Practical Worksheet 8

Version: 1.0 Date: 12/04/2018 Author: David Glance

## Learning Objectives

1. Build out a binary classification model using AWS Machine Learning
2. Explore parameters that affect the model’s training and evaluation process
3. Run the equivalent model using Python and Apache MXNet

## Technologies Covered

Ubuntu

AWS Machine Learning

Python

Apache MXNet

## Background

The aim of this lab is to write a program that will:

[1] The principles of the binary classifier using the AWS Machine Learning tutorial

[2] Understand how the classifier uses banking data to decide who is likely to open a deposit account

[3] Understand how to interpret the predictive performance of the model and set score thresholds

[4] Using a python program, understand the underlying steps in the training and testing of data using Apache MXNet

## Binary Classifier Model

## [Step 1] Follow the AWS Machine Learning Tutorial

The aim of this is to follow the steps in the tutorial outlined:

<https://docs.aws.amazon.com/machine-learning/latest/dg/tutorial.html>

Once you have cleaned up, you will need a slightly modified data set for the second part of the lab

## [Step 2] Follow the AWS Machine Learning Tutorial

### Data Preparation

To process the file banking.csv, we need to turn categorical labels that have multiple options into individual boolean labels. An example is the job type which can be one of:

1. Admin
2. Blue Collar
3. Entrepreneurial
4. Housemaid
5. Manager
6. Retired
7. Self Employed
8. Services
9. Student
10. Technician
11. Unemployed

The process of converting a field with one of these types into a separate true/false field is called One Hot Encoding (ref: https://en.wikipedia.org/wiki/One-hot).

The second process is to standardize (a form of normalization) variables that are scalar to have a mean of zero and a unit variance using the formula

where is the original feature, is the mean, and is its standard deviation.

The final step is splitting the file into two sets – a training set and a test set.

These steps have been done for you and the files are:

bankingtest.csv

bankingtrain.csv

### Apache MXNet

MXNet is a python framework that is installed:

pip install mxnet --pre

sudo apt-get install graphviz

pip install graphviz

Writing a binary classifier with logistic regression

We will not be considering the maths behind this process. The basic steps are as follows:

import mxnet as mx

from mxnet import nd, autograd, gluon

import matplotlib.pyplot as plt

with open("./bankingtrain.csv") as f:

train\_raw = f.read()

with open("./bankingtest.csv") as f:

test\_raw = f.read()

data\_ctx = mx.cpu()

model\_ctx = mx.cpu()

# initialise with zeros two vectors for labels Y and features X

# load the vectors with data from the files

def process\_data(raw\_data):

lines = raw\_data.splitlines()

num\_examples = len(lines)

num\_features = 63

X = nd.zeros((num\_examples, num\_features), ctx=data\_ctx)

Y = nd.zeros((num\_examples, 1), ctx=data\_ctx)

for i, line in enumerate(lines):

tokens = line.split(',')

label = int(tokens[63])

Y[i] = label

index = 0

for token in tokens[0:63]:

X[i, index] = float(token)

index += 1

return X, Y

Xtrain, Ytrain = process\_data(train\_raw)

Xtest, Ytest = process\_data(test\_raw)

At this stage we have two vectors with different shapes each for the training data and test data. We have not done anything particularly difficult. We can now process it.

We can now load this data from the vectors into MXNet using gluon

batch\_size = 64

train\_data = gluon.data.DataLoader(gluon.data.ArrayDataset(Xtrain, Ytrain),

batch\_size=batch\_size, shuffle=True)

test\_data = gluon.data.DataLoader(gluon.data.ArrayDataset(Xtest, Ytest),

batch\_size=batch\_size, shuffle=True)

We are now going to create our neural network. Notice that it only has one node (the 1) – but it is still a neural network. The initialize is to initialise randomly the weights used in the neural network.

net = gluon.nn.Dense(1)

net.collect\_params().initialize(mx.init.Normal(sigma=0.001), ctx=model\_ctx)

We can now create the training optimizer

trainer = gluon.Trainer(net.collect\_params(), 'sgd', {'learning\_rate': 0.01})

The sgd refers to Stochastic Gradient Descent – the mechanism by which the training is varied to optimise the fit of the model. This requires a way of measuring the “loss” at each step that is defined as:

def logistic(z):

return 1. / (1. + nd.exp(-z))

def log\_loss(output, y):

yhat = logistic(output)

return - nd.nansum( y \* nd.log(yhat) + (1-y) \* nd.log(1-yhat))

Now, we can do the code that will carry out the training:

epochs = 10

loss\_sequence = []

num\_examples = len(Xtrain)

for e in range(epochs):

cumulative\_loss = 0

for i, (data, label) in enumerate(train\_data):

data = data.as\_in\_context(model\_ctx)

label = label.as\_in\_context(model\_ctx)

with autograd.record():

output = net(data)

loss = log\_loss(output, label)

loss.backward()

trainer.step(batch\_size)

cumulative\_loss += nd.sum(loss).asscalar()

print("Epoch %s, loss: %s" % (e, cumulative\_loss ))

loss\_sequence.append(cumulative\_loss)

At this point, we have a trained model. We can now test the model on the data we set aside for testing. We can also calculate from that the accuracy of the model. Finally, we can visualise the loss curve for the training process.

plt.figure(num=None,figsize=(8, 6))

plt.plot(loss\_sequence)

# Adding some bells and whistles to the plot

plt.grid(True, which="both")

plt.xlabel('epoch',fontsize=14)

plt.ylabel('average loss',fontsize=14)

num\_correct = 0.0

num\_total = len(Xtest)

for i, (data, label) in enumerate(test\_data):

data = data.as\_in\_context(model\_ctx)

label = label.as\_in\_context(model\_ctx)

output = net(data)

prediction = (nd.sign(output) + 1) / 2

num\_correct += nd.sum(prediction == label)

print("Accuracy: %0.3f (%s/%s)" % (num\_correct.asscalar()/num\_total, num\_correct.asscalar(), num\_total))

plt.show()

Run the program and notice the accuracy.

**TODO**: You can alter the program to calculate the precision and recall by obtaining the number of true positives, true negatives and the total positives and total negatives in the test dataset.

Total True = nd.sum(Ytest == 1)

Total False = nd.sum(Ytest == 0)

To test the number of true positives you can use the following code:

true\_positives = 0.0

npLabel = label.asnumpy()  
npLabel[npLabel == 0] = -1  
modLabel = nd.array(npLabel)  
  
true\_positives += nd.sum(prediction == modLabel)

true\_negatives = 0.0

npLabel = label.asnumpy()  
npLabel[npLabel == 1] = -1  
modLabel = nd.array(npLabel)  
  
true\_negatives += nd.sum(prediction == modLabel)

Calculate the precision and recall. How does this differ from the AWS evaluation?

## Submission and Quiz

Submit the python file you wrote in [2]– respond to the quiz

## Respond to the Quiz

[1] Precision measures

[A] The proportion of true negatives over predicted negatives

[B] The proportion of false negatives over predicted negatives

[C] The proportion of true positives over predicted positives

[D] The proportion of predicted positives over true positives

[2] Considering the definition of a loss function, which of the following is true:

[A] The amount of accuracy that is lost each time data is put through a neural network

[B] Part of an optimization process that minimises the difference between a predicted output and an actual output

[C] Needs to be maximised to increase the accuracy of the predicted output

[D] Not needed in binary classification algorithms