HOMEWORK 2

Hafeez Ali Anees Ali
908 460 3423

Theta: 3

1. *Questions*

Q1. (Our algorithm stops at pure labels) [5 pts] If a node is not empty but contains training items with the same label, why is it guaranteed to become a leaf? Explain. You may assume that the feature values of these items are not all the same.

A: If all training items have the same class label, then the probability of predicting the class label for any item in the training dataset is always 1, which implies that the entropy of any candidate split for any feature is always equal to 0. Hence, according to the stopping criteria for a decision tree construction, the node must be converted to a leaf.

Conversely, if we force a split on any feature and continue further, all subsequent leafs in the sub-tree will always have the same class label. Hence, we can prune the subtree and make the parent node a leaf with the same class label.

Q2. (Our algorithm is greedy) [5 pts] Handcraft a small training set where both classes are present but the algorithm refuses to split; instead it makes the root a leaf and stops. Importantly, if we were to manually force a split, the algorithm will happily continue splitting the data set further and produce a deeper tree with zero training error. You should (1) plot your training set, (2) explain why. (Hint: you don't need more than a handful of items.)

A: Consider the following dataset containing 2 features X1, X2 and the label Y (XOR function) :

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

Here, for both features X1 and X2, any candidate split (0 or 1) has split entropy of 0. Therefore, the decision tree construction ceases at the root node itself converting the root to a leaf.

If we force a split on any of the features the algorithm will continue to construct a deeper tree with 3 levels with a training accuracy of 100%. It will have 4 boolean expressions for each possible combination of X1 and X2. This can be seen in the images below:

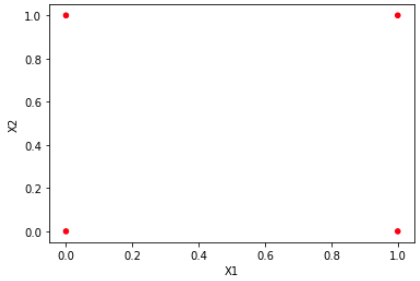
Plot:

The screenshot of the code shown below has the dataset plotted on an X1-X2 plane and also shown in the table.

```
In [37]: #Q2

In [38]: data = [[1,0,1],[0,1,1],[1,1,0],[0,0,0]]

In [39]: df = pd.DataFrame(data, columns=['X1','X2','Y'])

In [40]: df

Out[40]:
```

	X1	X2	Y
0	1	0	1
1	0	1	1
2	1	1	0
3	0	0	0

```
In [41]: q2_d3 = generate_d3(df, 'root')

Searching best split candidate for: X1
Best X1_split: 1, X1_entropy: 1.0, X1_gain_ratio: 0.0
Searching best split candidate for: X2
Best X2_split: 0, X2_entropy: 1.0, X2_gain_ratio: 0
X1 and X2 gain ratio is 0. Finding majority class and stopping

/var/folders/v0/g401c4x979d___jdl5ddw2_r0000gn/T/ipykernel_46588/1135821344.py:4: UserWarning: Boolean Series key wil
l be reindexed to match DataFrame index.
  else_positive_count = len(else_dataset[dataset[CLASS_LABEL_COLUMN_NAME] == 1])
/var/folders/v0/g401c4x979d___jdl5ddw2_r0000gn/T/ipykernel_46588/1135821344.py:6: UserWarning: Boolean Series key wil
l be reindexed to match DataFrame index.
  then_positive_count = len(then_dataset[dataset[CLASS_LABEL_COLUMN_NAME] == 1])

In [42]: printD3(q2_d3)

[Id: 62, label: 1, parent: root]

In [43]: decision_boundary(df, q2_d3)
```



Why:

Let’s say we force a split on X2 >= 1. Further dataset splits are shown below:

Split 1:

X1 X2 Y
0 0 0
1 0 1

Split 2:
X1 X2 Y
0 1 1
1 1 0

Now, for ‘Split 1’, we have Y = X1 and for ‘Split 2’, we have Y = !X1, which can be used to split the tree further to form a complete binary tree with boolean expressions representing an XOR function.

Q3: (Gain ratio exercise) [10 pts] Use the training set Druns.txt. For the root node, list all candidate cuts and their information gain ratio. If the entropy of the candidate split is zero, please list its mutual information (i.e., information gain). (Hint: to get log2(x) when your programming language may be using a different base, use log(x)/ log(2). Also, please follow the split rule in the first section.)

A: Shown below are screenshots of the code used to calculate entropy, best split and gain ratio. The second screenshot shows the gain ratio and information gain for each split in X1 and X2 in Druns dataset.

