Chen Hao Cheng

1. Perceptron

#!/bash/python

#The perceptron equation is S = sum(wi x xi) from i = 0 to i = n

#The function of the separated line is f(s) = 1 if S >= 0, 0 otherwise. I call it

#a step funciton

from random import choice

from numpy import array, dot, random

unitStep = lambda x: 0 if x < 0 else 1

training\_data = [

# array([A, B, C, bias]), expected output)]

# NOTE: bias is always 1

(array([0,0,0]), 0),

(array([0,1,1]), 1),

(array([1,0,1]), 1),

(array([1,1,1]), 1),

]

# uniform gives you a floating-point value from -1 to 1

# Initially, choose 3 random values for weight

w = [random.uniform(-1, 1) for i in range(3)]

print("Random weights are: ", w)

#The errors list is only used to store the error values so that they can be plotted later on

errors = []

# ETA controls the learning rate

ETA = 0.2

n = 8001

for i in xrange(n):

x, expected = choice(training\_data)

result = dot(w, x)

#we can compare to the expected value. If the expected value is bigger, we need to increase the weights, if it's smaller, we need to decrease them

error = expected - unitStep(result)

errors.append(error)

w += ETA \* error \* x

#print("w: ", w)

if i % 250 == 0:

print(i)

print("weight is: ", w)

print("unitStep: ", unitStep(result))

#print("error array: ", errors)

for x, \_ in training\_data:

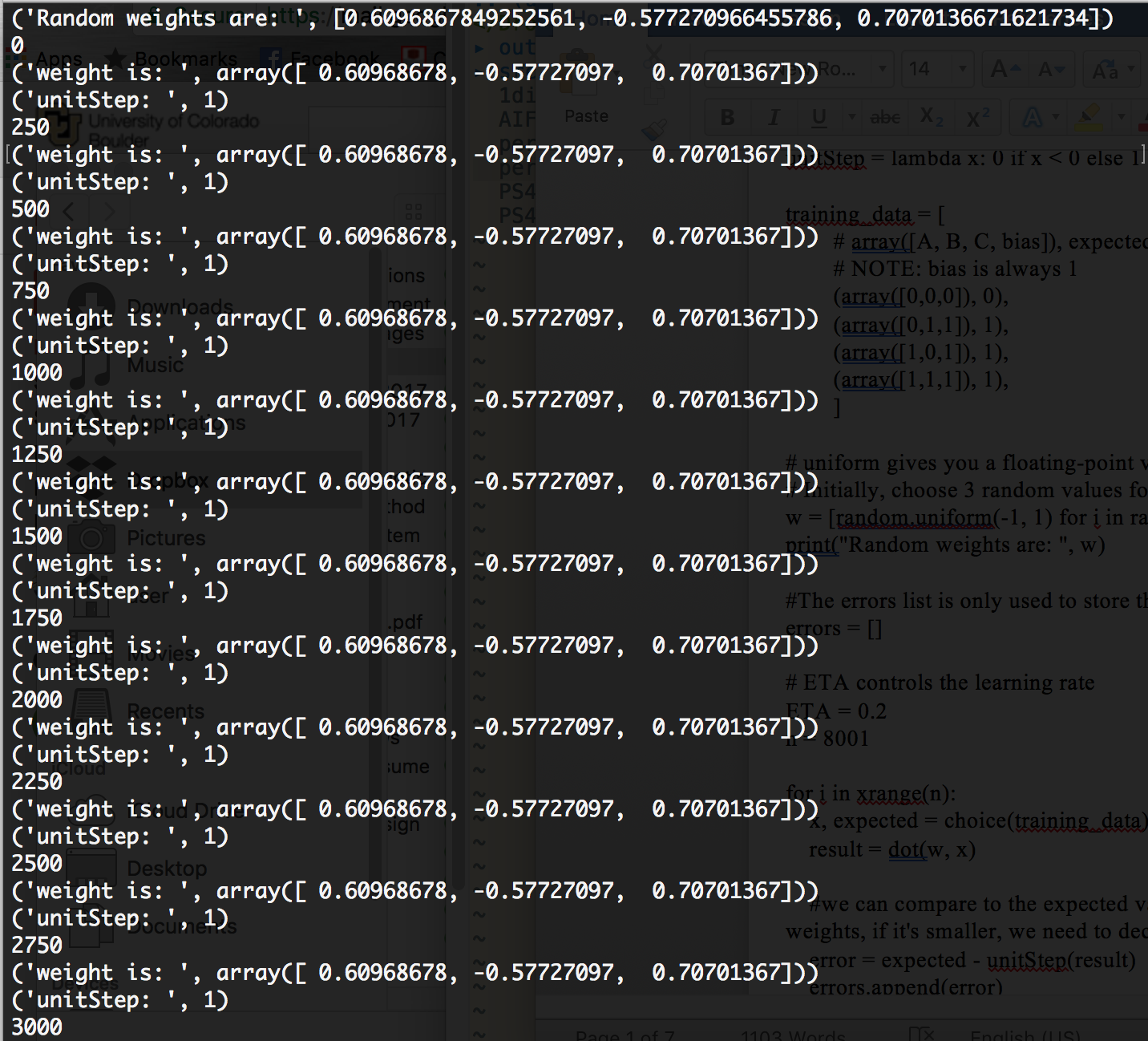
#print("x: ", x)

