Data Preprocessing:

Load dataset and split data into training, validation, and testing data

```
import sklearn
import numpy as np
from operator import index
from numpy import number
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import seaborn as sn
from sklearn.metrics import classification_report, confusion_matrix

# Load breast cancer dataset
data = sklearn.datasets.load_breast_cancer()

label = data.target
data = data.data

# Divide data into training, validation, and testing data. Data is
trainingFeatures, remainingFeatures, trainingLabels, remainingLabel
validationFeatures, testingFeatures, validationLabels, testingLabel
```

Creating a "Node" class, which creates the nodes found in the tree

```
class Node:
  def __init__(self, featureChosen=None, threshold=None, left=None,
    # Intializing each node with the values provided to it from the
    # Each node would contain the information gain associated with
    self.featureChosen = featureChosen
    self.threshold = threshold
    self.left = left
    self.right = right
    self.informationGain = informationGain
    self.entropy = entropy
    self.value = value
```

Creating a "Tree" class, which handles the logic for the implementation of the entire tree

```
class DecisionTree:
  def __init__(self, minSamplesSplit=2, maxDepth=2):
```

```
    def __init__(self, minSamplesSplit=2, maxDepth=2):
      # Intializes the root of the tree. Placed as None as the tree or
      self.root = None
      # Identifies the stopping condition for the tree.
      self.minSamplesSplit = minSamplesSplit
      self.maxDepth = maxDepth

    def makeTree(self, trainingFeatures, trainingLabels, currentDepth=
      numberofSamples = trainingLabels.shape[0]
      numberofFeatures = trainingFeatures.shape[1]
      pureSplit = len(np.unique(trainingLabels)) == 1

      # Account for stopping conditions
      if currentDepth <= self.maxDepth and numberofSamples >= self.min
        # Get optimal split
        optimalSplit = self.getOptimalSplit(trainingFeatures, training

        # Ensure current optimal split has a positive information gain
        if optimalSplit["informationGain"] > 0:
          # Build left subtree
          leftSubtree = self.makeTree(optimalSplit["leftFeatures"], op

          # Build right subtree
          rightSubtree = self.makeTree(optimalSplit["rightFeatures"],

          return Node(optimalSplit["featureChosen"], optimalSplit["thr

      # If stopping conditions are met, then this is currently a leaf
      leaf = self.makeLeafNode(trainingLabels)
      return Node(value=leaf)

    def getOptimalSplit(self, trainingFeatures, trainingLabels, number
      # Create dictionary to store best split
      bestSplit = {}
      # Intialize the maximum information gain with minus infinity
      # This is crucial as the greedy algorithm must intially start fr
      maxInformationGain = -float("inf")

      # Loop over all possible features to determine which feature wil
      for featureIndex in range(numberofFeatures):
        featureValues = trainingFeatures[:, featureIndex]
        possibleThresholds = np.unique(featureValues)
        for threshold in possibleThresholds:
          # Split according to current given values
          dataLeft, dataRight, labelsLeft, labelsRight = self.split(tra
          # Ensure both left and right children are not null
          if len(dataLeft) > 0 and len(dataRight) > 0:
            informationGain = self.informationGain(trainingLabels, labe
            # Update current split if the information gain obtained fr
```

```
            if informationGain > maxInformationGain:
              bestSplit["featureChosen"] = featureIndex
              bestSplit["threshold"] = threshold
              bestSplit["leftFeatures"] = dataLeft
              bestSplit["leftLabels"] = labelsLeft
              bestSplit["rightFeatures"] = dataRight
              bestSplit["rightLabels"] = labelsRight
              bestSplit["informationGain"] = informationGain
              maxInformationGain = informationGain

    # Return the current best split
    return bestSplit

