1a) Initial run:

D - initial dataset.

H(D) = 0.9852

IG(D, sky) = 0.5216

IG(D, air) = 0.1281

IG(D, humid) = 0.0202

IG(D, wind) = 0.2916

IG(D, water) = 0.0059

IG(D, forecast) = 0.0202

=> split on sky as it has the largest IG. Splitting on sky results in all rainy results going to -ve.

Considering remaining sunny results (let these be D1):

IG(D1, air) = 0 (H(D1, warm) = H(D1))

IG(D1, humid) = 0.4852

IG(D1, wind) = 0.8113 (as both strong and weak lead to one class each, so their entropies are 0)

IG(D1, water) = 0.2965

IG(D1, forecast) = 0.2965

So split on wind, which cleanly splits into +ve and -ve classes.

b) We could define a distance function dist(x, row) by comparing x[i] to row[i] over all dimensions i, and adding 1 for each differing value. The sum of differences is returned. A weight function can be defined as weight(x) = 1/x.

Algorithm (written here as pseudocode):

def classify(x):

# Array is ordered so nearest\_dists[0] <= nearest\_dists[1]. Rows correspond to rows with

# nearest dist.

nearest\_rows = [inf, inf]

nearest\_dists = [inf, inf]

for i in range(len(rows)):

row = rows[i]

distance = dist(x, row)

# Assign new rows

if nearest\_dists[0] > distance:

nearest\_dists[1] = nearest\_dists[0]

nearest\_rows[1] = nearest\_rows[0]

nearest\_dists[0] = distance

nearest\_rows[0] = row

elif nearest\_dists[1] > distance:

nearest\_dists[1] = distance

nearest\_rows[1] = row

# All rows have same length.

len\_row = len(rows[0])

# Assuming class is last dimension

fst\_class = nearest\_rows[0][len\_row - 1]

snd\_class = nearest\_rows[0][len\_row - 1]

if fst\_class == snd\_class:

return fst\_class

fst\_weight = weight(nearest\_dists[0])

snd\_weight = weight(nearest\_dists[1])

if fst\_weight > snd\_weight:

return fst\_class

return snd\_class

Applying above defs:

Dists for each row: 3, 2, 3, 2, 4, 5, 3.

Weights: ⅓, ½, ⅓, ½, ¼, ⅕, ⅓.

We take rows 2 and 4 (counting from 1) as they have the lowest distances. Both classes are + so we return +.

c) GA parameters: initial training set (the rows), classes for each row, number of branches we use (to keep as a fixed structure we would need to set the number of branches beforehand and then adjust only the values we split on until we get the splits we want).

Result of applying a single round is a tree of the same structure but with potentially misclassified values per leaf node (so values are in the wrong leaf nodes).

Fitness func: check class for each 1

, add 1 per correct answer (max is 7).

Cross over: swap branches at random (leading to swapping contents of leaf nodes following design of q1a). Pick two to randomly swap per iteration?

Selection operator: biased roulette wheel.

Mutation: swap value/category we split on with probability 1/7 per value.

2a)

cost/error function E = ½ sum(t\_d - o\_d)

w\_0 := w\_0 - alpha \* sum(t\_d - o\_d)

w\_i := w\_i - alpha \* sum(t\_d - o\_d)x\_i

I would suggest that it’s something like

w\_i := w\_i - alpha \* (x\_i + x\_i^3) \* sum(t\_d - o\_d) since it’s the partial diff wrt w\_i

3a) (y^ - predicted val, y - actual val)

E0 = Loss + lambda \* sum of w (abs(w))

Loss = 1/N \* sum of i = 1 to N (y(i) - y ^(i))^2

dLoss/dy^ = (-2 / N) \* sum of i = 1 to N (y(i) - y^(i)) [ minus sign arises from chain rule applied to - y^(i)]

dLoss/dA = dLoss/dy^

dLoss/dZ = dLoss/dA multiplied elementwise by g’(z)

g(z) is sigmoid func (= 1/(1+ e^-(z))), g’(z) = g(z)(1-g(z)

=> dLoss/dZ = (2 / N) \* sum of i = 1 to N (y(i) - y^(i)) \* g’(z)

and dLoss/dW = X^T \* dLoss/dZ.

Update rule per weight w:

w = w - alpha\*(dLoss/dw + lambda\*sign(w))

======== ALT

Loss = MSE = ½ (Y^ - Y)^2

Loss’ = Loss + L1 Regularization = ½ (Y^ - Y)^2 + sum i to n abs(w\_i)

Where ‘n’ is the number of neurons in the output layer (not important tho)

Y^ = sum i to n (w\_i \* y\_i) + b # forward pass definition

dLoss / dY^ = Y^ - Y

dY^ / dw\_i = y\_i

dLoss' / dw\_i = (dLoss / dw\_i) + d(sum i to n abs(w\_i)) / dw\_i

# By distributing the derivative over the 2 summands in Loss’

= (dLoss / dw\_i) + lambda\*sign(w\_i)

# Obtain “sign” since the derivative depends on the output of “abs”, which # depends on the sign value of “w\_I"

Update rule for w\_i:

w\_i = w\_i – alpha \* dLoss’ / dw\_i # definition of update rule

(substitute): = w\_i – alpha \* (dLoss / dw\_i + lambda\*sign(w\_i))

# Which is the update rule for L1 Regularization

========

3b) If we add l2 regularisation to our loss function (assuming error func = loss func) then we add lambda \* the squared weights to our loss and then use this new loss for grad desc.

Difference is l1 regularisation encourages feature sparsity (due to adding abs value of all weights) whereas l2 regularisation encourages more features but with smaller weights (due to penalising larger weights more by squaring them).

3c) Shuffle dataset. Split the dataset into 60:20:20 (training:validation:test) and train classifier on training. Test performance on validation and optimise using various hyperparameters (lambda for regularisation, batch size if using mini-batch grad desc during training, learning rate, num iterations) on the validation set (maybe running multiple times). Evaluate once on the final test set (not used for testing) to show an approximation of how the classifier generalises to test data.

3d) Accuracy = 1000/1200 = ⅚. Not necessarily a good metric as the test data is heavily biased towards class 1, so accuracy will be misleading as the classifier will classify many values as class 1 as a result (and classes 2, 3 have no correctly predicted values so their precisions and recall equal 0 and their f1 score is undefined). Could use macro-averaged recall to cover all classes and get a more accurate result.

(Could argue use of ROC-AUC, but I don’t know how to)

[UNSURE]

Can try resolve the imbalanced data distribution by “downsampling” the majority class, “upsampling” the minority class, or dividing each entry in the confusion matrix by its row sum (the total number of samples in its class). Wouldn’t be real since it’s effectively making up data, but better than making deductions on an imbalanced dataset.