Shane Sarnac
CSCI 3202
Problem Set 4
Fall 2017

Problem 4.1

Output:

```
Iteration: 0, Weights = -0.8656469494051806,-0.4609708697328927,-0.5316674550785798,-1
Iteration: 5, Weights = -0.5915395889725564,-0.4444115615288309,-0.29933373144198683,-1
Iteration: 10, Weights = -0.19467641233814018,-0.38444928145530255,0.04627211206055163,-1
Iteration: 15, Weights = 0.2928292642736138,-0.3113176196883515,0.47404248808605826,-1
Iteration: 20, Weights = 0.6193647295933138,-0.351794759366568,0.7490838134825926,-1
Iteration: 25, Weights = 0.7705855078015033,-0.4635135872067986,0.8652260918048704,-1
Iteration: 30, Weights = 0.8508143280790135,-0.5701097861066806,0.9219655226447535,-1
Iteration: 35, Weights = 0.9017717488905633,-0.6545547627962289,0.9568444684090099,-1
Iteration: 40, Weights = 0.9373488120989167,-0.7186617742233319,0.9812127402487848,-1
Iteration: 45, Weights = 0.9632571962551736,-0.7669859842501556,0.999196765409892,-1
Iteration: 50, Weights = 0.9825035915602022,-0.8034910980378258,1.0127787809861089,-1
Iteration: 55, Weights = 0.9969561973032568,-0.8311799273883459,1.0231468048725996,-1
Iteration: 60, Weights = 1.0078822324660115,-0.8522643688634648,1.0311066780213263,-1
Iteration: 65, Weights = 1.0161804696813772,-0.868372934770187,1.0372386907588087,-1
Iteration: 70, Weights = 1.0225042007315759,-0.8807126331710964,1.0419731026696835,-1
Iteration: 75, Weights = 1.0273355337494832,-0.890185007890075,1.0456339740036686,-1
Iteration: 80, Weights = 1.031033932734453,-0.8974681719829538,1.0484676810899738,-1
Iteration: 85, Weights = 1.0338694238746993,-0.9030751691988965,1.0506626909271446,-1
Iteration: 90, Weights = 1.0360459879538455,-0.9073959898890731,1.0523637835012971,-1
Iteration: 95, Weights = 1.0377183775122096,-0.9107281964627358,1.0536825130596523,-1
```

Note: The weights in the output given above are listed in the order: A, B, C, D. For this output, I let $\alpha = 0.5$, which allowed the weights to normalize quickly. While these edge weights are approaching the theoretical solution of (1,-1,1,-1), the errors remain pretty high and consistent, even with many iterations.

Note also, that I tried running this program with various α values, including 0.1 and 0.01 with similar results. However, while the relative magnitude is about the same for each weight, the magnitude of edge B is always slightly smaller than the magnitude of each A and C, which gives correct (albeit barely) results when rounding the output function.

Problem 4.2
Output:

```
011000021021221121121212222121112122222111221111122121121222
0111221212212200111012222222111122222222111211111120121101222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
111102222222222201110022222222111122222222111111111111111100222
```

Note: Each line represents the city after 20 iterations for a total of 400 iterations. Note that the city becomes stable within the first 60 iterations. Consider the following which shows the city after 100 iterations, each line representing 5 iterations:

```
2000120122110022111212211222212111212211221122212121221111
2201121102110202111212010222212111222211221122212121221111
0201121102111202111212112222212111222201221022212121221111
2201101110111202111012112222212111222201221222222121221111
2211111110111202111012102222212111222221020222222121221111
2011111110111200111112122222212111222202212222221212022111
0211111111111010111110122222202111222222221222222120222211111
0011111111111011111111222222202111222222221222222202022221111
1011111111111011111111222222200111222222220222222222022221111
1011111111111011111111222222200111222222220222222222022221111
1011111111111011111111222222200111222222220222222222022221111
1011111111111011111111222222200111222222220222222222022221111
1011111111111011111111222222200111222222220222222222022221111
1011111111111011111111222222200111222222220222222222022221111
1011111111111011111111222222200111222222220222222222022221111
1011111111111011111111222222200111222222220222222222022221111
1011111111111011111111222222200111222222220222222222022221111
1011111111111011111111222222200111222222220222222222022221111
1011111111111011111111222222200111222222220222222222022221111
1011111111111011111111222222200111222222220222222222022221111
```

Much like the first output above, the city reaches a steady state within 60 iterations (albeit normalizing around 50 iterations). This shows that neighborhoods will form if it is assumed all people will want to have two neighbors be like them as the clusters are pretty clearly identifiable.