  def split(self, trainingFeatures, trainingLabels, featureIndex, th
    # Split the data and labels into left and right data and features
    dataLeft = np.array([row for row in trainingFeatures if row[featu
    dataRight = np.array([row for row in trainingFeatures if row[fea
    labelsLeft = trainingLabels[trainingFeatures[:, featureIndex] <=
    labelsRight = trainingLabels[trainingFeatures[:, featureIndex] >
    return dataLeft, dataRight, labelsLeft, labelsRight

  def makeLeafNode(self, trainingLabels):
    #
    trainingLabels = np.array(trainingLabels)
    return max(set(trainingLabels), key=list(trainingLabels).count)

  def getEntropy(self,trainingLabels):
    labels = np.unique(trainingLabels)
    # Intialize entropy with zero
    entropy = 0
    for label in labels:
      # Calculate the probability of occurence of the label
      probability = len(trainingLabels[trainingLabels == label]) / le
      entropy += -probability * np.log2(probability)
    return entropy

  def informationGain(self, labelCurrent, labelLeft, labelRight):
    # Get the weight of the left and right children from the total nu
    weightedLeft = len(labelLeft) / len(labelCurrent)
    weightedRight = len(labelRight) / len(labelCurrent)

    # Calculate the information gain
    infoGain = self.getEntropy(labelCurrent) - (weightedLeft * self.g
    return infoGain

  def printTree(self, tree=None, indent=" "):
    if not tree:
      tree = self.root
```

```
        if tree.value is not None:
          if tree.value == 0:
            print("Malignant")
          elif tree.value == 1:
            print("Benign")
          else:
            print(tree.value)
        else:
          print("X"+str(tree.featureChosen), "<=", tree.threshold, "?")
          print("%sInformation Gain: " % (indent),tree.informationGain)
          print("%sleft:" % (indent), end="")
          self.printTree(tree.left, indent + indent)
          print("%sright:" % (indent), end="")
          self.printTree(tree.right, indent + indent)

      def fit(self, trainingFeatures, trainingLabels):
        self.root = self.makeTree(trainingFeatures, trainingLabels)

      def predict(self, testingFeatures):
        predictions = [self.makePredictions(row, self.root) for row in te
        return predictions

      def makePredictions(self, testingFeatures, tree):
        # If currently at a leaf node, predict
        if tree.value != None:
          return tree.value
        featureValue = testingFeatures[tree.featureChosen]
        # Else compare the featureValue to the current level tree thresho
        if featureValue <= tree.threshold:
          return self.makePredictions(testingFeatures, tree.left)
        else:
        # Else, the right subtree will be traveresed
          return self.makePredictions(testingFeatures, tree.right)

      def accuracy(self, testingFeatures, testingLabels):
        predictions = self.predict(testingFeatures)
        return np.mean(np.array(predictions) == np.array(testingLabels))
```

```
tree = DecisionTree(minSamplesSplit=3,maxDepth=3)
tree.fit(trainingFeatures, trainingLabels)
tree.printTree()
```

```
X22 <= 105.9 ?
 Information Gain:  0.645947668978021
 left:X27 <= 0.1423 ?
  Information Gain:  0.09557508006087603
  left:X3 <= 690.2 ?
    Information Gain:  0.03145500992451482
    left:X14 <= 0.00328 ?
        Information Gain:  0.023338451601787248
        left:Benign
        right:Benign
    right:Malignant
  right:X13 <= 18.15 ?
    Information Gain:  0.6394725269414906
    left:X1 <= 20.22 ?
        Information Gain:  0.6500224216483541
        left:Benign
        right:Malignant
    right:Malignant
 right:X27 <= 0.1489 ?
  Information Gain:  0.2080739414679902
  left:X21 <= 19.31 ?
    Information Gain:  0.29469601591956374
    left:Benign
    right:X20 <= 16.76 ?
        Information Gain:  0.3515358797217579
        left:Benign
        right:Malignant
  right:Malignant
```

```
# Testing the tree
print("Tree Accuracy on Validation Data:", tree.accuracy(validation
```

```
Tree Accuracy on Validation Data: 0.9176470588235294
```

Fine Tuning the Hyperparameters:

Maximum Tree Depth and Minimum Samples Per Split

```
def tuning(minSamplesSplit, maxDepth):
  # Create a new instance of a tree with every pass
  trainingTree = DecisionTree(minSamplesSplit=minSamplesSplit,maxDe
  # Fit the tree with the training labels and features
  trainingTree.fit(trainingFeatures, trainingLabels)
  # Test the tree with the validation labels and features
  return trainingTree.accuracy(validationFeatures, validationLabels

def multipleRuns(minSamplesSplit=[2,5,10], maxDepth=[2,4,6,8,10]):
  # Sentinal values (intial guard values)
  optimalSampleSplit = 0
  optimalMaxDepth = 0
  maxAccuracy = 0

  # Attempting every possible combination of minimum samples split
  for sampleSplit in minSamplesSplit:
    for depth in maxDepth:
      accuracy = tuning(sampleSplit, depth)
      # Changing the values of the maximum accuracy, optimal sample
      if accuracy > maxAccuracy:
        maxAccuracy = accuracy
        optimalSampleSplit = sampleSplit
        optimalMaxDepth = depth
  print("Hyperparameter Tuning:")
  print("Minimum Samples Per Split:", optimalSampleSplit)
  print("Maximum Tree Depth:", optimalMaxDepth)
  print("Maximum Accuracy:", maxAccuracy)
  return optimalSampleSplit, optimalMaxDepth, maxAccuracy

minSampleSplit, maxDepth, maxAccuracy = multipleRuns()
```

```
Hyperparameter Tuning:
Minimum Samples Per Split: 2
Maximum Tree Depth: 4
Maximum Accuracy: 0.8941176470588236
```

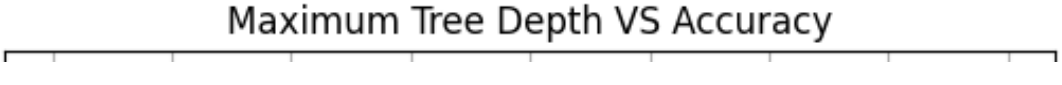
```
tree = DecisionTree(minSamplesSplit=minSampleSplit,maxDepth=maxDept
tree.fit(trainingFeatures, trainingLabels)
# Test optimal minSamplesSplit and maxDepth on testing data
predictions = tree.predict(testingFeatures)
print("Tree Accuracy on Testing Data:", tree.accuracy(testingFeatur
```

