code

September 3, 2023

0.1 Assignment 1

0.1.1 Name: Bannuru Rohit Kumar Reddy

0.1.2 Roll Number: 21CS30011

[204]:
```
# import all the necessary libraries here
import pandas as pd

import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from graphviz import Digraph
```

[205]:
```
df = pd.read_csv('../../dataset/decision-tree.csv')
print(df.shape)
```

```
(768, 9)
```

Understanding the type of Dataset

[206]:
```
# Analyzing the data before we proceed further :

print(df.dtypes)
print(df.describe())
print(df.isnull().sum())
```

```
Pregnancies                   int64
Glucose                       int64
BloodPressure                 int64
SkinThickness                 int64
Insulin                       int64
BMI                         float64
DiabetesPedigreeFunction    float64
Age                           int64
Outcome                       int64
dtype: object
       Pregnancies     Glucose  BloodPressure  SkinThickness     Insulin  \
count   768.000000  768.000000     768.000000     768.000000  768.000000
mean      3.845052  120.894531      69.105469      20.536458   79.799479
```

```
std       3.369578   31.972618     19.355807     15.952218  115.244002
min       0.000000    0.000000      0.000000      0.000000    0.000000
25%       1.000000   99.000000     62.000000      0.000000    0.000000
50%       3.000000  117.000000     72.000000     23.000000   30.500000
75%       6.000000  140.250000     80.000000     32.000000  127.250000
max      17.000000  199.000000    122.000000     99.000000  846.000000

              BMI  DiabetesPedigreeFunction         Age     Outcome
count  768.000000                768.000000  768.000000  768.000000
mean    31.992578                  0.471876   33.240885    0.348958
std      7.884160                  0.331329   11.760232    0.476951
min      0.000000                  0.078000   21.000000    0.000000
25%     27.300000                  0.243750   24.000000    0.000000
50%     32.000000                  0.372500   29.000000    0.000000
75%     36.600000                  0.626250   41.000000    1.000000
max     67.100000                  2.420000   81.000000    1.000000
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

Splitting the dataset into training,test and validation set

[207]:
```
# Since there are no NULL values, let us proceed with normalizing the dataset␣
↪and then split it in the 80 - 20 Fashion

# split the dataset into train and test and validation here
train_df, temp_data_df = train_test_split(df, test_size=0.5, random_state=39)
validation_df, test_df = train_test_split(temp_data_df, test_size=0.4,␣
↪random_state=39)

# Convert DataFrame to numpy arrays
train_data = train_df.to_numpy()
test_data = test_df.to_numpy()
validation_data = validation_df.to_numpy()

# Split data into input and output features
X_train = train_data[:, :-1]
X_test = test_data[:, :-1]
X_validation = validation_data[:, :-1]
y_train = train_data[:, -1]
y_test = test_data[:, -1]
```

```
y_validation = validation_data[:, -1]
```

Implementing ID3 Decision Tree

Steps Involved :

1. Calculating Enotrpy
2. Find the information gain based on the entropy
3. Find the best attribute based on the information gained function, This must return the best attribute along with the position at which the attribute must be split to get the tree

[208]:
```
def entropy(y):
    total_entropy = 0
    unique_classes, class_counts = np.unique(y, return_counts=True)
    for count in class_counts:
        probability = count / len(y)
        total_entropy -= probability * np.log2(probability)
    return total_entropy

def info_gain(X, y, attribute_index, threshold):
    initial_entropy = entropy(y)
    entropy_after_split = 0
    y_left = []
    y_right = []
    for i in range(len(y)):
        if X[i][attribute_index] <= threshold:
            y_left.append(y[i])
        else:
            y_right.append(y[i])
    entropy_after_split = (len(y_left) / len(y)) * entropy(y_left) +
 ↪(len(y_right) / len(y)) * entropy(y_right)
    return initial_entropy - entropy_after_split
```

[209]:
```
# Finding the best attribute and threshold value
def best_attribute_threshold(X, y):

    best_attribute_index = 0
    best_threshold = 0
    max_info_gain = 0
    for i in range(X.shape[1]):
        for j in range(X.shape[0]):
            info_gain_val = info_gain(X, y, i, X[j][i])
            if info_gain_val > max_info_gain:
                max_info_gain = info_gain_val
                best_attribute_index = i
                best_threshold = X[j][i]
    return (best_attribute_index, best_threshold)
```

Creating the base of the tree in terms of Nodes

1. Define Node
2. Build Tree based on the nodes

[210]:
```
# Node class for the decision tree
class Node:
    def __init__(self, attribute_index=None, threshold=None, left=None,␣
↪right=None, label=None):
        self.attribute_index = attribute_index
        self.threshold = threshold
        self.left = left
        self.right = right
        self.label = label

