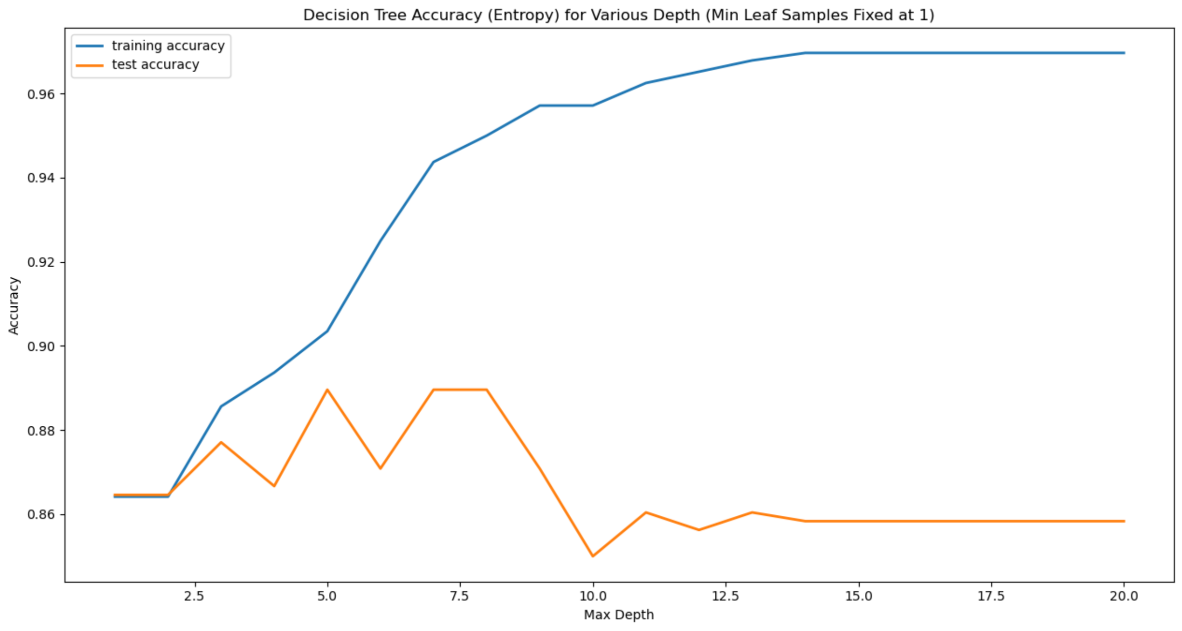
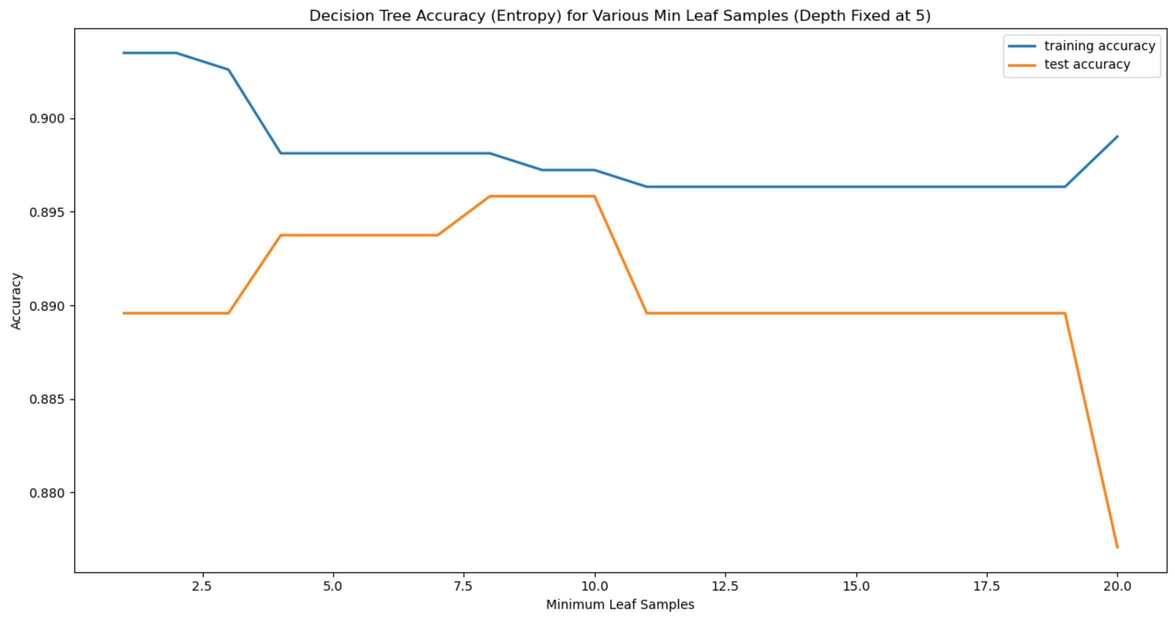
/* THIS CODE IS MY OWN WORK, IT WAS WRITTEN WITHOUT CONSULTING CODE WRITTEN BY OTHER STUDENTS OR LARGE LANGUAGE MODELS SUCH AS CHATGPT. Tommy Skodje */