Feature X1:
Split: 0.1, Gain Ratio: 0.10051807676021852, Information Gain: 0.04417739186726144
Split: 0.0, Gain Ratio: 0, Information Gain: 0.0

Feature X2:
Split: -2, Gain Ratio: 0, Information Gain: 0.0
Split: -1, Gain Ratio: 0.10051807676021852, Information Gain: 0.04417739186726144
Split: 0, Gain Ratio: 0.055953759631263686, Information Gain: 0.03827452220629257
Split: 1, Gain Ratio: 0.005780042205152451, Information Gain: 0.004886164091842837
Split: 2, Gain Ratio: 0.0011443495172768668, Information Gain: 0.0010821659130776373
Split: 3, Gain Ratio: 0.016411136842102245, Information Gain: 0.016313165825732168
Split: 4, Gain Ratio: 0.04974906418177866, Information Gain: 0.04945207278939412
Split: 5, Gain Ratio: 0.1112402958633981, Information Gain: 0.1051955320700464
Split: 6, Gain Ratio: 0.2360996061436081, Information Gain: 0.19958702318968746
Split: 7, Gain Ratio: 0.055953759631263686, Information Gain: 0.03827452220629257
Split: 8, Gain Ratio: 0.43015691613098095, Information Gain: 0.18905266854301628

```
In [193]: # Tries all possible numbers in feature as potential split values
def determine_best_split_entropy_and_gain_ratio(dataset, feature_column, debug = False):
    print("Searching best split candidate for: " + str(feature_column))
    best_split = None
    max_entropy = None
    max_gain_ratio = None
    class_entropy = calculate_class_entropy(dataset)
    for split in dataset[feature_column]:
        conditional_entropy, feature_entropy = calculate_conditional_and_feature_entropy(dataset, feature_column, split
        gain = class_entropy - conditional_entropy
        if feature_entropy == 0:
            gain_ratio = 0
        else:
            gain_ratio = gain/feature_entropy
        if best_split is None or gain_ratio > max_gain_ratio:
            best_split = split
            max_gain_ratio = gain_ratio
            max_entropy = conditional_entropy
        if debug == True:
            print("Split: " + str(split) + ", Gain Ratio: " + str(gain_ratio) + ", Information Gain: " + str(conditiona
    return best_split, max_entropy, max_gain_ratio
```

```
In [229]: determine_best_split_conditional_entropy_and_gain_ratio(Druns_dataset, FEATURE_1, True)

Searching best split candidate for: X1
Split: 0.1, Gain Ratio: 0.10051807676021852, Information Gain: 0.04417739186726144
Split: 0.0, Gain Ratio: 0, Information Gain: 0.0
Split: 0.0, Gain Ratio: 0, Information Gain: 0.0
Split: 0.0, Gain Ratio: 0, Information Gain: 0.0
Split: 0.0, Gain Ratio: 0, Information Gain: 0.0
Split: 0.0, Gain Ratio: 0, Information Gain: 0.0
Split: 0.0, Gain Ratio: 0, Information Gain: 0.0
Split: 0.0, Gain Ratio: 0, Information Gain: 0.0
Split: 0.0, Gain Ratio: 0, Information Gain: 0.0
Split: 0.0, Gain Ratio: 0, Information Gain: 0.0
Split: 0.0, Gain Ratio: 0, Information Gain: 0.0

/var/folders/v0/g401c4x979d___jdl5ddw2_r0000gn/T/ipykernel_36454/1135821344.py:4: UserWarning: Boolean Series key wil
l be reindexed to match DataFrame index.
  else_positive_count = len(else_dataset[dataset[CLASS_LABEL_COLUMN_NAME] == 1])
/var/folders/v0/g401c4x979d___jdl5ddw2_r0000gn/T/ipykernel_36454/1135821344.py:6: UserWarning: Boolean Series key wil
l be reindexed to match DataFrame index.
  then_positive_count = len(then_dataset[dataset[CLASS_LABEL_COLUMN_NAME] == 1])
```

```
In [230]: determine_best_split_conditional_entropy_and_gain_ratio(Druns_dataset, FEATURE_2, True)

Searching best split candidate for: X2
Split: -2, Gain Ratio: 0, Information Gain: 0.0
Split: -1, Gain Ratio: 0.10051807676021852, Information Gain: 0.04417739186726144
Split: 0, Gain Ratio: 0.055953759631263686, Information Gain: 0.03827452220629257
Split: 1, Gain Ratio: 0.005780042205152451, Information Gain: 0.004886164091842837
Split: 2, Gain Ratio: 0.0011443495172768668, Information Gain: 0.0010821659130776373
Split: 3, Gain Ratio: 0.016411136842102245, Information Gain: 0.016313165825732168
Split: 4, Gain Ratio: 0.04974906418177866, Information Gain: 0.04945207278939412
Split: 5, Gain Ratio: 0.1112402958633981, Information Gain: 0.1051955320700464
Split: 6, Gain Ratio: 0.2360996061436081, Information Gain: 0.19958702318968746
Split: 7, Gain Ratio: 0.055953759631263686, Information Gain: 0.03827452220629257
Split: 8, Gain Ratio: 0.43015691613098095, Information Gain: 0.18905266854301628

/var/folders/v0/g401c4x979d___jdl5ddw2_r0000gn/T/ipykernel_36454/1135821344.py:4: UserWarning: Boolean Series key wil
l be reindexed to match DataFrame index.
  else_positive_count = len(else_dataset[dataset[CLASS_LABEL_COLUMN_NAME] == 1])
/var/folders/v0/g401c4x979d___jdl5ddw2_r0000gn/T/ipykernel_36454/1135821344.py:6: UserWarning: Boolean Series key wil
l be reindexed to match DataFrame index.
  then_positive_count = len(then_dataset[dataset[CLASS_LABEL_COLUMN_NAME] == 1])
```

Q4: (The king of interpretability) [10 pts] Decision trees are not the most accurate classifiers in general. However, they are useful, largely due to their interpretability: a data scientist can easily explain a tree to a non-data scientist. Build a tree from D3leaves.txt. Then manually convert your tree to a set of logic rules. Show the tree and the rules.