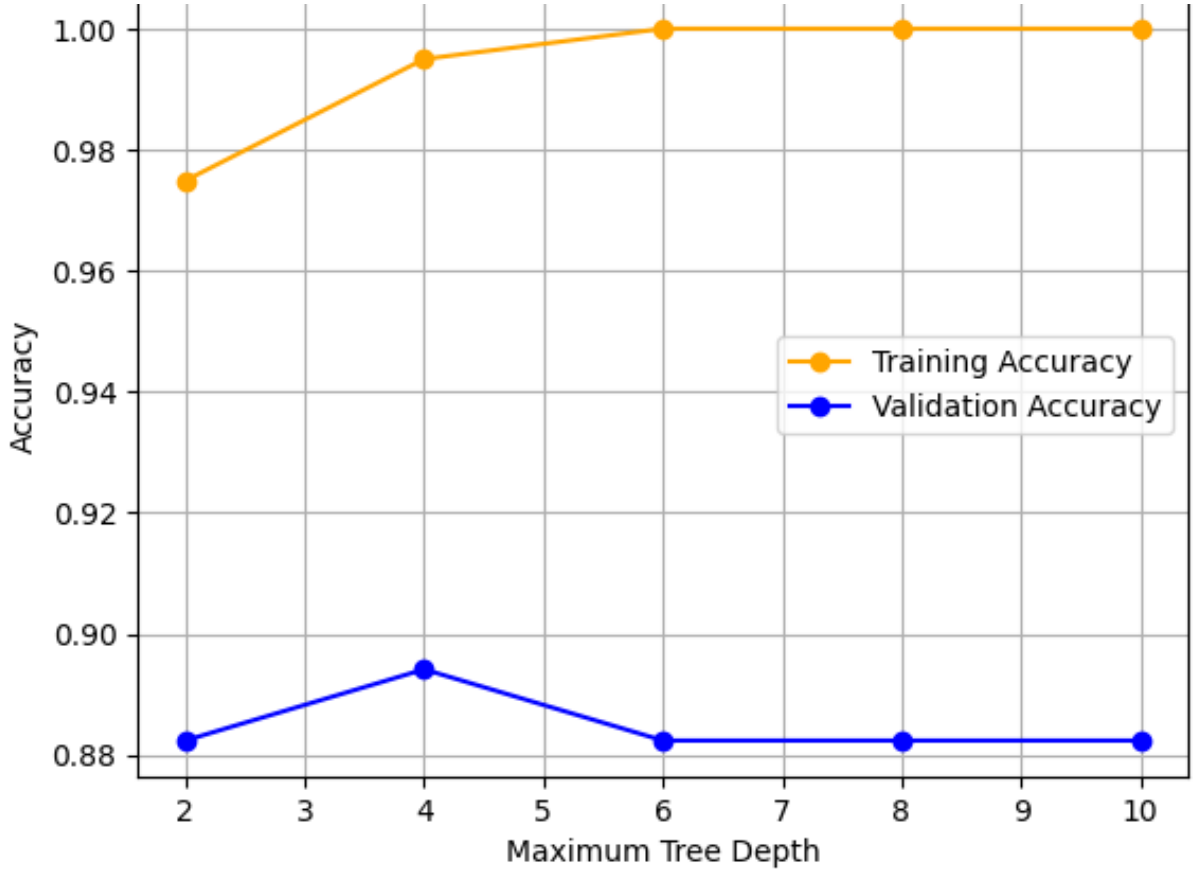
```
Tree Accuracy on Testing Data: 0.9302325581395349
```

```
# Examining training and validation accuracy change with maximum de
samples=[2,4,6,8,10]
```

```
trainingAccuracyData=[]
validationAccuracyData=[]
def accuracyChange(samples=[2,4,6,8,10]):
  # Changing the maximum depth only while keeping the minimum sampl
  for depth in samples:
    # Create a new instance of a tree with every pass
    trainingTree = DecisionTree(minSamplesSplit=2,maxDepth=depth)
    trainingTree.fit(trainingFeatures, trainingLabels)
    # Store the tree's accuracy on the training and validation data
    trainingAccuracy = trainingTree.accuracy(trainingFeatures, trai
    validationAccuracy = trainingTree.accuracy(validationFeatures,
    print("Maximum Tree Depth:", depth)
    print("Training Accuracy:", trainingAccuracy)
    print("Validation Accuracy:", validationAccuracy)
    trainingAccuracyData.append(trainingAccuracy)
    validationAccuracyData.append(validationAccuracy)

# Plottting
def plotAccuracyChange(samples, trainingAccuracyData, validationAcc
  plt.plot(samples, trainingAccuracyData, label='Training Accuracy'
  plt.plot(samples, validationAccuracyData, label='Validation Accur
  plt.xlabel('Maximum Tree Depth')
  plt.ylabel('Accuracy')
  plt.legend()
  plt.grid()
  plt.title('Maximum Tree Depth VS Accuracy')
  plt.show()

accuracyChange()
plotAccuracyChange(samples, trainingAccuracyData, validationAccurac
```

```
Maximum Tree Depth: 2
Training Accuracy: 0.9748743718592965
Validation Accuracy: 0.8823529411764706
Maximum Tree Depth: 4
Training Accuracy: 0.9949748743718593
Validation Accuracy: 0.8941176470588236
Maximum Tree Depth: 6
Training Accuracy: 1.0
Validation Accuracy: 0.8823529411764706
Maximum Tree Depth: 8
Training Accuracy: 1.0
Validation Accuracy: 0.8823529411764706
Maximum Tree Depth: 10
Training Accuracy: 1.0
Validation Accuracy: 0.8823529411764706
```





The results of the accuracy of the training and validation data versus the the tree depth present the following findings:

1- The deeper the tree was allowed to become, the easier it overfitted.

