds, 2)
  writeToFile(pred_idx)
end

function main()
  -- Parse input params
  opt = cmd:parse(arg)
  classifier = opt.classifier
  alpha = opt.alpha
  eta = opt.eta
  lambda = opt.lambda
  n_epochs = opt.n_epochs
  batch_size = opt.m
  embed = opt.embed

  local f = hdf5.open(opt.datafile, 'r')

  -- Training, valid, and test windows
  train_input_word_windows = f:read('train_input_word_windows'):all()
  train_input_cap_windows = f:read('train_input_cap_windows'):all()
  train_output = f:read('train_output'):all()

  valid_input_word_windows = f:read('valid_input_word_windows'):all()
  valid_input_cap_windows = f:read('valid_input_cap_windows'):all()
  valid_output = f:read('valid_output'):all()
```

```
  test_input_word_windows = f:read('test_input_word_windows'):all()
  test_input_cap_windows = f:read('test_input_cap_windows'):all()

  -- Useful values across models
  nwords = f:read('nwords'):all():long()[1]
  nclasses = f:read('nclasses'):all():long()[1]
  dwin = f:read('dwin'):all():long()[1]
  ncaps = train_input_cap_windows:max()
  -- embeddings = f:read('matrix'):all()
  dhid = 300
  eta = 0.01

  -- Run models
  if opt.classifier == 'nb' then
    naive_bayes()
  else
    model(opt.classifier)
  end
end

main()
```