(Names, ID's):

יהושע גורדון - תז 332307073

סיוון נעמה עזרי - תז 208479311

אלעד וינברנד - תז 203306014

Part C:

C2 table results:

Skilearns Multi layer perceptron model received an accuracy of 77.5% using back-propagation

We used a 5 layer network (3 hidden layers) of 2, 8, 6, 8, 2 and received an multi array of each weights:

[array([[0.38251742, -0.74827219]]), array([[1.45485634e-001, 2.06844784e-002, 4.58356682e-001,

9.37591746e-151, 8.42873527e-001, 3.49209302e-001,

9.42805812e-120, 9.07108194e-116],

[-9.74426172e-002, 6.71354721e-001, -1.32029139e-001,

3.91745190e-002, 5.02715457e-002, 2.98911521e-001,

1.01683973e-001, 1.47387692e-098]]), array([[2.79584982e-001, 2.79556571e-001, -2.21251407e-001,

2.32451787e-001, 1.30886096e-001, 3.65547830e-001],

[-3.73600633e-002, 5.88092564e-001, 5.42504259e-001,

5.58001166e-001, 5.37037430e-001, 3.66359179e-001],

[4.77119561e-001, -3.58338674e-001, -9.91344133e-004,

-2.97069435e-001, -5.84947014e-001, -9.30605293e-003],

[-1.79410882e-169, -1.92867729e-001, 1.50390957e-001,

2.76116167e-001, 5.13114524e-001, -8.69674611e-002],

[3.75809643e-001, -1.09277458e-002, 3.13064164e-001,

-4.16546885e-001, -4.16332171e-001, -7.99728386e-001],

[3.89013828e-001, 1.18389654e-001, -1.86061057e+000,

-6.20618624e-001, -8.07712502e-001, 8.71670362e-001],

[-2.17357269e-010, -2.92107086e-001, 1.72464527e-001,

-1.00420032e-001, 1.20420263e-001, 1.81033928e-001],

[-9.70094359e-161, 4.87701916e-160, 6.72049812e-134,

3.52314647e-084, 8.99183410e-182, 1.27153413e-156]]), array([[6.94511722e-001,
9.89952913e-002, 7.04953280e-088,