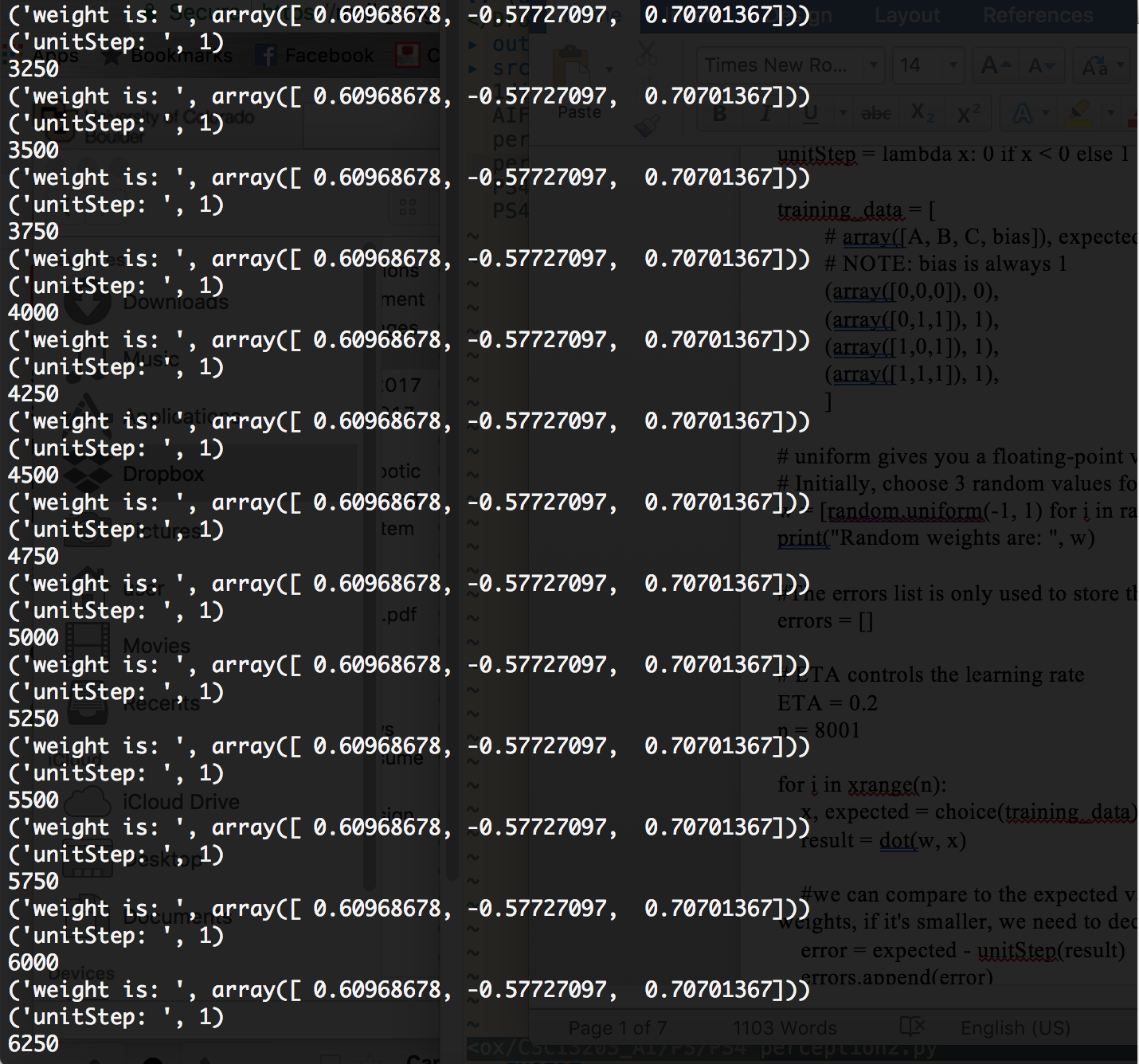
#print("\_: ", \_)

