Dataset Downloading

```
!wget https://archive.ics.uci.edu/ml/machine-learning-databases/00475/audit_data.z
!unzip audit_data.zip
!wget https://archive.ics.uci.edu/ml/machine-learning-databases/00275/Bike-Sharing
!unzip Bike-Sharing-Dataset.zip
```

```
--2023-05-06 06:07:37--  https://archive.ics.uci.edu/ml/machine-learning-data
Resolving archive.ics.uci.edu (archive.ics.uci.edu)... 128.195.10.252
Connecting to archive.ics.uci.edu (archive.ics.uci.edu)|128.195.10.252|:443..
HTTP request sent, awaiting response... 200 OK
Length: 28447 (28K) [application/x-httpd-php]
Saving to: 'audit_data.zip'

audit_data.zip      100%[===================>]  27.78K  --.-KB/s    in 0.1s

2023-05-06 06:07:38 (214 KB/s) - 'audit_data.zip' saved [28447/28447]

Archive:  audit_data.zip
  inflating: audit_data/audit_risk.csv
  inflating: audit_data/trial.csv
--2023-05-06 06:07:38--  https://archive.ics.uci.edu/ml/machine-learning-data
Resolving archive.ics.uci.edu (archive.ics.uci.edu)... 128.195.10.252
Connecting to archive.ics.uci.edu (archive.ics.uci.edu)|128.195.10.252|:443..
HTTP request sent, awaiting response... 200 OK
Length: 279992 (273K) [application/x-httpd-php]
Saving to: 'Bike-Sharing-Dataset.zip'

Bike-Sharing-Datase 100%[===================>] 273.43K   527KB/s    in 0.5s

2023-05-06 06:07:39 (527 KB/s) - 'Bike-Sharing-Dataset.zip' saved [279992/279

Archive:  Bike-Sharing-Dataset.zip
  inflating: Readme.txt
  inflating: day.csv
  inflating: hour.csv
```

Part 1 (DT Classifier)