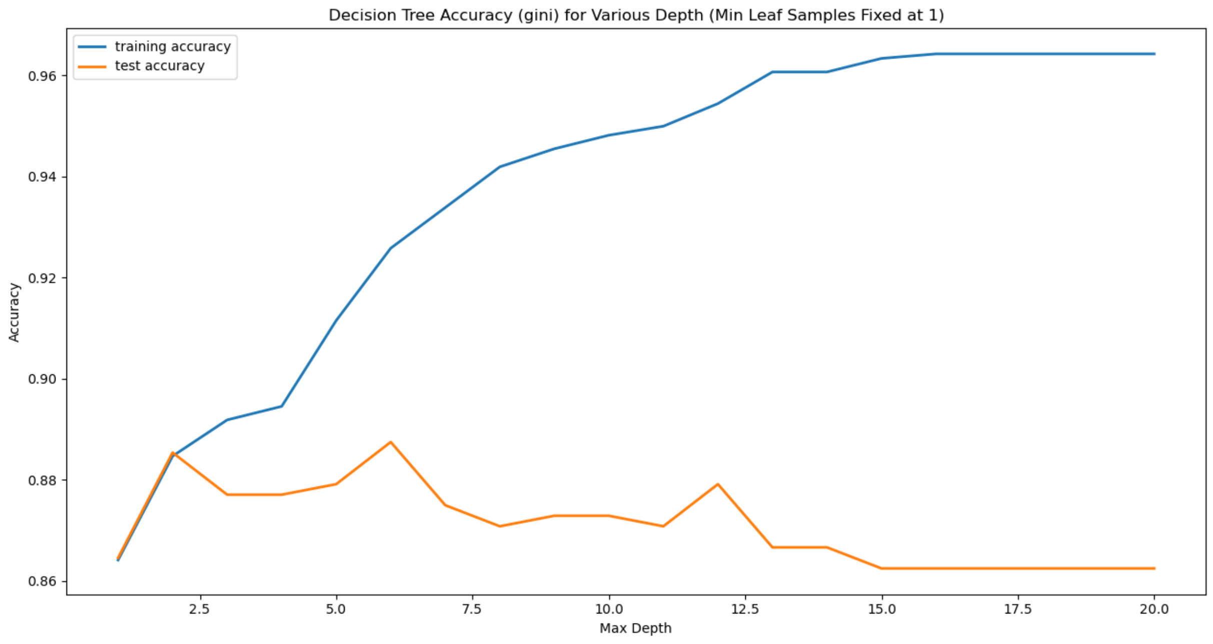
I collaborated with the following classmates for this homework:
None

1. d.