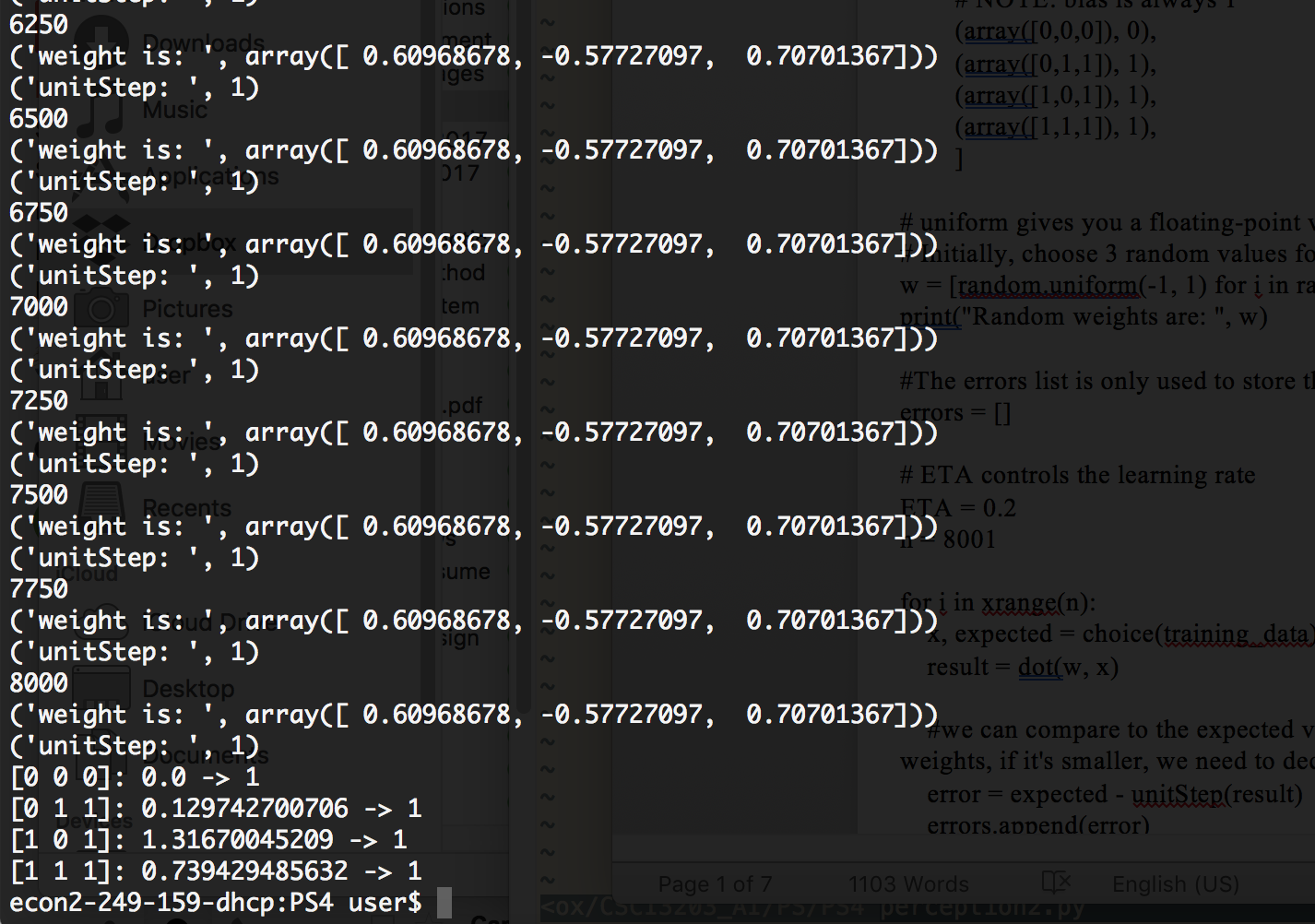
#print("w: ", w)