    def is_leaf_node(self):
        return self.label is not None


# Building the decision tree using the ID3 algorithm
# tree having  min_size as stopping criteria

def build_tree(X, y, min_size):

    if len(y) <= min_size:
        unique, counts = np.unique(y, return_counts=True)
        return Node(label=unique[np.argmax(counts)])
    best_attribute_index, best_threshold = best_attribute_threshold(X, y)
    y_left = []
    y_right = []
    X_left = []
    X_right = []
    for i in range(len(y)):
        if X[i][best_attribute_index] <= best_threshold:
            y_left.append(y[i])
            X_left.append(X[i])
        else:
            y_right.append(y[i])
            X_right.append(X[i])
    if len(y_left) == 0 or len(y_right) == 0:
        unique, counts = np.unique(y, return_counts=True)
        return Node(label=unique[np.argmax(counts)])
    left = build_tree(np.array(X_left), np.array(y_left), min_size)
    right = build_tree(np.array(X_right), np.array(y_right), min_size)
    return Node(best_attribute_index, best_threshold, left, right)
```

Utility Functions :

[211]:
```python
def predict(node, data_point):
    if node.is_leaf_node():
        return node.label
    if data_point[node.attribute_index] <= node.threshold:
        return predict(node.left, data_point)
    else:
        return predict(node.right, data_point)

def predict_labels(root, X):
    y_pred = []
    for i in range(len(X)):
        y_pred.append(predict(root, X[i]))
    return np.array(y_pred)

def accuracy(y_pred, y_true):
    correct = 0
    for i in range(len(y_pred)):
        if y_pred[i] == y_true[i]:
            correct += 1
    return correct/len(y_pred)
```

0.1.3 Visualize the tree

[212]:
```python
def visualize_decision_tree(node, feature_names, graph=None):
    if graph is None:
        graph = Digraph(format='png')  # You can change the format if you␣
 ↪prefer a different image format

    if node.is_leaf():
        graph.node(str(id(node)), label=str(node.output_label))
    else:
        feature_name = feature_names[node.split_attribute]
        graph.node(str(id(node)), label=f"{feature_name}\nThreshold {node.
 ↪split_threshold}")
        if node.left:
            visualize_decision_tree(node.left, feature_names, graph)
            graph.edge(str(id(node)), str(id(node.left)), label='correct')
        if node.right:
            visualize_decision_tree(node.right, feature_names, graph)
            graph.edge(str(id(node)), str(id(node.right)), label='incorrect')

    return graph
```

0.2 Prune the tree

[213]:
```
# Pruning:
def prune_decision_tree(node, validation_data, validation_labels):
    if node.is_leaf():
        return node

    if node.left.is_leaf() and node.right.is_leaf():
        # Calculate accuracy before pruning
        predicted_labels_before_pruning = predict_labels(node, validation_data)
        accuracy_before = compute_accuracy(predicted_labels_before_pruning,␣
↪validation_labels)

        # Prune the node by setting it as a leaf with the majority class
        unique_labels, label_counts = np.unique(validation_labels,␣
↪return_counts=True)
        most_common_label = unique_labels[np.argmax(label_counts)]
        node.set_as_leaf(most_common_label)

        # Calculate accuracy after pruning
        predicted_labels_after_pruning = predict_labels(node, validation_data)
        accuracy_after = compute_accuracy(predicted_labels_after_pruning,␣
↪validation_labels)

        # If accuracy doesn't improve, revert the pruning
        if accuracy_after < accuracy_before:
            node.revert_pruning()

        return node

    # Recursively prune the left and right subtrees
    node.left = prune_decision_tree(node.left, validation_data,␣
↪validation_labels)
    node.right = prune_decision_tree(node.right, validation_data,␣
↪validation_labels)

    return node
```

1 Build the main tree and the pruned tree

[214]:
```
base_decision_tree = build_tree(X_train, y_train, 10)

# Names of the features
feature_names = list(df.columns[:-1])
```

```python
graph = visualize_tree(base_decision_tree, feature_names)
graph.render('decision_tree', view=True)
y_pred = predict_labels(base_decision_tree, X_test)
test_accuracy = accuracy(y_pred, y_test)


# Prune the tree and repeat the same thing
pruned_tree = reduced_error_pruning(base_decision_tree, X_validation,␣
↪y_validation)
graph = visualize_tree(pruned_tree, feature_names)
graph.render('pruned_decision_tree', view=True)
y_pred = predict_labels(pruned_tree, X_test)
test_accuracy = accuracy(y_pred, y_test)

print(f"Test accuracy: {test_accuracy}")
print(f"Test accuracy pruning: {test_accuracy}")
```

```
Test accuracy: 0.7337662337662337
Test accuracy pruning: 0.7337662337662337
```