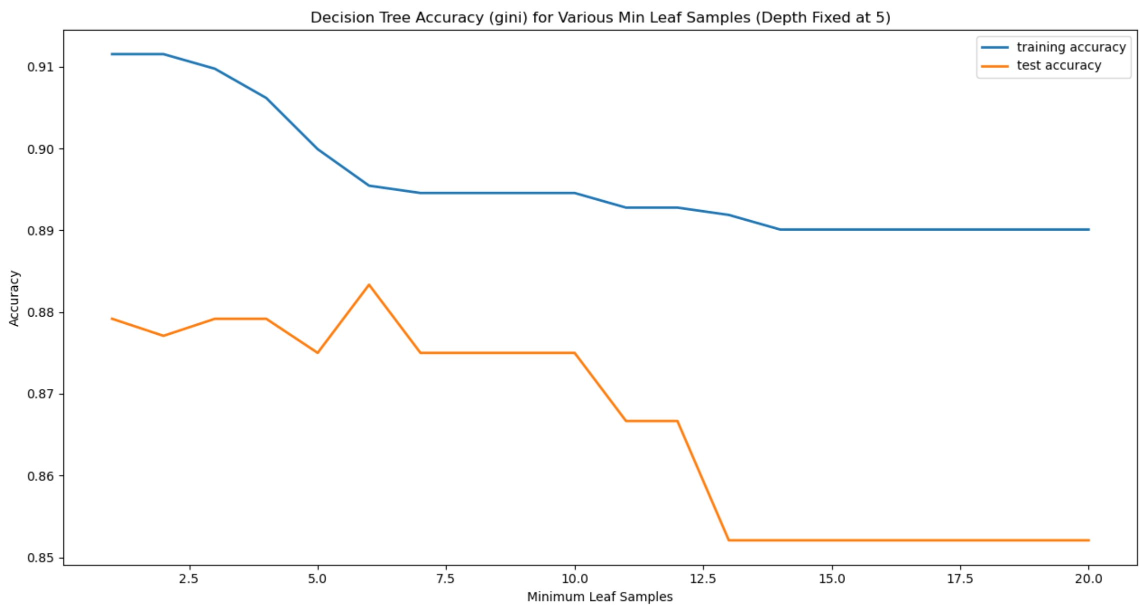




Decision Tree Accuracy (Entropy) for Various Min Leaf Samples (Depth Fixed at 5)
training accuracy
test accuracy
Accuracy
Minimum Leaf Samples
0.900
0.895
0.890
0.885
0.880
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0



Decision Tree Accuracy (gini) for Various Depth (Min Leaf Samples Fixed at 1)
training accuracy
test accuracy
Accuracy
Max Depth
0.96
0.94
0.92
0.90
0.88
0.86
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0



As the graphs above show, training accuracy increases as depth increases, while test accuracy increases up until about depth = 8 for entropy and depth = 12 for gini. Afterwards, test accuracy decreases. In contrast, as min leaf samples increase, training accuracy generally decreases. However, test accuracy increases until about min leaf samples = 10 for entropy and min leaf samples = 6 for gini, then decreases afterward.

1. e. I will first calculate the computational complexity of the train() function. train() calls CreateTree() unless depth = 0, the root node is already pure, or the root node cannot be split without going under the minimum leaf samples.

```
# Call a recursive method to build the tree
self.CreateTree(attributes=xFeat, target=y, depth=0,
root=self)
```

If the decision tree stops without splitting a single time, the algorithm takes O(n) time since the majority class needs to be calculated by looping through the n entries in the dataset.

```
if entropy == 0.0 or self.maxDepth == 0 or rows <=
self.minLeafSample\
or (rows - self.minLeafSample) < self.minLeafSample:

    counts = {}
    for value in y:
        if value not in counts:
            counts[value] = 1
        else:
            counts[value] = counts.get(value) + 1

    majority_class = max(counts, key=counts.get)

    self.decision_tree = {"lchild": None, "rchild": None,
    "svar": None, "sval": None, "class": majority_class}

    return self
```

If the decision tree continues past the root node, there are no loops or recursion within train(), and operations within train() take constant time no matter the values of n, d, or p. train() then calls the recursive method CreateTree().

The recursive case of CreateTree() calculates the entropy or gini index for the current dataset. Before gini or entropy can be calculated, the current row needs to be sorted using the numpy argsort() function. This function uses the quicksort algorithm, and therefore takes O(n log n) time.

```
# Sort in ascending order by the left column
cols_argsort = col_target[:, 0].argsort()
col_target = col_target[cols_argsort]
```

