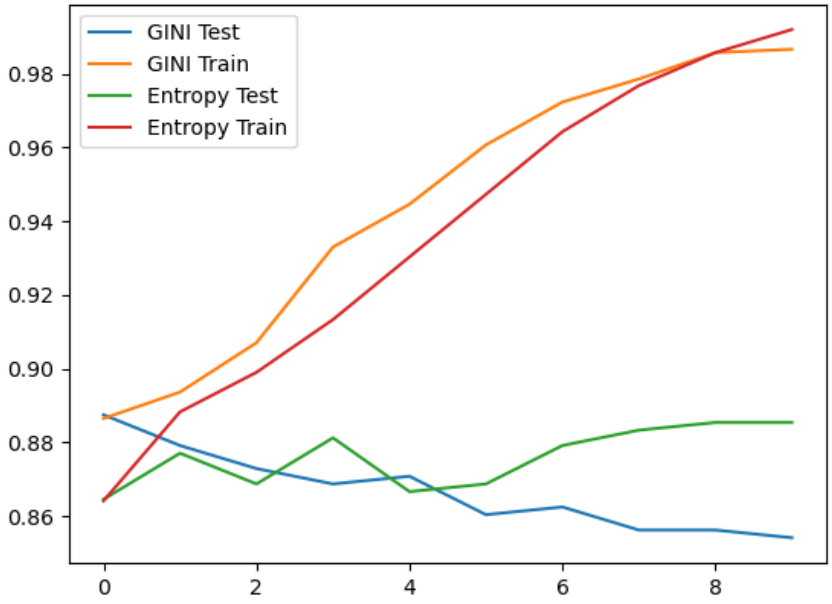
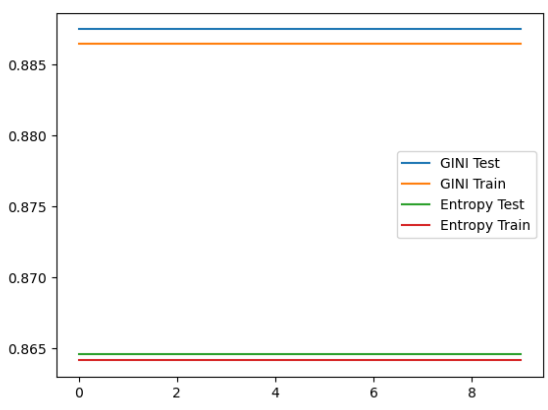
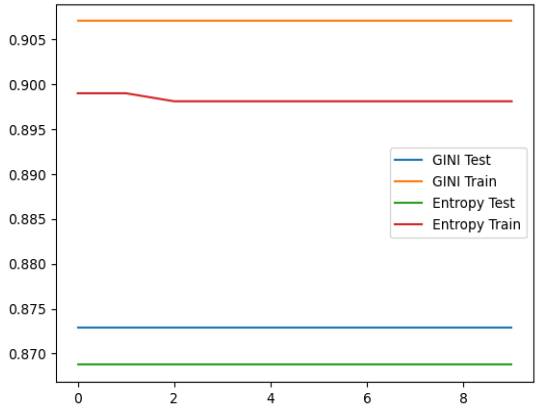
CS 334 Machine Learning HW #2

1. Decision Tree Implementation

d. Figure1: y→accuracy, x→maxDepth {minimum leaf sample = 2 for all}







For the two graphs above: y→ accuracy, x→MLS

Left {maxDepth = 2 for all} Right {maxDepth = 4 for all}

e. The time complexity for train and predict of my decision tree classifier in terms of n (training size), d (number of features), and p (depth) is O($n^2 d$) and O(np) respectively.

Time Complexity Calculation for my code:

def _calculate_split_score(size):

Everytime this function is called, it takes O(size) since a summation is taken on each value.

def_best_split(size):

For each feature (d), and for each data (size), we are calling calculate_split twice: O(2*d*size)*calculate_split.

def_decision_tree(size):

This function calls best_split each time it is called once and it calls itself 2 times for every split (max split depends on number of feature): best_split(size) + decision_tree(left) + decision_tree(right)

= best_split(size) + best_split(left) + decision_tree(left:left) + decision_tree(right:right) ... and so on

def train: simply calls decision_tree once on the entire dataset:

O(1)* decision_tree

In Total:

decision_tree's calls to best_split are decreasing in size starting with n, into best_split(n-a(n)) and best_split(n -(1-a)n) for a maximum of max depth times (p). **where a is a proportion of the data splitted**.