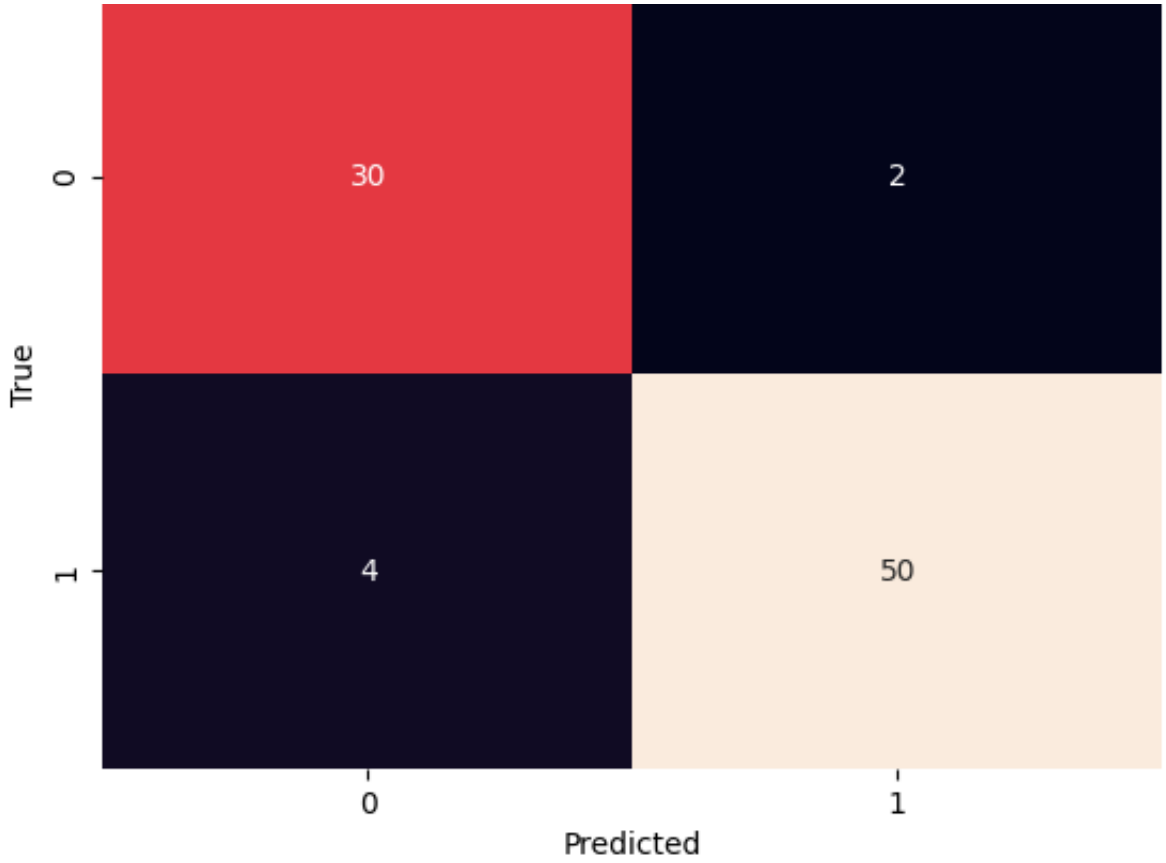
2- A depth of 6 or more would leave the tree having well defined aspects for the training data (no misses would result and the accuracy of the training data would be 100%)

3- The performance of the validation data does not vary much with the change in the depth. This is predicted as the tree originally splits on the most distinctive features for 70% of the provided data (training data). A similar trend would be expected in the validation and testing data.

```
    tree = DecisionTree(minSamplesSplit=minSampleSplit,maxDepth=maxDept
    tree.fit(trainingFeatures, trainingLabels)
    # Test optimal minSamplesSplit and maxDepth on testing data
    predictions = tree.predict(testingFeatures)

    numpyPredictions = np.array(predictions)
    numpyActualLabels = np.array(testingLabels)
```

```
# Performance Metrics

# Calculating the precision, recall, and f-score
# Precision: How many positively predicted instances the model pred
# Recall: Out of all of the positive instances, how many did the mo
# F1-Score: Harmonic mean of both the precision and the recall. Hea
true_positives = np.sum(np.logical_and(numpyPredictions == 0, numpy
true_negatives = np.sum(np.logical_and(numpyPredictions == 1, numpy
false_positives = np.sum(np.logical_and(numpyPredictions == 0, nump
false_negatives = np.sum(np.logical_and(numpyPredictions == 1, nump

precision = true_positives / (true_positives + false_positives)
recall = true_positives / (true_positives + false_negatives)
f1_score = 2 * ((precision * recall) / (precision + recall))

print("Classification Report for Testing Data:")
print("------------")
print("Benign: Label 0 || Malignant: Label 1")
print("Precision: ",precision)
print("Recall: ",recall)
print("F1-Score: ",f1_score)
print("------------")

# Confusion Matrix

print("Confusion Matrix for Testing Data:")
print("------------")

confusionMatrix = confusion_matrix(testingLabels, predictions)
plt.figure()
sn.heatmap(confusionMatrix, annot=True, fmt="d", cbar=False)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix for Testing Data:')
plt.show()
```

```
Classification Report for Testing Data:
------------
Benign: Label 0 || Malignant: Label 1
Precision:  0.8823529411764706
Recall:  0.9375
F1-Score:  0.9090909090909091
------------
Confusion Matrix for Testing Data:
------------
```