After the column is sorted, the entropy or gini index is calculated within the calculate_split_score() method. Both gini and entropy require 2 non-nested for loops that loop through all the columns in the dataset, which is equivalent to n, the training size. All of the operations within the for loop take constant time, so calculating the split score takes O(n) time.

```python
if criterion == "entropy":
     entropies = []
     # Calculate p(X=k)log2p(X=k) for each category
     for category in counts.keys():
          current_count = counts.get(category)
          prob = current_count / rows
          current_entropy = prob * np.log2(prob)
          entropies.append(current_entropy)

     # Sum the entropies and return
     score = 0
     for entropy in entropies:
          score += entropy

     score = score * -1
```

```python
elif criterion == "gini":
     ginis = []
     # Calculate p(X=k)*(1-p(X=k)) for each category
     for category in counts.keys():
          current_count = counts.get(category)
          prob = current_count / rows
          current_gini = prob * (1 - prob)
          ginis.append(current_gini)

     # Calculate and return the total gini index
     score = 0
     for gini in ginis:
          score += gini
```

Returning back to the CreateTree() method, the entropy/gini index needs to be calculated for each of the d features, which is the outer for loop in this function. This for loop runs d times. Within this for loop, the entropy/gini needs to be calculated for all points within the range of the leftmost split up to the rightmost split. Both the leftmost split and rightmost split depend on how many values there are within the dataset. In addition, the number of possible splitting points scales linearly with respect to the number of rows in the dataset, so this inner for loop runs up to n times. Even if it doesn't run n times, the number of times the for loop runs is linearly proportional to n.

NOTE: Some operations that have already been discussed, as well as operations that take constant time have been removed from the code snippets below for brevity.

```
for col in attributes.T:
     col_target = np.stack((col, target), axis=1)
for split in range(leftmost_split, rightmost_split + 1):
     left_samples = col_target[:split, 0]
     right_samples = col_target[split:, 0]
     left_targets = col_target[:split, 1]
     right_targets = col_target[split:, 1]
```

Up to this point, the algorithm takes n logn + n + nd time. n grows at a slower rate than n log n and nd as long as d and n are greater than 1. Therefore the time complexity up to this point is O(n logn + nd).

However, there is the final step of this algorithm, which is to recurse on the left and right child nodes of the decision tree. The algorithm creates children until the maximum depth is reached, and each time a child node is created, the algorithm takes O(n logn + nd) time to run. Since the decision tree is a binary tree, it has a worst-case height of n (which is possible when the minimum leaf nodes are set to 1 and the maximum depth is set to n or higher). Assuming n child nodes are created, the algorithm would take O(n(n logn + nd)) time. Simplifying this, we get O(n^2 logn + n^2d) time taken for the train function.

Moving on to the predict function, the function starts by looping through each row in xFeat, starting at the root node.

```
for row in xFeat:
     current_node = self
     # Method continues below this
```

Afterwards, there is a while loop that continues to run while the current decision tree node is not a leaf node.

```
while ((current_node.decision_tree.get("lchild") is not
None) and
(current_node.decision_tree.get("rchild") is not None)):
     current_node_var =
     current_node.decision_tree.get("svar")
```

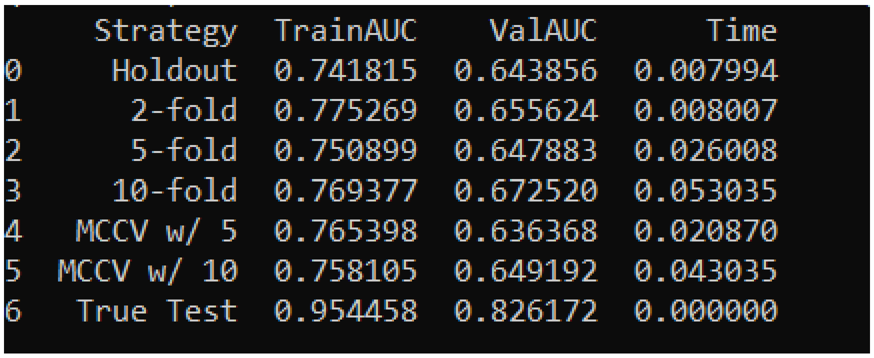
```
    current_node_val =
    current_node.decision_tree.get("sval")

    # Go left
    if row[current_node_var] < current_node_val:
        current_node =
        current_node.decision_tree.get("lchild")
    # Go right
    elif row[current_node_var] >= current_node_val:
        current_node =
        current_node.decision_tree.get("rchild")
    else:
        break
```

After this, the predicted label is appended to yHat. After all predicted labels have been appended to yHat, yHat is returned.