-5.63694333e-001, 2.91939012e-001, 5.13668943e-001,

3.39282122e-001, 5.13303013e-180],

[-6.25304178e-001, -5.48147365e-001, -5.15212785e-084,

-9.08995318e-002, 5.86896729e-001, -8.57112957e-002,

4.79530727e-001, -1.69615839e-130],

[-5.95869410e-002, -5.61873320e-002, 1.08282821e-105,

-2.31980294e-002, -7.67036036e-001, -1.18863663e+000,

4.86437013e-002, -1.06305724e-174],

[5.93653120e-001, -6.27731987e-001, -5.17769230e-168,

-1.39802005e-001, -6.13826538e-001, -4.08381521e-081,

-3.30216551e-001, 4.73785209e-133],

[-1.19793106e-001, 3.53817636e-001, -5.53699348e-095,

-1.10225706e-147, -2.93809456e-002, 3.57312883e-158,

-2.54877087e-001, 2.45171994e-084],

[-2.82769684e-001, 4.93610520e-001, 4.20026569e-080,

-6.41184690e-002, -4.22495477e-001, 5.06158158e-014,

7.42721438e-001, 5.77679337e-137]]), array([[8.68180173e-001, 1.59554016e-091],

[4.01734657e-001, -3.87877795e-158],

[4.57974685e-147, -2.25193763e-157],

[-2.53020958e-002, 4.28655418e-069],

[4.37246443e-001, 3.65863114e-096],

[2.20317089e+000, -1.35459712e-091],

[-6.26075202e-001, -1.77112268e-160],

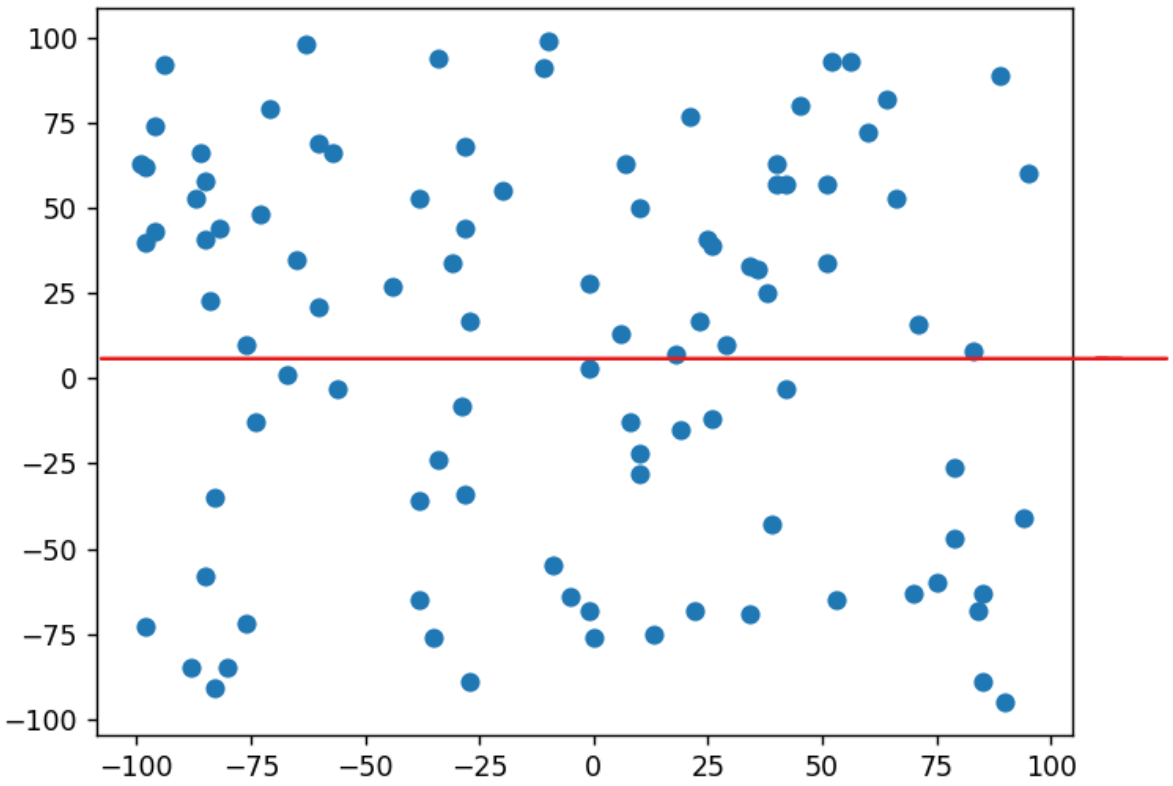
[1.08369590e-164, 1.12322912e-136]]), array([[1.60382769e+00],

[-4.77658335e-35]])]

We received a log loss which is considered good

We then randomly made an array with 500 points, and classified the expected output for each point.

Afterwards, we ran the training data on the algorithm and received results for that training data as well. Only 4 points were classified incorrectly.



We received the table for the expected output, vs the output we received from training the neuron network.

X: -1	Y: -68	Expected: 0,	Got: 0
X: -82	Y: 44	Expected: 1,	Got: 1

X: 18	Y: 7	Expected: 0,	Got: 1
X: -10	Y: 99	Expected: 1,	Got: 1
X: 34	Y: 33	Expected: 1,	Got: 1
X: 40	Y: 63	Expected: 1,	Got: 1
X: -1	Y: 28	Expected: 1,	Got: 1
X: -98	Y: 62	Expected: 1,	Got: 1
X: -76	Y: -72	Expected: 0,	Got: 0
X: -83	Y: -35	Expected: 0,	Got: 0
X: 56	Y: 93	Expected: 1,	Got: 1
X: -71	Y: 79	Expected: 1,	Got: 1
X: 40	Y: 57	Expected: 1,	Got: 1
X: 84	Y: -68	Expected: 0,	Got: 0
X: -34	Y: -24	Expected: 0,	Got: 0
X: -67	Y: 1	Expected: 0,	Got: 1
X: -74	Y: -13	Expected: 0,	Got: 0
X: 51	Y: 57	Expected: 1,	Got: 1
X: -28	Y: -34	Expected: 0,	Got: 0
X: -31	Y: 34	Expected: 1,	Got: 1
X: 79	Y: -26	Expected: 0,	Got: 0
X: -85	Y: 58	Expected: 1,	Got: 1
X: 64	Y: 82	Expected: 1,	Got: 1
X: -28	Y: 44	Expected: 1,	Got: 1
X: 6	Y: 13	Expected: 1,	Got: 1
X: -76	Y: 10	Expected: 1,	Got: 1
X: -20	Y: 55	Expected: 1,	Got: 1
X: 94	Y: -41	Expected: 0,	Got: 0
X: -38	Y: 53	Expected: 1,	Got: 1
X: -9	Y: -55	Expected: 0,	Got: 0
X: 70	Y: -63	Expected: 0,	Got: 0
X: -63	Y: 98	Expected: 1,	Got: 1
X: -28	Y: 68	Expected: 1,	Got: 1
X: 90	Y: -95	Expected: 0,	Got: 0
X: -98	Y: -73	Expected: 0,	Got: 0
X: -73	Y: 48	Expected: 1,	Got: 1