A: The below screenshot shows the decision tree constructed on D3leaves.txt. The tree is depicted in level order traversal format starting from level 0 till the deepest level. Each level has nodes, where each node has the following attributes:

Id: ‘Unique identifier’ for every node
Predicate: ‘feature_name >= split_value’ [the feature chosen to split the dataset at the node and the value on which the dataset is being split]
Parent: ‘root’ if it is root, else ‘parent_id’ followed by ‘_left’ or ‘_right’ indicating whether the node is the left or right child of the parent found using ‘parent_id’
Label: 1 or 0, if node is leaf

To obtain the logic rules for each leaf node, a depth first traversal starting from root is done storing all predicates. If the right branch is taken a ! (not) is added to the predicate indicating the else branch.

The boolean expressions are:

X2 >= 2 -> 1
X2 < 2 and X1 >= 10 -> 1
X2 < 2 and X1 < 10 -> 0
The tree levels are:

Level: 0
[Id: 2, predicate: X2 >= 2, parent: root]
Level: 1
[Id: 3, label: 1, parent: 2_left] [Id: 4, predicate: X1 >= 10, parent: 2_right]
Level: 2
[Id: 5, label: 1, parent: 4_left] [Id: 6, label: 0, parent: 4_right]

```
In [350]: printD3(q4_d3)

          Level: 0
          [Id: 2, predicate: X2 >= 2, parent: root]
          Level: 1
          [Id: 3, label: 1, parent: 2_left] [Id: 4, predicate: X1 >= 10, parent: 2_right]
          Level: 2
          [Id: 5, label: 1, parent: 4_left] [Id: 6, label: 0, parent: 4_right]

In [245]: fetch_boolean_expressions(q4_d3, [])

          X2 >= 2 == 1
          !(X2 >= 2) and X1 >= 10 == 1
          !(X2 >= 2) and !(X1 >= 10) == 0
```

The code logic is shown in the below screenshots:

In [202]:
```
def print_boolean_expression(label, predicates):
    result = ""
    for pred in predicates:
        if len(result) == 0:
            result = result + pred
        else:
            result = result + " and " + pred
    result = result + " == " + str(label)
    print(result)
```

In [203]:
```
def fetch_boolean_expressions(node, boolean_stack):
    left_stack = boolean_stack.copy()
    right_stack = boolean_stack.copy()
    if node.leaf is False:
        if node.left is not None:
            left_stack.append(node.predicate)
            fetch_boolean_expressions(node.left, left_stack)
        if node.right is not None:
            right_stack.append("!(" + node.predicate + ")")
            fetch_boolean_expressions(node.right, right_stack)
    else:
        print_boolean_expression(node.label, boolean_stack)
```

In [345]:
```
# Tree is printed one level at a time.
def printD3(d3_node):
    level = []
    level_no = 0
    level.append(d3_node)
    print("Level: " + str(level_no))
    while(len(level) > 0):
        nextLevel = []
        while(len(level) > 0):
            node = level.pop(0)
            print_node(node)
            if node.left is not None:
                nextLevel.append(node.left)
            if node.right is not None:
                nextLevel.append(node.right)
        level = nextLevel
        level_no = level_no + 1
        print("")
        if len(level) > 0:
            print("Level: " + str(level_no))
```

Q5. (Or is it?) [20 pts] For this question only, make sure you DO NOT VISUALIZE the datasets or plot your tree's decision boundary in the 2D x space. If your code does that, turn it off before proceeding. This is because you want to see your own reaction when trying to interpret a tree. You will get points no matter what your interpretation is. And we will ask you to visualize them in the next question anyway.

• Build a decision tree on D1.txt. Show it to us in any format (e.g., could be a standard binary tree with nodes and arrows, and denote the rule at each leaf node; or as simple as plaintext output where each line represents a node with appropriate line number pointers to child nodes; whatever is convenient for you). Again, do not visualize the data set or the tree in the x input space. In real tasks, you will not be able to visualize the whole high dimensional input space anyway, so we don't want you to "cheat" here.
• Look at your tree in the above format (remember, you should not visualize the 2D dataset or your tree's decision boundary) and try to interpret the decision boundary in human understandable English.
• Build a decision tree on D2.txt. Show it to us.
• Try to interpret your D2 decision tree. Is it easy or possible to do so without visualization?