The for loop runs for all rows in xFeat, so it runs in O(n) time. As discussed under the time complexity discussion for the train() method, the worst case height for a binary tree is n, which could occur if min leaf samples is set to 1, and max depth is set to n or higher. In this case, the while loop would run n times. The operations within the while loop take constant time.

Therefore, in the worst case where the entire decision tree of depth n is traversed for each row of the dataset, predict() takes O(n * n) = O(n^2) time.

2. d.



```
     Strategy  TrainAUC    ValAUC      Time
0     Holdout  0.741815  0.643856  0.007994
1      2-fold  0.775269  0.655624  0.008007
2      5-fold  0.750899  0.647883  0.026008
3     10-fold  0.769377  0.672520  0.053035
4   MCCV w/ 5  0.765398  0.636368  0.020870
5  MCCV w/ 10  0.758105  0.649192  0.043035
6   True Test  0.954458  0.826172  0.000000
```

The above image is a screenshot of the console output of the code from parts 2a, 2b, and 2c. Both the trainingAUC and valAUC were lower for all methods tested than for the true test performed at the end.

The AUC for both the training and test data varies significantly between techniques. Holdout (with a train-test split of 70:30) has an AUC of about 0.7418 on training data, which is the lowest for all strategies, and 0.6439 on unseen data, which is the second lowest value for all the strategies.

K-fold cross-validation fares significantly better than holdout in regards to AUC, with the area under the curve increasing for both training and test data. Strangely, both AUC values drop when k is increased from 2 to 5. However, 10-fold outperforms both 2-fold and 10-fold, with the highest test AUC value for all strategies, and the second highest training AUC value.

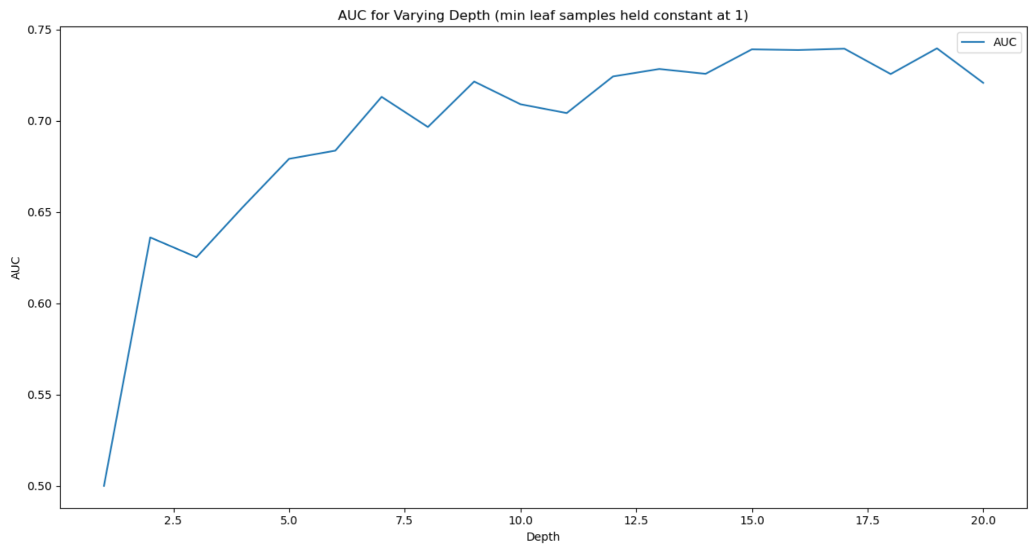
Monte-Carlo cross-validation provided a lower test AUC for both 5 and 10 samples compared to 5-fold and 10-fold validation. However, the training AUC values are very comparable to k-fold cross-validation.

Out of these three methods, it appears that 10-fold cross-validation provided the most robust estimate for unseen data, with holdout being the least robust. This is because 10-fold cross-validation creates many “splits” of the data, so adjusting hyperparameters and kinds of data is less likely to affect the outcome. Holdout is the least robust because it only takes 1 split of the data.