X: -98	Y: 40	Expected: 1,	Got: 1
X: 21	Y: 77	Expected: 1,	Got: 1
X: 7	Y: 63	Expected: 1,	Got: 1
X: -35	Y: -76	Expected: 0,	Got: 0
X: -99	Y: 63	Expected: 1,	Got: 1
X: 42	Y: -3	Expected: 0,	Got: 0
X: -60	Y: 21	Expected: 1,	Got: 1
X: -11	Y: 91	Expected: 1,	Got: 1
X: 34	Y: -69	Expected: 0,	Got: 0
X: -27	Y: -89	Expected: 0,	Got: 0
X: 75	Y: -60	Expected: 0,	Got: 0
X: 38	Y: 25	Expected: 1,	Got: 1
X: 8	Y: -13	Expected: 0,	Got: 0
X: 10	Y: -28	Expected: 0,	Got: 0
X: 10	Y: -22	Expected: 0,	Got: 0
X: -44	Y: 27	Expected: 1,	Got: 1
X: 71	Y: 16	Expected: 1,	Got: 1
X: 60	Y: 72	Expected: 1,	Got: 1
X: 26	Y: -12	Expected: 0,	Got: 0
X: -80	Y: -85	Expected: 0,	Got: 0
X: -83	Y: -91	Expected: 0,	Got: 0
X: -96	Y: 43	Expected: 1,	Got: 1
X: -29	Y: -8	Expected: 0,	Got: 0
X: 23	Y: 17	Expected: 1,	Got: 1
X: 13	Y: -75	Expected: 0,	Got: 0
X: 19	Y: -15	Expected: 0,	Got: 0
X: 85	Y: -63	Expected: 0,	Got: 0
X: 42	Y: 57	Expected: 1,	Got: 1
X: -38	Y: -65	Expected: 0,	Got: 0
X: 85	Y: -89	Expected: 0,	Got: 0
X: -96	Y: 74	Expected: 1,	Got: 1
X: -5	Y: -64	Expected: 0,	Got: 0
X: 10	Y: 50	Expected: 1,	Got: 1
X: 39	Y: -43	Expected: 0,	Got: 0

X: -85	Y: 41	Expected: 1,	Got: 1
X: 25	Y: 41	Expected: 1,	Got: 1
X: 51	Y: 34	Expected: 1,	Got: 1
X: -1	Y: 3	Expected: 0,	Got: 1
X: 79	Y: -47	Expected: 0,	Got: 0
X: -87	Y: 53	Expected: 1,	Got: 1
X: 26	Y: 39	Expected: 1,	Got: 1
X: -65	Y: 35	Expected: 1,	Got: 1
X: 0	Y: -76	Expected: 0,	Got: 0
X: 53	Y: -65	Expected: 0,	Got: 0
X: -60	Y: 69	Expected: 1,	Got: 1
X: -94	Y: 92	Expected: 1,	Got: 1
X: 95	Y: 60	Expected: 1,	Got: 1
X: -88	Y: -85	Expected: 0,	Got: 0
X: 89	Y: 89	Expected: 1,	Got: 1
X: -27	Y: 17	Expected: 1,	Got: 1
X: 29	Y: 10	Expected: 1,	Got: 1
X: -57	Y: 66	Expected: 1,	Got: 1
X: 83	Y: 8	Expected: 0,	Got: 0
X: 66	Y: 53	Expected: 1,	Got: 1
X: -85	Y: -58	Expected: 0,	Got: 0
X: -86	Y: 66	Expected: 1,	Got: 1
X: 36	Y: 32	Expected: 1,	Got: 1
X: -38	Y: -36	Expected: 0,	Got: 0
X: 45	Y: 80	Expected: 1,	Got: 1
X: -84	Y: 23	Expected: 1,	Got: 1
X: 52	Y: 93	Expected: 1,	Got: 1
X: -56	Y: -3	Expected: 0,	Got: 1
X: -34	Y: 94	Expected: 1,	Got: 1
X: 22	Y: -68	Expected: 0,	Got: 0