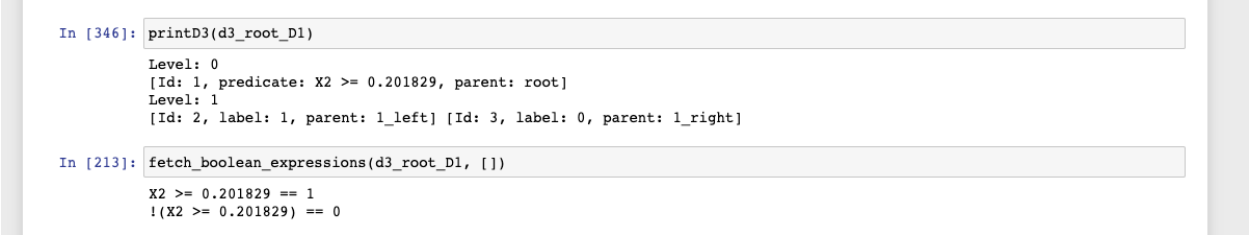
A: The level order traversal of the decision tree on D1.txt looks as shown below (screenshot attached as well):

Level: 0
[Id: 1, predicate: X2 >= 0.201829, parent: root]
Level: 1
[Id: 2, label: 1, parent: 1_left] [Id: 3, label: 0, parent: 1_right]

Interpretation:

If X2 >= 0.201829 for any data point, then the class label for such data points is 1, else it is 0. This is also represented by the logic rules of the leaves below:

X2 >= 0.201829 == 1
!(X2 >= 0.201829) == 0



The level order traversal of the decision tree on D2.txt is very large as shown below (screenshot attached as well):

Level: 0
[Id: 1, predicate: X1 >= 0.533076, parent: root]
Level: 1
[Id: 2, predicate: X2 >= 0.228007, parent: 1_left] [Id: 29, predicate: X2 >= 0.88635, parent: 1_right]
Level: 2
[Id: 3, predicate: X2 >= 0.424906, parent: 2_left] [Id: 16, predicate: X1 >= 0.887224, parent: 2_right] [Id: 30, predicate: X1 >= 0.041245, parent: 29_left] [Id: 37, predicate: X2 >= 0.691474, parent: 29_right]
Level: 3
[Id: 4, label: 1, parent: 3_left] [Id: 5, predicate: X1 >= 0.708127, parent: 3_right] [Id: 17, predicate: X2 >= 0.037708, parent: 16_left] [Id: 24, predicate: X1 >= 0.850316, parent: 16_right] [Id: 31, predicate: X1 >= 0.104043, parent: 30_left] [Id: 36, label: 0, parent: 30_right] [Id: 38, predicate: X1 >= 0.254049, parent: 37_left] [Id: 49, predicate: X2 >= 0.534979, parent: 37_right]
Level: 4
[Id: 6, label: 1, parent: 5_left] [Id: 7, predicate: X2 >= 0.32625, parent: 5_right] [Id: 18, predicate: X2 >= 0.082895, parent: 17_left] [Id: 23, label: 0, parent: 17_right] [Id: 25, predicate: X2 >= 0.169053, parent: 24_left] [Id: 28, label: 0, parent: 24_right] [Id: 32, label: 1, parent: 31_left] [Id: 33, predicate: X2 >= 0.964767, parent: 31_right] [Id: 39, label: 1, parent: 38_left] [Id: 40, predicate: X1 >= 0.191915, parent: 38_right] [Id: 50, predicate: X1 >= 0.426073, parent: 49_left] [Id: 61, label: 0, parent: 49_right]
Level: 5

[Id: 8, predicate: X1 >= 0.595471, parent: 7_left] [Id: 15, label: 0, parent: 7_right] [Id: 19, label: 1, parent: 18_left] [Id: 20, predicate: X1 >= 0.960783, parent: 18_right] [Id: 26, label: 1, parent: 25_left] [Id: 27, label: 0, parent: 25_right] [Id: 34, label: 1, parent: 33_left] [Id: 35, label: 0, parent: 33_right] [Id: 41, predicate: X2 >= 0.792752, parent: 40_left] [Id: 44, predicate: X2 >= 0.864128, parent: 40_right] [Id: 51, label: 1, parent: 50_left] [Id: 52, predicate: X1 >= 0.409972, parent: 50_right]
Level: 6
[Id: 9, predicate: X1 >= 0.646007, parent: 8_left] [Id: 14, label: 0, parent: 8_right] [Id: 21, label: 1, parent: 20_left] [Id: 22, label: 0, parent: 20_right] [Id: 42, label: 1, parent: 41_left] [Id: 43, label: 0, parent: 41_right] [Id: 45, predicate: X2 >= 0.865999, parent: 44_left] [Id: 48, label: 0, parent: 44_right] [Id: 53, predicate: X2 >= 0.597713, parent: 52_left] [Id: 56, predicate: X1 >= 0.393227, parent: 52_right]
Level: 7
[Id: 10, label: 1, parent: 9_left] [Id: 11, predicate: X2 >= 0.403494, parent: 9_right] [Id: 46, label: 0, parent: 45_left] [Id: 47, label: 1, parent: 45_right] [Id: 54, label: 1, parent: 53_left] [Id: 55, label: 0, parent: 53_right] [Id: 57, predicate: X2 >= 0.639018, parent: 56_left] [Id: 60, label: 0, parent: 56_right]
Level: 8
[Id: 12, label: 1, parent: 11_left] [Id: 13, label: 0, parent: 11_right] [Id: 58, label: 1, parent: 57_left] [Id: 59, label: 0, parent: 57_right]

The boolean logic rules for this decision tree are as follows:

X1 >= 0.533076 and X2 >= 0.228007 and X2 >= 0.424906 == 1
X1 >= 0.533076 and X2 >= 0.228007 and !(X2 >= 0.424906) and X1 >= 0.708127 == 1
X1 >= 0.533076 and X2 >= 0.228007 and !(X2 >= 0.424906) and !(X1 >= 0.708127) and X2 >= 0.32625 and X1 >= 0.595471 and X1 >= 0.646007 == 1
X1 >= 0.533076 and X2 >= 0.228007 and !(X2 >= 0.424906) and !(X1 >= 0.708127) and X2 >= 0.32625 and X1 >= 0.595471 and !(X1 >= 0.646007) and X2 >= 0.403494 == 1
X1 >= 0.533076 and X2 >= 0.228007 and !(X2 >= 0.424906) and !(X1 >= 0.708127) and X2 >= 0.32625 and X1 >= 0.595471 and !(X1 >= 0.646007) and !(X2 >= 0.403494) == 0
X1 >= 0.533076 and X2 >= 0.228007 and !(X2 >= 0.424906) and !(X1 >= 0.708127) and X2 >= 0.32625 and !(X1 >= 0.595471) == 0
X1 >= 0.533076 and X2 >= 0.228007 and !(X2 >= 0.424906) and !(X1 >= 0.708127) and !(X2 >= 0.32625) == 0
X1 >= 0.533076 and !(X2 >= 0.228007) and X1 >= 0.887224 and X2 >= 0.037708 and X2 >= 0.082895 == 1
X1 >= 0.533076 and !(X2 >= 0.228007) and X1 >= 0.887224 and X2 >= 0.037708 and !(X2 >= 0.082895) and X1 >= 0.960783 == 1
X1 >= 0.533076 and !(X2 >= 0.228007) and X1 >= 0.887224 and X2 >= 0.037708 and !(X2 >= 0.082895) and !(X1 >= 0.960783) == 0
X1 >= 0.533076 and !(X2 >= 0.228007) and X1 >= 0.887224 and !(X2 >= 0.037708) == 0
X1 >= 0.533076 and !(X2 >= 0.228007) and !(X1 >= 0.887224) and X1 >= 0.850316 and X2 >= 0.169053 == 1
X1 >= 0.533076 and !(X2 >= 0.228007) and !(X1 >= 0.887224) and X1 >= 0.850316 and !(X2 >= 0.169053) == 0
X1 >= 0.533076 and !(X2 >= 0.228007) and !(X1 >= 0.887224) and !(X1 >= 0.850316) == 0

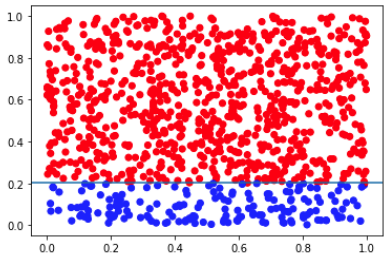
!(X1 >= 0.533076) and X2 >= 0.88635 and X1 >= 0.041245 and X1 >= 0.104043 == 1
!(X1 >= 0.533076) and X2 >= 0.88635 and X1 >= 0.041245 and !(X1 >= 0.104043) and X2 >= 0.964767 == 1
!(X1 >= 0.533076) and X2 >= 0.88635 and X1 >= 0.041245 and !(X1 >= 0.104043) and !(X2 >= 0.964767) == 0
!(X1 >= 0.533076) and X2 >= 0.88635 and !(X1 >= 0.041245) == 0
!(X1 >= 0.533076) and !(X2 >= 0.88635) and X2 >= 0.691474 and X1 >= 0.254049 == 1
!(X1 >= 0.533076) and !(X2 >= 0.88635) and X2 >= 0.691474 and !(X1 >= 0.254049) and X1 >= 0.191915 and X2 >= 0.792752 == 1
!(X1 >= 0.533076) and !(X2 >= 0.88635) and X2 >= 0.691474 and !(X1 >= 0.254049) and X1 >= 0.191915 and !(X2 >= 0.792752) == 0
!(X1 >= 0.533076) and !(X2 >= 0.88635) and X2 >= 0.691474 and !(X1 >= 0.254049) and !(X1 >= 0.191915) and X2 >= 0.864128 and X2 >= 0.865999 == 0
!(X1 >= 0.533076) and !(X2 >= 0.88635) and X2 >= 0.691474 and !(X1 >= 0.254049) and !(X1 >= 0.191915) and X2 >= 0.864128 and !(X2 >= 0.865999) == 1
!(X1 >= 0.533076) and !(X2 >= 0.88635) and X2 >= 0.691474 and !(X1 >= 0.254049) and !(X1 >= 0.191915) and !(X2 >= 0.864128) == 0
!(X1 >= 0.533076) and !(X2 >= 0.88635) and !(X2 >= 0.691474) and X2 >= 0.534979 and X1 >= 0.426073 == 1
!(X1 >= 0.533076) and !(X2 >= 0.88635) and !(X2 >= 0.691474) and X2 >= 0.534979 and !(X1 >= 0.426073) and X1 >= 0.409972 and X2 >= 0.597713 == 1
!(X1 >= 0.533076) and !(X2 >= 0.88635) and !(X2 >= 0.691474) and X2 >= 0.534979 and !(X1 >= 0.426073) and X1 >= 0.409972 and !(X2 >= 0.597713) == 0
!(X1 >= 0.533076) and !(X2 >= 0.88635) and !(X2 >= 0.691474) and X2 >= 0.534979 and !(X1 >= 0.426073) and !(X1 >= 0.409972) and X1 >= 0.393227 and X2 >= 0.639018 == 1
!(X1 >= 0.533076) and !(X2 >= 0.88635) and !(X2 >= 0.691474) and X2 >= 0.534979 and !(X1 >= 0.426073) and !(X1 >= 0.409972) and X1 >= 0.393227 and !(X2 >= 0.639018) == 0
!(X1 >= 0.533076) and !(X2 >= 0.88635) and !(X2 >= 0.691474) and X2 >= 0.534979 and !(X1 >= 0.426073) and !(X1 >= 0.409972) and !(X1 >= 0.393227) == 0
!(X1 >= 0.533076) and !(X2 >= 0.88635) and !(X2 >= 0.691474) and !(X2 >= 0.534979) == 0