As for the computational time taken for each method, 2-fold and holdout are very similar in time taken, with 2-fold narrowly having an advantage. 5-sample MCCV and 5-fold also have very similar times, with 5-sample MCCV having the narrow advantage. Both 5-sample MCCV and 5-fold take more time than 2-fold and holdout. 10-sample MCCV takes about twice as long as the two previous techniques, with 10-fold taking the longest out of the strategies tested at about 0.0530 seconds.

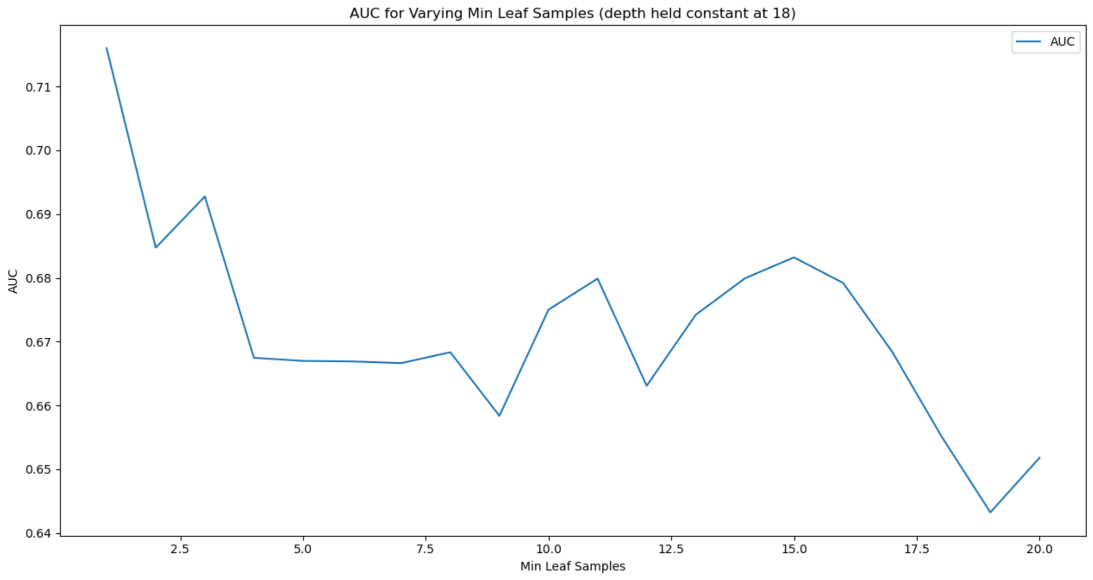
3. a. I will be using k=10 for my k-fold cross-validation. The reason I am doing this is because decision tree performed the best for all techniques tested when k=10. In addition, the AUC increase between 5-fold and 10-fold cross-validation was about 0.03. It appears that there are diminishing returns for increasing k, and the increase from k=5 to k=10 was small. Therefore, I do not feel that increasing k beyond 10 would be worth the extra time taken to perform the cross-validation.

   Here is the graph for AUC as a function of max depth, holding min leaf samples constant at 1.



   A max depth of around 18 seems to provide the highest AUC.

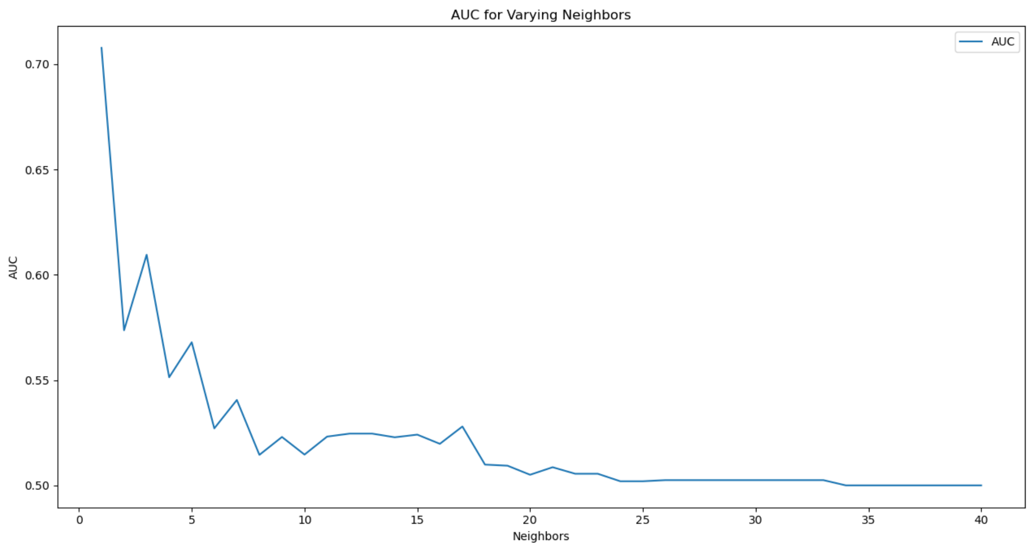
Here is the graph for AUC as a function of min leaf samples, holding max depth constant at 18 (since a max depth of 18 seemed to provide the highest AUC value).