C3:

Discussions:

I believe the “bagel problem is an XOR problem” as stated in my previous assignment where we converted our(x,y) coordinates to polar coordinates (r, delta)

There for it is not surprising that our model was able to get better results the the adaline model which can not linear separate our new transformed data as well as our Multi layer perceptron model.

C4 codes:

```
x = np.random.uniform(-10, 10, size= 10000)
y = np.random.uniform(-10, 10, size= 10000)
l = np.where(y>1 , 1, 0)

df_x = pd.DataFrame(x)
df_y = pd.DataFrame(y)
df_l = pd.DataFrame(l)

df_x_y = df_x + df_y

cols = np.where(y > 1 , 'b', 'r')
# showing data
plt.scatter(x,y, c=cols, s=5 , linewidths=0)
plt.show()

X_train, X_test, y_train, y_test = train_test_split(df_x_y, df_l,
test_size=0.2, random_state= 0)

mlp_classifier = MLPClassifier(hidden_layer_sizes=(2, 8, 6 ,8, 2))
mlp_classifier.fit(X_train, y_train)

mlp_score = mlp_classifier.score(X_test, y_test)

coeffs = mlp_classifier.coefs_
loss = mlp_classifier.loss_

print(mlp_score)
print(coeffs)
print(loss)
```

Code:

function named initialize_network() that creates a new neural network ready for training. It accepts three parameters, the number of inputs, the number of neurons to have in the hidden layer and the number of outputs.

```
# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
    network = list()
    hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in range(n_hidden)]
    network.append(hidden_layer)
    output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in range(n_outputs)]
    network.append(output_layer)
    return network

# Calculate neuron activation for an input
def activate(weights, inputs):
    activation = weights[-1]
    for i in range(len(weights)-1):
        activation += weights[i] * inputs[i]
    return activation
```

```
# Transfer neuron activation
def transfer(activation):
    return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output
def forward_propagate(network, row):
    inputs = row
    for layer in network:
        new_inputs = []
        for neuron in layer:
            activation = activate(neuron['weights'], inputs)
            neuron['output'] = transfer(activation)
            new_inputs.append(neuron['output'])
        inputs = new_inputs
    return inputs
```

The network is trained using stochastic gradient descent

```
def transfer_derivative(output):
    return output * (1.0 - output)

# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
    for i in reversed(range(len(network))):
        layer = network[i]
        errors = list()
        if i != len(network)-1:
            for j in range(len(layer)):
                error = 0.0
                for neuron in network[i + 1]:
                    error += (neuron['weights'][j] * neuron['delta'])
                errors.append(error)
        else:
            for j in range(len(layer)):
                neuron = layer[j]
                errors.append(neuron['output'] - expected[j])
        for j in range(len(layer)):
            neuron = layer[j]
            neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])
```

Once errors are calculated for each neuron in the network via the back propagation method above, they can be used to update weights

```
# Update network weights with error
def update_weights(network, row, l_rate):
    for i in range(len(network)):
        inputs = row[:-1]
        if i != 0:
            inputs = [neuron['output'] for neuron in network[i - 1]]
        for neuron in network[i]:
            for j in range(len(inputs)):
                neuron['weights'][j] -= l_rate * neuron['delta'] * inputs[j]
            neuron['weights'][-1] -= l_rate * neuron['delta']
```

Part D:

Now use the trained neurons from the next to last level of Part C as input and only an Adaline for the output. (That is, you will give the adaline the output of the neurons from Part 3 in the level below the output, and train only the Adaline.) Describe how accurate the Adaline can be. Give diagrams

Using the weights from the second last layer (the layer before output) we made an Adaline modle however did not get any reasonable results:

Code:

```
last_weights = coeffs[-2]
print('***************')
print(last_weights)

ada = Adaline
ada.fit(last_weights, np.array([1, 0]))

print('********************')
print(ada.score())
```

We disused and concluded that the fact is that these weights and activation functions on each neuron is random and just taking the last layer wont say or do much for our Adaline model