Obviously, it is very hard to interpret this decision tree without visualizing the decision boundary of the dataset. The tree has 8 levels and around 60 nodes (including leaves).

Q6. (Hypothesis space) [10 pts] For D1.txt and D2.txt, do the following separately:
• Produce a scatter plot of the data set.
• Visualize your decision tree's decision boundary (or decision region, or some other ways to clearly visualize how your decision tree will make decisions in the feature space). Then discuss why the size of your decision trees on D1 and D2 differ. Relate this to the hypothesis space of our decision tree algorithm.

A: The scatter plot for the D1 decision tree is shown below. The decision boundary is a straight line represented by X2 = 0.201829. The data points above this line are classified with label 1 (shown as red) and the data points below this line are labeled 0 (shown as blue dots).

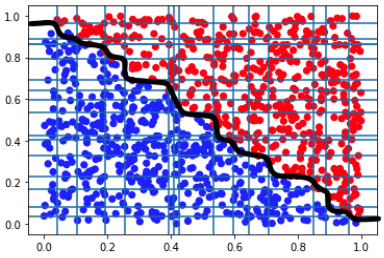
```
In [332]: plot_decision_boundary(D1_dataset, d3_root_D1, X1_lines, X2_lines)
```



The scatter plot and decision boundary (shown by the 'hand-drawn' black line (looks more like a step function)) for the D2 decision tree is shown below. The reason the D2 decision tree is very big and hard to interpret without the below visualization is because the step function cannot be represented as a simple boolean expression of X1 or X2 (like it can be done for D1). It is a combination of multiple boolean expressions involving X1 and X2 as can be seen in the logic rules shown in 'Q5' for the D2 decision tree.

The hypothesis space of the decision tree algorithm comprises all possible trees of varying sizes. The best fit is the tree which maximizes classification accuracy which in this case is a very large decision tree. For D1, this is represented as a tree with just 2 levels, the first level containing the root node splitting on X2 >= 0.201829 and the second level containing the left branch leaf (label: 1) and right branch leaf (label: 0)

```
In [335]: plot_decision_boundary(D2_dataset, d3_root_D2, X1_lines, X2_lines)
```



Q7. (Learning curve) [20 pts]We provide a data set Dbig.txt with 10000 labeled items. Caution: Dbig.txt is sorted.
• You will randomly split Dbig.txt into a candidate training set of 8192 items and a test set (the rest). Do this by generating a random permutation, and split at 8192. You should use as seed in the random generator, the last digit of your wisc-id (θ). A code for this task should be the following:

import numpy as np
np.random.RandomState(seed=Last-Digit).permutation(n)

Generate a sequence of five nested training sets D32 ⊂ D128 ⊂ D512 ⊂ D2048 ⊂ D8192 from the candidate training set. The subscript n in Dn denotes training set size. The easiest way

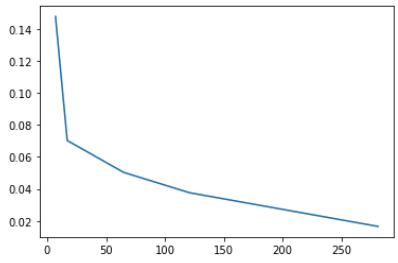
is to take the first n items from the (same) permutation above. (Use the same seed as above). This sequence simulates the real world situation where you obtain more and more training data.
• For each Dn above, train a decision tree. Measure its test set error errn. Show three things in your answer: (1) List n, number of nodes in that tree, errn. (2) Plot n vs. errn. This is known as a learning curve (a single plot). (3) Visualize your decision trees' decision boundary (five plots).

A: The below screenshot shows the number of nodes in decision trees constructed on D32, D128, D512, D2048 and D8192 respectively. The error is also shown with the n vs error plot.

