Decision_Tree_Implementation_Python

April 7, 2024

1 Decision Tree Classifier Implementation in Python using Numpy:

1.1 Importing tools

[ ]:
```
import numpy as np
import pandas as pd
```

1.2 Loading data

[ ]:
```
col_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',␣
 ↪'type']
data = pd.read_csv("Iris.csv", skiprows=1, header=None, names=col_names)
data.head(10)
```

[ ]:

	sepal_length	sepal_width	petal_length	petal_width	type
1	5.1	3.5	1.4	0.2	Iris-setosa
2	4.9	3.0	1.4	0.2	Iris-setosa
3	4.7	3.2	1.3	0.2	Iris-setosa
4	4.6	3.1	1.5	0.2	Iris-setosa
5	5.0	3.6	1.4	0.2	Iris-setosa
6	5.4	3.9	1.7	0.4	Iris-setosa
7	4.6	3.4	1.4	0.3	Iris-setosa
8	5.0	3.4	1.5	0.2	Iris-setosa
9	4.4	2.9	1.4	0.2	Iris-setosa
10	4.9	3.1	1.5	0.1	Iris-setosa

1.3 Encoding target variable

[ ]:
```
from sklearn.preprocessing import LabelEncoder
lc = LabelEncoder()
data['type'] = lc.fit_transform(data['type'])
data.head()
```

[ ]:

	sepal_length	sepal_width	petal_length	petal_width	type
1	5.1	3.5	1.4	0.2	0
2	4.9	3.0	1.4	0.2	0

3	4.7	3.2	1.3	0.2	0
4	4.6	3.1	1.5	0.2	0
5	5.0	3.6	1.4	0.2	0

[ ]:
```
X = data.iloc[:, :-1].values
Y = data.iloc[:, -1].values.reshape(-1,1)
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=.2,␣
 ↪random_state=41)
```

1.4 View the variables

Let's get more familiar with the dataset.
- A good place to start is to just print out each variable and see what it contains.

The code below prints the first few elements of `X_train` and the type of the variable.

[ ]:
```
print("First few elements of X_train:\n", X_train[:5])
print("Type of X_train:",type(X_train))
```

```
First few elements of X_train:
 [[5.7 2.6 3.5 1. ]
 [6.5 2.8 4.6 1.5]
 [4.9 2.5 4.5 1.7]
 [5.5 2.6 4.4 1.2]
 [6.7 3.  5.2 2.3]]
Type of X_train: <class 'numpy.ndarray'>
```

1.5 Checking the dimension of variables

Another useful way to get familiar with your data is to view its dimensions.

Please print the shape of `X_train` and `y_train` and see how many training examples you have in your dataset.

[ ]:
```
print ('The shape of X_train is:', X_train.shape)
print ('The shape of y_train is: ', Y_train.shape)
print ('Number of training examples (m):', len(X_train))
```

```
The shape of X_train is: (120, 4)
The shape of y_train is:  (120, 1)
Number of training examples (m): 120
```

1.6 Decision Tree Refresher

Decision trees are a fundamental machine learning model used for both classification and regression tasks. They work by recursively partitioning the feature space into regions, with the goal of making predictions based on the majority class (for classification) or mean value (for regression) within each region. Here's a breakdown of key concepts:

1. **Splitting Criterion**: At each node of the tree, a decision is made to split the data based on a chosen feature and threshold value. The objective is to maximize the information gain or minimize impurity in the resulting subsets.
2. **Information Gain**: Measures the reduction in entropy (for classification) or variance (for regression) achieved by splitting the data at a particular node. Higher information gain indicates a more effective split.
3. **Leaf Nodes**: Terminal nodes of the decision tree that represent the final prediction for a given subset of the data. For classification, the majority class in the leaf node is used as the prediction.

1.7 DecisionTreeClassifier Implementation Details

Below are the details of my entire approach

1.7.1 Components:

1. **Node Class** (`Node`):
   - Represents a node in the decision tree.
   - Attributes:
     - `feature_index`: Index of the feature used for splitting at this node.
     - `threshold`: Threshold value for the feature used in the splitting decision.
     - `left`: Pointer to the left child node.
     - `right`: Pointer to the right child node.
     - `info_gain`: Information gain achieved by the split at this node (for decision nodes).
     - `value`: Predicted value (class label or regression target) for leaf nodes.