Problem 1 Code: perceptron.js

```
function Perceptron(iterations, printing_frequency) {
	var data_set = [
		[0, 0, 0, 1],
		[1, 0, 0, 1],
		[0, 1, 0, 1],
		[0, 0, 1, 1],
		[1, 1, 0, 1],
		[0, 1, 1, 1],
		[1, 0, 1, 1],
		[1, 1, 1, 1]
	];
	var expected = [0, 0, 0, 0, 0, 0, 1, 1];
	var nodes = [];
	var node_weights = [];
	var threshold = 0.5;
	// Set initial edge weights
	node_weights = defineRandomWeights(3);
	node_weights = node_weights.concat(-1);
	runSimulation(iterations, printing_frequency, data_set, expected, node_weights);
};

function runSimulation(iterations, printing_frequency, data_set, expected, node_weights) {
	var sum, output, error;
	for (var i = 0; i < iterations; i++) {
		if (i % printing_frequency == 0) {
			printWeights(i, node_weights);
		}
		for (var j = 0; j < data_set.length; j++) {
			sum = calculateSum(data_set[j], node_weights);
			output = outputFunction(sum);
			error = calculateError(output, expected[j]);
			if (i % printing_frequency == 0) {
				//printResults(data_set[j], expected[j], output);
			}
			for (var k = 0; k < node_weights.length-1; k++) {
				node_weights[k] = adjustWeight(sum, error, data_set[j][k],
node_weights[k]);
			}
			/*console.log("edge_weights = " + node_weights + " (after) \n");*/
			/*if (i == iterations-1) {
				printResults(data_set[j], expected[j], output);
			}*/
		}
	}
}
```

```
function adjustWeight(sum, error, node_value, node_weight) {
	var alpha = 0.01;
	var new_weight = node_weight + alpha * error * outputFunctionPrime(sum) * node_value;
	return new_weight;
}

function calculateSum(node_values, node_weights) {
	var sum = 0;
	for (var i = 0; i < node_values.length; i++) {
		sum += node_weights[i] * node_values[i];
	}
	return sum;
}

function sigmoidalFunction(x) {
	return 1 / (1 + Math.exp(-x));
}

function outputFunction(sum) {
	return sigmoidalFunction(sum);
}

function outputFunctionPrime(sum) {
	return sigmoidalFunction(sum) * (1 - sigmoidalFunction(sum));
	//return Math.exp(sum) / Math.pow(Math.exp(sum) + 1, 2);
}

function calculateError(output, expected) {
	return expected - output;
}

function main() {
	new Perceptron(10000, 500);
}
main()
```

Problem 2 Code: neighborhood.py

```
class House:
	occupied = 0
	def __init__(self, occ):
		if occ < 0:
			print("invalid occupant")
			exit()
		self.occupied = occ

	def isOccupied(self):
		return self.occupied != 0

	def getOccupant(self):
		return self.occupied

class Neighborhood:
	total_ones = 0
	total_twos = 0
	total_empty = 0
	neighborhood = []
	empty = [] # holds the indices of empty homes in the neighborhood

	def __init__(self, ones, twos, empty):
		self.total_ones = ones
		self.total_twos = twos
		self.total_empty = empty
		self.buildNeighborhood()
		#self.printNeighborhood()

	def buildNeighborhood(self):
		ones_remaining = self.total_ones
		twos_remaining = self.total_twos
		empty_remaining = self.total_empty


		while ones_remaining > 0 or twos_remaining > 0 or empty_remaining > 0:
			occupant = randint(0,2)
			if occupant == 2 and twos_remaining > 0:
				self.neighborhood.append(House(occupant))
				twos_remaining = twos_remaining - 1
			elif occupant == 1 and ones_remaining > 0:
				self.neighborhood.append(House(occupant))
				ones_remaining = ones_remaining - 1
			elif occupant == 0 and empty_remaining > 0:
				self.neighborhood.append(House(occupant))
				empty_remaining = empty_remaining - 1
				self.empty.append(len(self.neighborhood)-1)
```

```
def printNeighborhood(self):
	neighborhood_str = ""
	zeroes_str = ""
	for i in range(len(self.neighborhood)):
		neighborhood_str += str(self.neighborhood[i].getOccupant())

	for i in range(len(self.empty)):
		zeroes_str += str(self.empty[i]) + ","
	# print(neighborhood_str + "     " + zeroes_str)
	# print(neighborhood_str + "\n")
	print(neighborhood_str)

def findDissatisfied(self):
	neighborhood_size = len(self.neighborhood)
	dissatisfied = [] # contains index of dissatisfied occupants
	left_1 = 0
	left_2 = 0
	right_1 = 0
	right_2 = 0

	for i in range(len(self.neighborhood)):
		preferred_neighbor_count = 0
		left_2 = self.neighborhood[(i-2)%neighborhood_size].getOccupant()
		left_1 = self.neighborhood[(i-1)%neighborhood_size].getOccupant()
		occupant = self.neighborhood[i].getOccupant()
		right_1 = self.neighborhood[(i+1)%neighborhood_size].getOccupant()
		right_2 = self.neighborhood[(i+2)%neighborhood_size].getOccupant()

		if occupant == left_2:
			preferred_neighbor_count += 1
		if occupant == left_1:
			preferred_neighbor_count += 1
		if occupant == right_1:
			preferred_neighbor_count += 1
		if occupant == right_2:
			preferred_neighbor_count += 1

		if preferred_neighbor_count < 2 and occupant != 0:
			dissatisfied.append(i)
	return dissatisfied
```

```
    def moveDissatisified(self, dissatisfied):
        if len(dissatisfied) == 0:
            return
        random_dissatisfied_index = dissatisfied[randint(0, len(dissatisfied)-1)]
        random_empty_index = randint(0, len(self.empty)-1)

        # get the unhappy resident from the neighborhood and make the home empty
        unhappy_occupant = self.neighborhood[random_dissatisfied_index]
        self.neighborhood[random_dissatisfied_index] = House(0)

        # find a new empty home, move the resident into it,
        # and store the the new empty home in the empty list
        new_home = self.empty[random_empty_index]
        self.neighborhood[new_home] = unhappy_occupant
        self.empty[random_empty_index] = random_dissatisfied_index

    def runSimulation(self, print_iter, max_iter):
        for i in range(max_iter):
            if i % print_iter == 0:
                self.printNeighborhood()
            dissatisfied = self.findDissatisfied()
            self.moveDissatisified(dissatisfied)

def main():
    neighborhood = Neighborhood(27, 27, 6)
    neighborhood.runSimulation(5, 100)
main()
```