Implementation ...:

```
import numpy as np
import math
```

```
# Extract the midpoints from given feature column
def get_midpoints(feature_data):
    # Sort the data in ascending order
    sorted_data = sorted(feature_data)
    # Remove duplicates
    sorted_data = np.unique(sorted_data)

    # Find midpoints between adjacent values
    midpoints = []
    for i in range(len(sorted_data) - 1):
        midpoint = (sorted_data[i] + sorted_data[i+1]) / 2.0
        midpoints.append( midpoint )

    return midpoints


# Class for decision nodes
class Node:

    # Init function
    def __init__(self, x, y, attribute_types):
        self.x = x
        self.y = y
        self.attribute_types = attribute_types
        self.left = None
        self.right = None
        self.class_label = None     # If the node is leaf node

        self.decision_value = None
        self.decision_type = None
        self.decision_attribute = None

    # Calculate the entropy of the node with their data
    def calc_entropy(self):
        entropy = 0
        classes = np.unique(self.y)
        for cls in classes:
            probab = np.count_nonzero(self.y == cls) / self.y.size
            entropy -= probab * math.log2(probab)
        return entropy

    # Find all split points on the data
    def find_split_pts(self):
        split_pts = list()

        # Traverse all attributes
        for (i,attribute) in enumerate(self.attribute_types):

            # Numeric
            if(attribute == 1):
                mid_pts = get_midpoints(self.x[:,i])
                split_pts += [(i,x) for x in mid_pts]

            # Categoric
            elif(attribute == 2):
```

```
                uniques = np.unique(self.x[:,i])

                # Binary class --> just add one of them
                if(len(uniques) == 2):
                    split_pts.append( (i, uniques[0]) )

                # Multiclass --> add each one
                elif(len(uniques) > 2):
                    split_pts +=  [ (i,x) for x in uniques ]

                # If It has only one unique class for this data then do not add it

            # Invalid attribute type
            else:
                raise TypeError("Attribute type is not valid. Only 1 and 2 is vali

        return  split_pts


    # Generate left and right nodes with given split point
    def generate_nodes(self, split_pt):
        decision_type = self.attribute_types[split_pt[0]]
        decision_value = split_pt[1]

        # Numeric
        if(decision_type == 1):
            mask = self.x[:,split_pt[0]] < decision_value

        # Categoric
        elif(decision_type == 2):
            mask = self.x[:,split_pt[0]] == decision_value

        # Invalid attribute type
        else:
            raise TypeError("Attribute type is not valid. Only 1 and 2 is valid ")

        left = Node(self.x[mask], self.y[mask], self.attribute_types)
        right = Node(self.x[~mask], self.y[~mask], self.attribute_types)

        return (left, right)


    # Calculate the decision score by using left and right nodes.
    # Explanation: Calculate the entropies of both left and right nodes and normal
    def calc_decision_score(self, node_left:'Node', node_right:'Node'):

        # Sizes of nodes
        left_size = float( len(node_left.x) )
        right_size = float( len(node_right.x) )
        total = left_size + right_size

        # Normalization
        score = left_size / total * node_left.calc_entropy() + right_size / total

        return score
```

```
    # Predict the given data
    # Go until leaf node recursively and return class label
    def predict(self, data):

        if(self.class_label != None):
            return self.class_label

        # Numeric
        if(self.decision_type == 1):
           if( data[self.decision_attribute] < self.decision_value):
               return self.left.predict(data)

           else:
               return self.right.predict(data)

        # Categoric
        elif(self.decision_type == 2):
            if( data[self.decision_attribute] == self.decision_value):
               return self.left.predict(data)

            else:
                return self.right.predict(data)


        # Invalid attribute type
        else:
            raise TypeError("Attribute type is not valid. Only 1 and 2 is valid ")



# Recursive and entropy based DT generation algorithm  by using greedy algorithm
# Take data included root node and generate the DT recursively
def generate_tree(node:Node , max_depth):

    # If max depth is reached, then label the leaf node and terminates
    if(max_depth <= 0):
        counts = np.bincount(node.y)
        most_freq = np.argmax(counts)
        node.class_label = node.y[most_freq]
        return

    # If the entropy of the current node is 0 then no need to continue anymore
    # Label the leaf node and return
    if(node.calc_entropy() == 0):
        node.class_label = node.y[0]
        return

    # Find all split points
    split_points = node.find_split_pts()

    # Variables to hold best split
    best_split = None
    best_score = 1.1
```

```
        nodes = None

        # Calculate scores of all split points and get the best split
        for split_pt in split_points:

            # Generate the child nodes with split
            node_left, node_right = node.generate_nodes(split_pt)

            # Calculate score
            score = node.calc_decision_score(node_left, node_right)

            if( score < best_score ):
                best_score = score
                best_split = split_pt
                nodes = (node_left, node_right)


        # End of For : Best split found.

        # Place the children to the left and right
        node.left, node.right = nodes[0],nodes[1]

        # Place the decision value and data type on the node
        node.decision_value = best_split[1]
        node.decision_type = node.attribute_types[best_split[0]]
        node.decision_attribute = best_split[0]

        # Recursive call for children
        generate_tree(node.left, max_depth-1)
        generate_tree(node.right, max_depth-1)




    # DT builder
    def buid_dt(X, y, attribute_types, max_depth):
        root = Node(X,y,attribute_types)
        generate_tree(root, max_depth)
        return root





    # Takes DT and X matrix returns a vector for predicted predicted labels
    def predict_dt(dt:Node, X):
        predict_vector = [dt.predict(x) for x in X]
        return np.array( predict_vector )




    import pandas as pd

    df1 = pd.read_csv("/content/audit_data/trial.csv")
```

```
# Handling missing and NaN values
# The values that can not cast to number will be NaN
df1["LOCATION_ID"] = pd.to_numeric( df1["LOCATION_ID"], errors='coerce' )
df1["LOCATION_ID"].isna().sum()
df1 = df1.dropna()

# Split x and y
Y = df1["Risk"].values
X = df1.drop("Risk",axis=1).values

attribute_types = [2,2,1,2,1,2,1,2,2,1,2,2,2,2,2,2,2]


from sklearn.model_selection import KFold
from sklearn.metrics import confusion_matrix, classification_report

k_fold = KFold(n_splits=6, shuffle=True, random_state=42)
```

Results:

```
for k, (train, test) in enumerate(k_fold.split(X, Y)):
  # Train
  dt = buid_dt(X[train], Y[train],attribute_types,5)
  y_pred = predict_dt(dt, X[test])
  print("\n\nFold",k,":")
  # Confusion matrix
  conf_mat = confusion_matrix(Y[test], y_pred)
  # Display confusion matrix
  cm_df = pd.DataFrame(conf_mat, columns=['Predicted 0', 'Predicted 1'], index=['T
  print('Confusion matrix:')
  print(cm_df)
  print("\nResult:")
  print( classification_report(Y[test],y_pred) )
  print()
```

```
    accuracy                           1.00       129
   macro avg       1.00      1.00      1.00       129
weighted avg       1.00      1.00      1.00       129




Fold 4 :
Confusion matrix:
        Predicted 0  Predicted 1
True 0           50            0
True 1            0           78

Result:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       1.00      1.00      1.00        78

    accuracy                           1.00       128
   macro avg       1.00      1.00      1.00       128
weighted avg       1.00      1.00      1.00       128




Fold 5 :
Confusion matrix:
        Predicted 0  Predicted 1
True 0           49            0
True 1            0           79

Result:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        49
           1       1.00      1.00      1.00        79

    accuracy                           1.00       128
   macro avg       1.00      1.00      1.00       128
weighted avg       1.00      1.00      1.00       128
```