[ ]:
```
from typing import List, Tuple, Set, Dict, Optional, Any
```

[ ]:
```
class Node:
    """A class representing a node in a decision tree.

    Attributes:
        feature_index (Optional[int]): The index of the feature used for␣
↪splitting at this node.
        threshold (Optional[float]): The threshold value for the feature used␣
↪in the splitting decision.
        left (Optional[Node]): The left child node.
        right (Optional[Node]): The right child node.
        info_gain (Optional[float]): The information gain achieved by the split␣
↪at this node (for decision nodes).
        value (Optional[int]): The predicted value for leaf nodes (e.g., class␣
↪label for classification).

    Note:
        - For decision nodes (non-leaf), `feature_index` and `threshold`␣
↪determine the splitting condition.
```

```
        - `left` and `right` are pointers to the left and right child nodes␣
↪resulting from the split.
        - `info_gain` represents the information gain achieved by the split.
        - For leaf nodes, `value` holds the predicted value (e.g., class label)␣
↪based on majority voting.

    """

    def __init__(
        self,
        feature_index: Optional[int] = None,
        threshold: Optional[float] = None,
        left: Optional['Node'] = None,
        right: Optional['Node'] = None,
        info_gain: Optional[float] = None,
        value: Optional[int] = None,
    ):
        """Initialize a Node object with specified attributes.

        Args:
            feature_index (Optional[int]): The index of the feature used for␣
↪splitting.
            threshold (Optional[float]): The threshold value for the feature␣
↪used in splitting.
            left (Optional[Node]): The left child node.
            right (Optional[Node]): The right child node.
            info_gain (Optional[float]): The information gain achieved by the␣
↪split.
            value (Optional[int]): The predicted value for leaf nodes.

        """
        self.feature_index = feature_index
        self.threshold = threshold
        self.left = left
        self.right = right
        self.info_gain = info_gain
        self.value = value
```

2. **Decision Tree Classifier Class** (`DecisionTreeClassifier`):
    - Implements a decision tree classifier using a binary tree structure.

1.7.2 Methods:

1. **`fit(X, Y)`**:
    - Trains the decision tree classifier on the input training data (`X` features, `Y` labels).
    - Constructs the tree using the provided training dataset by recursively calling `build_tree`.

2. `build_tree(dataset)`

- Iterative function to build the decision tree using a stack-based approach.
- Inputs:
  - `dataset`: Training dataset represented as a NumPy array with features and labels.
- Steps:
  1. Initialize the root node of the tree.
  2. Use a stack to manage nodes and their corresponding datasets during tree construction.
  3. Push the initial state onto the stack with the entire dataset and the root node.
  4. While the stack is not empty:
  - Pop a node and its associated dataset from the stack.
  - Extract features (`X`) and labels (`Y`) from the dataset.
  - Check stopping conditions (minimum samples or maximum depth):
    - If stopping conditions are met, compute and assign the predicted value (`value`) for the leaf node based on `Y`.
    - Otherwise, proceed with tree expansion:
      - Find the best split (`get_best_split`) based on information gain using the current dataset.
      - If a valid split is found (positive information gain), create decision nodes (`Node`) for the current feature and threshold.
      - Partition the dataset into left and right subsets based on the best split.
      - Push the left and right child nodes along with their respective datasets onto the stack for further processing.

3. `get_best_split(dataset, num_samples, num_features)`:
   - Finds the best feature and threshold to split the dataset based on maximum information gain.
   - Loops through each feature and possible thresholds to calculate information gain for potential splits.
   - Returns a dictionary containing the best split information.
4. `split(dataset, feature_index, threshold)`:
   - Splits the dataset into left and right subsets based on a given feature and threshold.
5. `information_gain(parent, l_child, r_child, mode="gini")`:
   - Computes information gain based on the impurity measure (Gini index or entropy).
6. `calculate_leaf_value(Y)`:
   - Determines the predicted value for a leaf node based on the majority class (classification) or mean value (regression) of the labels `Y`.
7. `print_tree(tree=None, indent=" ")`:
   - Recursively prints the structure of the decision tree for visualization purposes.
8. `predict(X)`:
   - Makes predictions for input data `X` using the trained decision tree.
   - Calls `make_prediction` for each data point in `X`.
9. `make_prediction(x, tree)`:
   - Recursively traverses the decision tree to predict the label for a single data point `x`.
   - Handles cases where the tree traversal reaches leaf nodes (`tree.value` is not `None`) or decision nodes (based on feature and threshold).

1.7.3 Mathematical Equations

Entropy Calculation ($H(S)$) The entropy is a measure of impurity or randomness in a dataset. For a classification problem with classes $(C_1, C_2, \ldots, C_k)$, the entropy $H(S)$ of a set (S) with class labels (y) is calculated as:

$$H(S) = -\sum_{i=1}^{k} p_i \log_2(p_i)$$

Where:

- p_i is the proportion of examples in class C_i in the dataset (S).
- log_2 denotes the logarithm base 2.