result = dot(x, w)

print("{}: {} -> {}".format(x[:3], result, unitStep(result)))







There is no error before looping 250 times already, and the weight doesn’t change anymore. I spoke to professor about this, he mentioned this depends on how we train our data. Also, we aren’t supposed to train the input D because it’s a bias.

1. Schelling neighborhood model

#!/bash/python3

#each element of neighbotModel is a house, we begin with all random int from 0 to 2 without

# beyond 6 for 0. 27 for 1, 27 for 2.

#I generat a value(num), which is an index of list, neighborModel and compare its neighbots

#, if it has two same kind of types, it will print "The number is satisfied", not otherwise.

# If the random number in the list is satisfied, the list doesn't swap. If it's not, then swap.

# Choose a random index for the list and output 6 more indexes from the chosen index

from random import randint

from collections import Counter

neighborModel = []

def generateRanArr():

keepGenerating = False

numCounter = 3

zeroCounter = 1 # < 6

oneCounter = 1 # < 27

twoCounter = 1 # < 27

while(not keepGenerating):

m = randint(0, 2)

# Make the first 3 elements different

if m not in neighborModel:

neighborModel.append(m)

else:

if m == 0 and zeroCounter < 6:

neighborModel.append(m)

zeroCounter = zeroCounter + 1

elif m == 1 and oneCounter < 27:

neighborModel.append(m)

oneCounter = oneCounter + 1

elif m == 2 and twoCounter < 27:

neighborModel.append(m)

twoCounter = twoCounter + 1

if(zeroCounter + oneCounter + twoCounter == 60):

keepGenerating = True

#Check a random dissatisfied occupant

def checkSatisfied(num, indexZero, numZero):

#print("num is: ", num)

#print("numZero: ", numZero)

#print("Old indexZero: ", indexZero)

if(num == 58):

if (neighborModel[num + 1] == neighborModel[num] and neighborModel[0] == neighborModel[num]):

print("The number is satisfied")

elif(num == 59):

if (neighborModel[0] == neighborModel[num] and neighborModel[1] == neighborModel[num]):

print("The number is satisfied")

# Check if the random occupant is happy

elif (neighborModel[num + 1] == neighborModel[num] and neighborModel[num + 2] == neighborModel[num]):

print("The number is satisfied")

elif (neighborModel[num - 1] == neighborModel[num] and neighborModel[num -2] == neighborModel[num]):

print("The number is satisfied")

else:

print("The number is NOT satisfied")