It seems like a min leaf sample value of 1 provided the highest AUC.

Therefore, for decision tree the optimal parameters are
$\theta_{opt} = maximum\ depth:\ 18,\ minimum\ leaf\ samples:\ 1$

Next, I tuned the hyperparameter for K-Nearest Neighbors. The only hyperparameter to tune was the number of neighbors.



1 neighbor provicded the highest AUC value.

Therefore, for K-Nearest Neighbors, the optimal hyperparameter is:
$\theta_{opt} = n\, neighbors\colon\ 1$

3. b. After training K-NN on the entire training dataset with the optimal hyperparameter, the AUC and accuracy were
AUC: 0.7234476367006488
accuracy: 0.8583333333333333

After randomly removing 5% of the original training data, the AUC and accuracy were
5% removed testAuc: 0.7217794253938832
5% removed accuracy: 0.8666666666666667

After randomly removing 10% of the original training data, the AUC and accuracy were
10% removed testAuc: 0.7133456904541242
10% removed accuracy: 0.8520833333333333

After randomly removing 20% of the original training data, the AUC and accuracy were
20% removed testAuc: 0.6825764596848934
20% removed accuracy: 0.84375

3. c. After training a decision tree on the entire training dataset with the optimal hyperparameters, the AUC and accuracy were
testAuc: 0.7371640407784986
accuracy: 0.8708333333333333

After randomly removing 5% of the original training data, the AUC and accuracy were
5% removed testAuc: 0.7578313253012049
5% removed accuracy: 0.8729166666666667

After randomly removing 10% of the original training data, the AUC and accuracy were
10% removed testAuc: 0.772752548656163
10% removed accuracy: 0.8875

After randomly removing 20% of the original training data, the AUC and accuracy were
20% removed testAuc: 0.7489341983317888
20% removed accuracy: 0.86875

3. d.

	K-Nearest Neighbors AUC and Accuracy	Decision Tree AUC and Accuracy
Full Training Dataset	AUC: 0.7234476367006488 accuracy: 0.8583333333333333	testAuc: 0.7371640407784986 accuracy: 0.8708333333333333
5% Removed	AUC: 0.7217794253938832 accuracy: 0.8666666666666667	testAuc: 0.7578313253012049 accuracy: 0.8729166666666667
10% Removed	AUC: 0.7133456904541242 Accuracy: 0.8520833333333333	testAuc: 0.772752548656163 accuracy: 0.8875
20% Removed	AUC: 0.6825764596848934 accuracy: 0.84375	testAuc: 0.7489341983317888 accuracy: 0.86875

The k-nearest neighbors algorithm did not lose much AUC when 5% of the training data was removed or when 10% of the training data was removed, but lost a significant amount when 20% of the training data was removed. The accuracy remained fairly steady, but there was about a 1% decrease in accuracy when 10% of the training data was removed, and another drop of about 1% when 20% was removed. We can conclude that the k-nearest neighbors algorithm was not too sensitive to small portions of the data being removed, but produced significantly worse results when 20% was removed.

For decision tree, the testAuc actually increased when 5%, 10%, and 20% of the training data was removed. This could be the artifact of only doing a single trial. The accuracy remained fairly constant, with a slight drop-off accuracy coming when 20% of the training data was removed. We can conclude that the decision tree algorithm with optimal hyperparameters was not very sensitive to 5%, 10%, and 20% of the data being removed.