Assuming for worst split (where a = 0.5) and maxDepth (p), the cost of our decision tree on each split is as follows:

best_split(n) + 2*best_split(0.5n) + 4*best_split(0.25n) ... until maxDepth(p)

As we saw earlier, in general, best_split(x) = O(x*d*calculate_split), where calculate_split = x. Hence best_split(x) = O(x*x*d)

Plugging this back into the best split equation, we get:

(n*n*d) + 2*(½)^2*n*n*d) + 4*(¼)^2*n*n*d) … + 2^p*(½^p)^2*n*n*d)

⇒ (n*n*d) (1+½ +¼+⅛ + … + ½^p)

⇒ (n^2*d) * 2 * (1 - (1/2)^p) is the theoretical worst time complexity

Since (1 - (1/2)^p)*2 is a relatively small constant, the final time complexity is simply:

$O(n^2 d)$

The time complexity for traversing a tree recursively is O(p). Generally, p = n can happen, but since we set a maxDepth for our classifier, the entire predict function has a time complexity of O(p) for each prediction ⇒ O(np) for predicting all the test data.

2. **Exploring Model Assessment Strategies**

d.

	Strategy	TrainAUC	ValAUC	Time
0	Holdout	0.954831	0.784473	0.011061
1	2-fold	0.951588	0.775471	0.020458
2	5-fold	0.955085	0.788272	0.057291
3	10-fold	0.954634	0.794211	0.118528
4	MCCV w/ 5	0.953775	0.784141	0.055041
5	MCCV w/ 10	0.953556	0.785452	0.110470
6	True Test	0.953284	0.824975	0.000000

Each iteration of q2.py resulted in different values for each model, so an average was taken on 10 iterations *refer to the code for reference*. While the values are very similar, 10-fold cross validation resulted in an AUC that most closely resemble the True Test for the test AUC value. In terms of

computational time, holdout (unsurprisingly, since it's the most simple model) took the least amount of time while the iterative approaches of k-fold and monte carlo took more time.

3. **Robustness of Decision Trees and K-NN**

a. Using gridsearchCV {with CV values 2, 5, 10, 15, 20}, with various common values of mls, md, criterion, k, and p. I took the most frequent results from 10 iterations *refer to code in q3.py for reference*.

minimum leaf sample: 18

max depth: 10

criterion: entropy

k: 28

p: 1

b. Below are the accuracy scores and AUC values for each of the different KNN models: In general (running multiple instances of the code), accuracy scores and AUC scores seem to decrease with less data.

Accuracy Scores:

Total: 0.8604166666666667

-5%: 0.8708240534521158

-10%: 0.8669724770642202

-20%: 0.8597883597883598

AUC Scores:

Total: 0.4975903614457831

-5%: 0.4974554707379134 6

-10%: 0.49736842105263157

-20%: 0.5

c. Below are the accuracy scores and AUC values for each of the different DTC models: Like KNN, in general (running multiple instances of the code), accuracy scores and AUC scores seem to decrease with less data.

Accuracy Scores:

Total: 0.9

-5%: 0.8859649122807017

-10%: 0.8741258741258742

-20%: 0.8846153846153846

AUC Scores:

Total: 0.7215940685820204

-5%: 0.7005937234944868

-10%: 0.6674448217317487

-20%: 0.6608823529411764

d. Looking at these scores below, it seems like the hyper parameters that I selected generated similar levels of accuracy. However, there is a significant difference in AUC values for the two different models. While the DTC AUC values were within the expected range, I could not identify why my KNN AUC values are suggesting my model is no better than random guessing - especially with an average accuracy of ~86%.

	Model	Val AUC	Accuracy
0	**Knn All Data:**	**0.497590**	**0.860417**
1	**Knn - 5%:**	**0.497455**	**0.870824**
2	**Knn - 10%:**	**0.497368**	**0.866972**
3	**Knn - 20%:**	**0.500000**	**0.859788**
4	**DTC All Data:**	**0.754773**	**0.912500**

5 DTC - 5%: 0.700594 0.885965

6 DTC - 10%: 0.667445 0.874126

7 DTC - 20%: 0.660882 0.884615