N: [7, 17, 65, 121, 281]

Err: [0.14767699115044247,
0.0702433628318584,
0.05033185840707965,
0.03761061946902655,
0.016592920353982302]

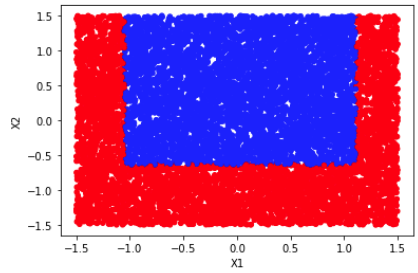
```
In [382]: # number of nodes in D32, D128, D512, D2048, D8192
          n

Out[382]: [7, 17, 65, 121, 281]

In [383]: # classification error on test set
          err

Out[383]: [0.14767699115044247,
           0.0702433628318584,
           0.05033185840707965,
           0.03761061946902655,
           0.016592920353982302]

In [384]: # n vs err
          plt.plot(n, err)

Out[384]: [<matplotlib.lines.Line2D at 0x7f9fbc0a6670>]
```



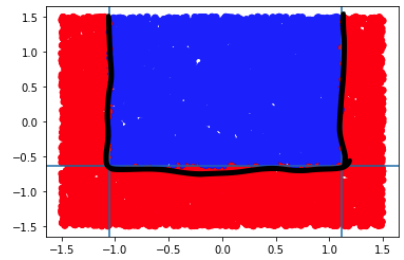
The decision boundaries are as shown below:

- D32

In [287]:
```
decision_boundary(Dbig_permutation, d32_d3)
```

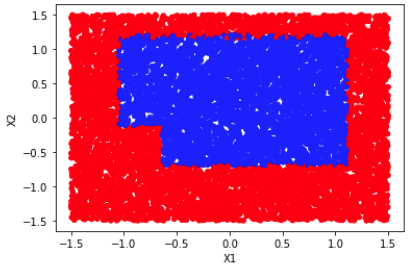


In [337]:
```
X1_lines, X2_lines = fetch_decision_lines(d32_d3)

plot_decision_boundary(Dbig_permutation, d32_d3, X1_lines, X2_lines)
```

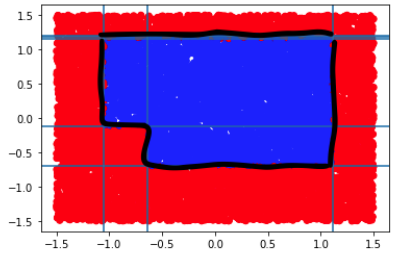


- D128

In [288]:
```
decision_boundary(Dbig_permutation, d128_d3)
```

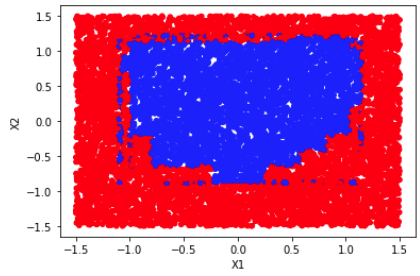


In [338]:
```
X1_lines, X2_lines = fetch_decision_lines(d128_d3)

plot_decision_boundary(Dbig_permutation, d128_d3, X1_lines, X2_lines)
```

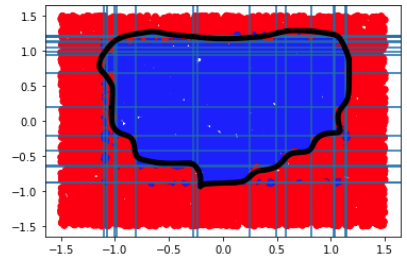


- D512

In [289]:
```
decision_boundary(Dbig_permutation, d512_d3)
```

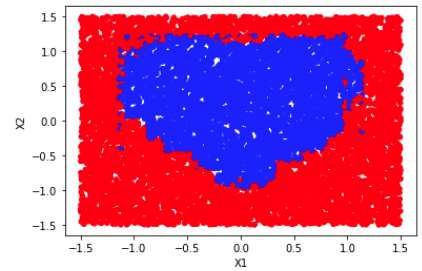


In [339]:
```
X1_lines, X2_lines = fetch_decision_lines(d512_d3)

plot_decision_boundary(Dbig_permutation, d512_d3, X1_lines, X2_lines)
```

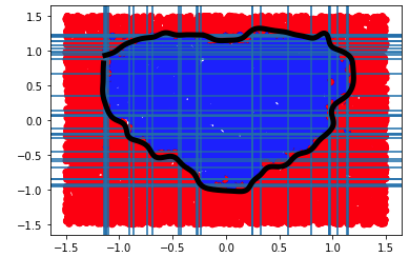


- D2048

In [290]:
```
decision_boundary(Dbig_permutation, d2048_d3)
```

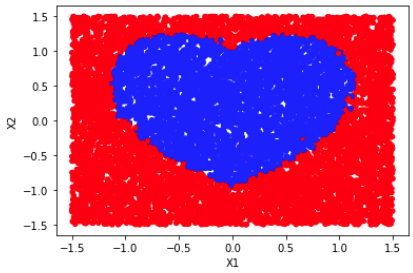


In [340]:
```
X1_lines, X2_lines = fetch_decision_lines(d2048_d3)

plot_decision_boundary(Dbig_permutation, d2048_d3, X1_lines, X2_lines)
```

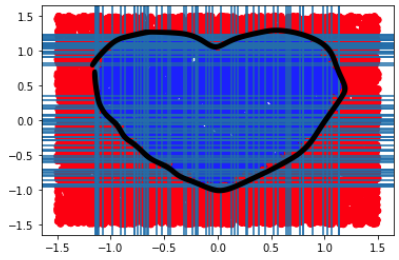


- D8192

```
In [291]: decision_boundary(Dbig_permutation, d8192_d3)
```




```
In [341]: X1_lines, X2_lines = fetch_decision_lines(d8192_d3)

          plot_decision_boundary(Dbig_permutation, d8192_d3, X1_lines, X2_lines)
```



[It is very hard to depict some of the smaller decision boundary regions accurately with a hand-drawn version :)]

2. sklearn

[10 pts] Learn to use sklearn https://scikit-learn.org/stable/. Use sklearn.tree.DecisionTreeClassifier to produce trees for datasets D32,D128, . . .D8192. Show two things in your answer: (1) List n, number of nodes in that tree, errn. (2) Plot n vs. errn.