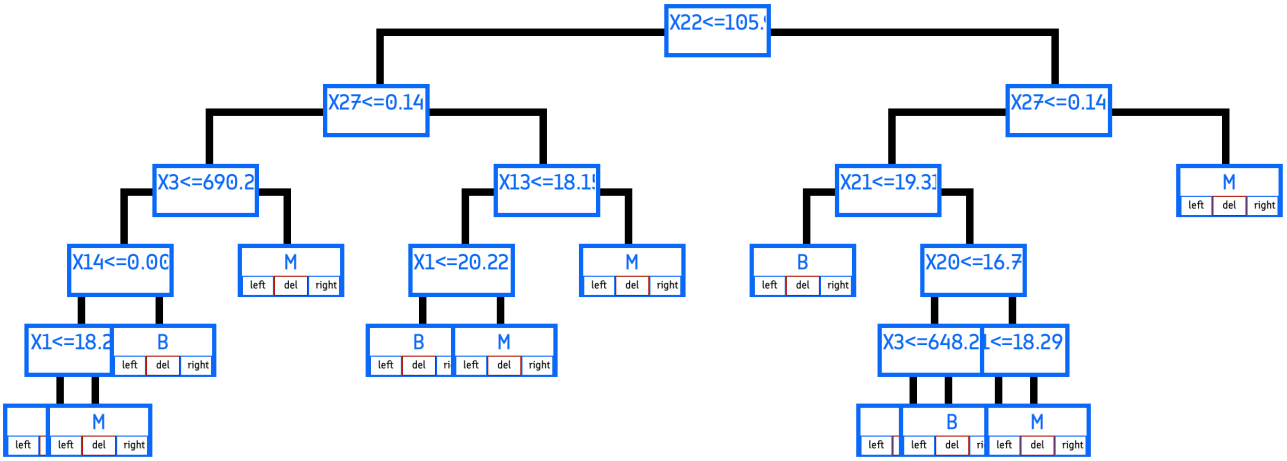
The tree correctly identifies 30 malignant cases and 50 benign cases. Unfortunately, the tree misses out on 6 samples, 4 of which are benign but predicted malignant and 2 of which are malignant but predicted benign. Although the last two samples are definetly misleading, the tree's main false identifications are tolerable ones (classifying as malignant while being benign, which is much better than the other way around).

```
# Commenting on the feature importance by printing the tree:
tree.printTree()
```

```
X22 <= 105.9 ?
 Information Gain:  0.645947668978021
 left:X27 <= 0.1423 ?
  Information Gain:  0.09557508006087603
  left:X3 <= 690.2 ?
    Information Gain:  0.03145500992451482
    left:X14 <= 0.00328 ?
        Information Gain:  0.023338451601787248
        left:X1 <= 18.22 ?
                Information Gain:  0.6500224216483541
                left:Benign
                right:Malignant
        right:Benign
    right:Malignant
  right:X13 <= 18.15 ?
    Information Gain:  0.6394725269414906
    left:X1 <= 20.22 ?
        Information Gain:  0.6500224216483541
        left:Benign
        right:Malignant
    right:Malignant
 right:X27 <= 0.1489 ?
  Information Gain:  0.2080739414679902
  left:X21 <= 19.31 ?
    Information Gain:  0.29469601591956374
    left:Benign
    right:X20 <= 16.76 ?
        Information Gain:  0.3515358797217579
        left:X3 <= 648.2 ?
                Information Gain:  0.5567796494470396
                left:Malignant
                right:Benign
        right:X1 <= 18.29 ?
                Information Gain:  0.13509531357637425
                left:Malignant
                right:Malignant
  right:Malignant
```

The output of the tree print portrays how every split was done to determine the most distinctive features chosen to split on. Each split was made according to the feature split that yielded the highest information gain.

Tree Visualized



Given the provided image, the most distinctive feature that intially splits the tree is X22, followed by X27. These two are shared by both trees in both levels.