Comments and discussion:

In this part, I implemented the DT builder as mentioned in the class. All techniques and methods are same as told in the class. The main idea is finding middle points of all features and calculate the score by using left and right entropy and normalize them. Then we take the best split point and put left and right nodes. We do this operation until the node is fully pure or max depth is reached. This algorithm works recursively as shown in the class. My first aim here was learning. So i tried the codes to be understandable as much as possible. To ease understanding and implementation i used node class. It contains the data on the node and related member functions like calc_entropy, find_split_pts etc. You may follow the comment lines for better understandanding of code. Results were amazing for me. It's same with sci-kit DT library.

Part 2 (DT Regressor)

Implementation ...:

```
import numpy as np
import math



# Extract the midpoints from given feature column
def get_midpoints(feature_data):
    # Sort the data in ascending order
    sorted_data = sorted(feature_data)
    # Remove duplicates
    sorted_data = np.unique(sorted_data)

    # Find midpoints between adjacent values
    midpoints = []
    for i in range(len(sorted_data) - 1):
        midpoint = (sorted_data[i] + sorted_data[i+1]) / 2.0
        midpoints.append( midpoint )

    return midpoints


# Class for decision nodes
class Node:

    # Init function
    def __init__(self, x, y, attribute_types):
        self.x = x
        self.y = y
        self.attribute_types = attribute_types
        self.left = None
        self.right = None
        self.regress_value = None     # If the leaf node

        self.decision_value = None
        self.decision_type = None
        self.decision_attribute = None

    # Calculates the error on a node
    def calc_error(self):
        mean = np.mean(self.y)
        sum_err = 0
        for r in self.y:
```

```
            sum_err += (r - mean)**2
        error = sum_err / len(self.y)
        return error

    # Find all split points on the data
    def find_split_pts(self):
        split_pts = list()

        # Traverse all attributes
        for (i,attribute) in enumerate(self.attribute_types):

            # Numeric
            if(attribute == 1):
                mid_pts = get_midpoints(self.x[:,i])
                split_pts += [(i,x) for x in mid_pts]

            # Categoric
            elif(attribute == 2):
                uniques = np.unique(self.x[:,i])

                # Binary class --> just add one of them
                if(len(uniques) == 2):
                    split_pts.append( (i, uniques[0]) )

                # Multiclass --> add each one
                elif(len(uniques) > 2):
                    split_pts +=  [ (i,x) for x in uniques ]

                # If It has only one unique class for this data then do not add it

            # Invalid attribute type
            else:
                raise TypeError("Attribute type is not valid. Only 1 and 2 is vali

        return  split_pts


    # Generate left and right nodes with given split point
    def generate_nodes(self, split_pt):
        decision_type = self.attribute_types[split_pt[0]]
        decision_value = split_pt[1]

        # Numeric
        if(decision_type == 1):
            mask = self.x[:,split_pt[0]] < decision_value

        # Categoric
        elif(decision_type == 2):
            mask = self.x[:,split_pt[0]] == decision_value

        # Invalid attribute type
        else:
            raise TypeError("Attribute type is not valid. Only 1 and 2 is valid ")
```

```
            # if(len(self.x[mask]) == 0):
            #     raise "split error"
            # if(len(self.x[~mask]) == 0):
            #     raise "split error"


            left = Node(self.x[mask], self.y[mask], self.attribute_types)
            right = Node(self.x[~mask], self.y[~mask], self.attribute_types)

            return (left, right)


        # Calculate the split error by using left and right nodes.
        # Explanation: Calculate the error of both left and right nodes and normalize
        def calc_split_error(self, node_left:'Node', node_right:'Node'):

            # Sizes of nodes
            left_size = float( len(node_left.x) )
            right_size = float( len(node_right.x) )
            total = left_size + right_size

            # Normalization
            split_error = left_size / total * node_left.calc_error() + right_size / to

            return split_error


        # Predict the given data
        # Go until leaf node recursively and return regress value
        def predict(self, data):

            if(self.regress_value != None):
                return self.regress_value

            # Numeric
            if(self.decision_type == 1):
               if( data[self.decision_attribute] < self.decision_value):
                   return self.left.predict(data)

               else:
                   return self.right.predict(data)

            # Categoric
            elif(self.decision_type == 2):
                if( data[self.decision_attribute] == self.decision_value):
                   return self.left.predict(data)

                else:
                    return self.right.predict(data)


            # Invalid attribute type
            else:
                raise TypeError("Attribute type is not valid. Only 1 and 2 is valid ")
```

```
# Recursive and entropy based DT generation algorithm  by using greedy algorithm
# Take data included root node and generate the DT recursively
# Arg "error_limit" : limits the DT with a error rate. If reach this error rate th
def generate_tree(node:Node , max_depth, error_limit):

    # If max depth is reached, then label the leaf node with mean value and return
    if(max_depth <= 0):
        node.regress_value = np.mean(node.y)
        return

    # If the error of the current node is 0 then no need to continue anymore
    # Label the leaf node with mean value and return
    if(node.calc_error() <= error_limit):
        node.regress_value = np.mean(node.y)
        return

    # Find all split points
    split_points = node.find_split_pts()

    # Variables to hold best split
    best_split = None
    least_error = float('inf')
    nodes = None

    # Calculate errors of all split points and get the best split
    for split_pt in split_points:

        # Generate the child nodes with split
        node_left, node_right = node.generate_nodes(split_pt)

        # Calculate error
        error = node.calc_split_error(node_left, node_right)

        if( error < least_error ):
            least_error = error
            best_split = split_pt
            nodes = (node_left, node_right)


    # End of For : Best split found.

    # Place the children to the left and right
    node.left, node.right = nodes[0], nodes[1]

    # Place the decision value and data type on the node
    node.decision_value = best_split[1]
    node.decision_type = node.attribute_types[best_split[0]]
    node.decision_attribute = best_split[0]

    # Recursive call for children
    generate_tree(node.left, max_depth-1, error_limit)
    generate_tree(node.right, max_depth-1, error_limit)
```

```
# DT regressor builder
def buid_rdf(X, y, attribute_types, N, error_limit):
    root = Node(X,y,attribute_types)
    generate_tree(root, N, error_limit)
    return root
```

```
# Takes DT and X matrix returns a vector for predicted predicted labels
def predict_rdf(dt:Node, X):
    predict_vector = [dt.predict(x) for x in X]
    return np.array( predict_vector )
```

```
# Load dataset
df1 = pd.read_csv("day.csv")
```

```
# Split x and y
Y = df1["cnt"].values
X = df1.drop("cnt",axis=1)
X = X.drop("dteday",axis=1)
X = X.drop("instant",axis=1)

X = X.values

attribute_types = [2,2,2,2,2,2,2,1,1,1,1,1,1]

from sklearn.model_selection import KFold
from sklearn.metrics import confusion_matrix, classification_report, r2_score

# K fold
k_fold = KFold(n_splits=6, shuffle=True, random_state=42)
```