#print("trackIndexZeroList: ", indexZero[numZero])

swap = neighborModel[num]

neighborModel[num] = neighborModel[indexZero[numZero]]

neighborModel[indexZero[numZero]] = swap

indexZero = [i for i, index in enumerate(neighborModel) if index == 0]

#print("new indexZero: ", indexZero)

#print ("new neighborModel: ", neighborModel)

generateRanArr()

counter = 1

i = 1

for i in range(60):

first = []

last = []

#print(Counter(neighborModel))

#print("Old neighborModel: ", neighborModel)

# find indexes of 0 (empty occupant) in list

indexZero = [i for i, index in enumerate(neighborModel) if index == 0]

# Pick a value from 1 to 60

num = randint(0, 59)

# Pick a random value for indexZero

numZero = randint(0, 5)

checkSatisfied(num, indexZero, numZero)

counter = counter + 1

# Output from a random index of list to 5 values more

if(counter % 20 == 0):

ranSixDigitIndex = randint(0, 59)

print("ranSixDigit: ", ranSixDigitIndex)

# Handle some certain cases, and extend the output array

if(ranSixDigitIndex == 55):

first = neighborModel[:1]

last = neighborModel[-5 : ]

last.extend(first)

print(last)

#print(neighborModel[-5: 0])

elif(ranSixDigitIndex == 56):

first = neighborModel[:2]

last = neighborModel[-4 : ]

last.extend(first)

print(last)

#print(neighborModel[-4: 1])

elif(ranSixDigitIndex == 57):

first = neighborModel[:3]

last = neighborModel[-3 : ]

last.extend(first)

print(last)

#print(neighborModel[-3 : 2])

elif(ranSixDigitIndex == 58):

first = neighborModel[:4]

last = neighborModel[-2 : ]

last.extend(first)

print(last)

#print(neighborModel[-2 : 3])

elif(ranSixDigitIndex == 59):

first = neighborModel[ : 5]

last = neighborModel[-1: ]

last.extend(first)

print(last)

#print(neighborModel[-1 : 4])

else:

print("six digits neighborModel", neighborModel[ranSixDigitIndex: ranSixDigitIndex + 6])

print(counter)

Output:

six digits neighborModel [1, 1, 1, 1, 1, 1]

20

six digits neighborModel [0, 1, 2, 0, 2, 2]

40

six digits neighborModel [1, 2, 1, 1, 1, 2]

60

six digits neighborModel [1, 0, 1, 2, 1, 2]

80

six digits neighborModel [2, 2, 2, 2, 1, 2]

100

six digits neighborModel [1, 1, 1, 1, 1, 1]

120

six digits neighborModel [2, 1, 2, 2, 2, 2]

140

six digits neighborModel [1, 1, 1, 1, 1, 1]

160

six digits neighborModel [1, 1, 1, 1, 2, 2]

180

six digits neighborModel [1, 1, 1, 1, 1, 1]

200

six digits neighborModel [2, 2, 2, 2, 1, 1]

220

six digits neighborModel [1, 1, 1, 1, 1, 1]

240

six digits neighborModel [2, 2, 2, 2, 2, 1]

260

six digits neighborModel [2, 2, 2, 2, 1, 1]

280

six digits neighborModel [1, 1, 1, 1, 1, 1]

300

[1, 1, 1, 1, 1, 2]

320

six digits neighborModel [2, 2, 2, 2, 2, 2]

340

six digits neighborModel [1, 1, 1, 1, 1, 1]

360

six digits neighborModel [2, 2, 2, 1, 1, 1]

380

[1, 1, 1, 0, 0, 0]

400

new neighborModel:  [1, 1, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

The “ring city’ does move toward a “totally satisfied” state every time I run it. Although sometimes it is just close to the state of “totally satisfied”, it is a really beautiful idea of swap. In some way, this looks like a sorting algorithm like the selection sort.

This time (the picture above), it is completely satisfied! I am really surprised this happens because when I got this assignment and read it, I told the professor that this will never be the totally satisfied state.