A:

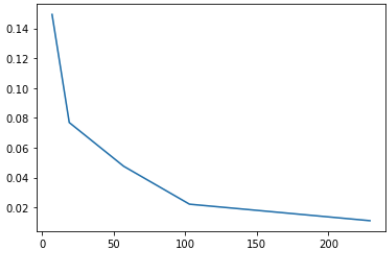
1. Number of nodes in D32, D128, D512, D2048 and D8192 respectively are shown below:

   N = [7, 19, 57, 103, 229]

2. The classification error for D32, D128, D512, D2048 and D8192 respectively are shown below:

   Err = [0.1493362831858407,
   0.07688053097345138,
   0.047566371681415975,
   0.02212389380530977,
   0.011061946902654829]

```
In [173]: plt.plot(n, err)
Out[173]: [<matplotlib.lines.Line2D at 0x7f9b7a7fc8e0>]
```




```
In [174]: err
Out[174]: [0.1493362831858407,
           0.07688053097345138,
           0.047566371681415975,
           0.02212389380530977,
           0.011061946902654829]

In [175]: n
Out[175]: [7, 19, 57, 103, 229]
```

3. Lagrange Interpolation

[10 pts] Fix the interval [0, 4π] and sample n = 100 points x from this interval (a) uniformly (b) by drawing samples according to a Gaussian distribution with mean μ = 2π − θπ/10 and standard deviation σ = π/6 and rejecting any sample outside the interval. Use these to build a training set consisting of n pairs (x, y) by setting function y = cos(x + θπ/10), θ is again the last digit of your id.

Build a model f by using Lagrange interpolation, as discussed in lecture. Generate a test set using the same distribution as your test set. Compute and report the resulting model's train and test error. What do you observe? What do you observe if you increase σ to be π/4 and π/2?

A:

a. Uniform:

```
In [16]: # Train Error
         np.mean(np.abs(uniform_dist_train_pred - uniform_dist_y_train))
Out[16]: 1700806403111.0098

In [17]: # Test Error
         np.mean(np.abs(uniform_dist_pred - uniform_dist_y_test))
Out[17]: 1100677.0316331435
```

Train error: 1700806403111.0098 (0 if the point for which lagrange is calculated is included in the lagrange function)
Test error: 1100677.0316331435

b. Gaussian Sample:
    i. Sigma = pi/6

```
In [25]: # Train Error
         np.mean(np.abs(normal_dist_pb6_train_pred - normal_dist_pb6_y_train))

Out[25]: 1.067385941921518e+58

In [26]: # Test Error
         np.mean(np.abs(normal_dist_pb6_pred - normal_dist_pb6_y_test))

Out[26]: 1.1809409536427378e+54
```

Train error: 1.067385941921518e+58 (0 if the point for which lagrange is calculated is included in the lagrange function)
Test error: 1.1809409536427378e+54

ii. Sigma = pi/4

```
In [33]: # Train Error
         np.mean(np.abs(normal_dist_pb4_train_pred - normal_dist_pb4_y_train))

Out[33]: 1.1861805350437908e+55

In [34]: # Test Error
         np.mean(np.abs(normal_dist_pb4_pred - normal_dist_pb4_y_test))

Out[34]: 7.102691120150103e+46
```

Train error: 1.1861805350437908e+55 (0 if the point for which lagrange is calculated is included in the lagrange function)
Test error: 7.102691120150103e+46

iii. Sigma = pi/2

```
In [41]: # Train Error
         np.mean(np.abs(normal_dist_pb2_train_pred - normal_dist_pb2_y_train))

Out[41]: 1.3884450232724312e+47

In [42]: # Test Error
         np.mean(np.abs(normal_dist_pb2_pred - normal_dist_pb2_y_test))

Out[42]: 1.191591564428518e+31
```

Train error: 1.3884450232724312e+47 (0 if the point for which lagrange is calculated is included in the lagrange function)
Test error: 1.191591564428518e+31

Each train set consists of 100 points and the test set consists of 20 points from their respective distributions.

Observations:

1. The error magnitude seems to indicate that the lagrange interpolation doesn't fit the data distribution well with error tending to very very large values (both for train and test set)
2. It is also observed that the error for the sample from the uniform distribution is smaller than the error for the samples from the normal distributions, indicating that the lagrange interpolation seems to fit the uniform distribution better than the normal distribution

3. The last observation is that as the standard deviation of the normal distribution increases the error decreases, indicating that the lagrange interpolation seems to fit data with a higher spread/deviation better.
4. The Lagrange interpolation has extreme overfitting on the training dataset. Each training data point is a root of the lagrange function that is being calculated.

Note: In the above case, train error is computed by excluding the point for which the lagrange value is being computed. If the train point is included in the function, then the curve will exactly go through the point since the point is a root of this curve. In which case the train error is always 0 for any distribution!