Results:

```
# Mean for R^2 score
mean = 0

for k, (train, test) in enumerate(k_fold.split(X, Y)):
  print("\n\nFold",k,"Result:")
  # Train
  dt = buid_rdf(X[train], Y[train], attribute_types, N=5, error_limit=0)
  y_pred = predict_rdf(dt, X[test])

  # Evaluate the model using R^2 score
  r2 = r2_score(Y[test], y_pred)
```

```
    print("R^2 score: {:.2f}".format(r2))
    mean += r2

  mean = mean / 6
  print("\nMean R^2 score:", mean)
```

```
Fold 0 Result:
R^2 score: 0.98


Fold 1 Result:
R^2 score: 0.97


Fold 2 Result:
R^2 score: 0.97


Fold 3 Result:
R^2 score: 0.98


Fold 4 Result:
R^2 score: 0.97


Fold 5 Result:
R^2 score: 0.97

Mean R^2 score: 0.9715394786157862
```

Comments and discussion:

For this part, the basic idea of generating DT is same. There are some differences like calc_error calc_split_error etc. For example, Instead of calc_entropy function i use calc_error function, it calculates average error as told in the class. And also there are some little changes in the generate_tree function. It uses one more parameter error_limit. This parameter can be used ignoring some error while training. For example if the error_limit is setted to 5, then if a node is less than 5. The node will not continue to generate any more child nodes. This will decrease the overfitting and increase the run time performance. When it reaches leaf nodes it labels the regress value to the leaf node. While predicting a value it goes recursively until reaching a leaf node.

I wrote comment lines. You may follow them for better understanding of the codes. I implemented all formulas, algorithms in the lectures. I tried to do best. Thanks

✓ 0s completed at 9:09 AM