In code:

```
def entropy(y):
    class_labels = np.unique(y)
    entropy = 0
    for cls in class_labels:
        p_cls = len(y[y == cls]) / len(y)
        entropy += -p_cls * np.log2(p_cls)
    return entropy
```

Gini Index Calculation ($Gini(S)$) The Gini index is another measure of impurity often used in decision trees. It measures the probability that a randomly chosen element from the set would be incorrectly classified if it was randomly labeled according to the distribution of labels in the set.

$$Gini(S) = 1 - \sum_{i=1}^{k} p_i^2$$

Where:

- p_i is the proportion of examples in class C_i in the dataset S.

In code:

```
def gini_index(y):
    class_labels = np.unique(y)
    gini = 0
    for cls in class_labels:
        p_cls = len(y[y == cls]) / len(y)
        gini += p_cls**2
    return 1 - gini
```

Information Gain Calculation ($IG(parent, l\_child, r\_child)$) Information gain measures the reduction in entropy (or increase in purity) achieved by splitting a dataset S into subsets S_{left} and S_{right}

$$\text{Information Gain} = \text{Impurity}(S) - \left(\frac{|S_{left}|}{|S|} \times \text{Impurity}(S_{left}) + \frac{|S_{right}|}{|S|} \times \text{Impurity}(S_{right}) \right)$$

Where:

- $\text{Impurity}(S)$ is the impurity measure (entropy or Gini index) of the dataset S.
- $|S|$ is the total number of examples in dataset (S).
- $|S_{left}|$ and $|S_{right}|$ are the number of examples in the left and right subsets after splitting.

In code:

```
def information_gain(parent, l_child, r_child, mode="gini"):
    weight_l = len(l_child) / len(parent)
    weight_r = len(r_child) / len(parent)

    if mode == "gini":
        impurity_parent = gini_index(parent)
        impurity_l = gini_index(l_child)
        impurity_r = gini_index(r_child)
    else:  # mode == "entropy"
        impurity_parent = entropy(parent)
        impurity_l = entropy(l_child)
        impurity_r = entropy(r_child)

    gain = impurity_parent - (weight_l * impurity_l + weight_r * impurity_r)
    return gain
```

Accuracy Calculation (accuracy$(Y_{\text{true}}, Y_{\text{pred}})$) The accuracy score measures the proportion of correctly predicted labels compared to the true labels in a dataset.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

In code (using `sklearn.metrics.accuracy_score`):

```
from sklearn.metrics import accuracy_score

# Example usage
Y_true = [true_label1, true_label2, ...]
Y_pred = [predicted_label1, predicted_label2, ...]
accuracy = accuracy_score(Y_true, Y_pred)
```

1.7.4 Implementation

[ ]:
```
class DecisionTreeClassifier:
    """A decision tree classifier implementation using an iterative build␣
    ↪approach.
```

```
    Attributes:
        root (Optional[Node]): The root node of the decision tree.
        min_samples_split (int): The minimum number of samples required to␣
↪split a node.
        max_depth (int): The maximum depth of the decision tree.

    """

    def __init__(self, min_samples_split: int = 2, max_depth: int = 2):
        """Initialize the DecisionTreeClassifier.

        Args:
            min_samples_split (int): The minimum number of samples required to␣
↪split a node.
            max_depth (int): The maximum depth of the decision tree.

        """
        self.root: Optional[Node] = None
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth

    def build_tree(self, dataset: np.ndarray) -> Node:
        """Iteratively build the decision tree.

        Args:
            dataset (np.ndarray): The training dataset (features + labels).

        Returns:
            Node: The root node of the constructed decision tree.

        """
        self.root = Node()
        stack: List[Tuple[np.ndarray, Node, int]] = [(dataset, self.root, 0)]

        while stack:
            data, node, depth = stack.pop()
            X, Y = data[:, :-1], data[:, -1]
            num_samples, num_features = np.shape(X)

            if num_samples < self.min_samples_split or depth > self.max_depth:
                node.value = self.calculate_leaf_value(Y)
            else:
                best_split = self.get_best_split(data, num_samples,␣
↪num_features)

                if best_split["info_gain"] > 0:
```

```
                    node.feature_index = best_split["feature_index"]
                    node.threshold = best_split["threshold"]
                    node.info_gain = best_split["info_gain"]

                    left_subtree = Node()
                    node.left = left_subtree
                    stack.append((best_split["dataset_left"], left_subtree,␣
↪depth + 1))

                    right_subtree = Node()
                    node.right = right_subtree
                    stack.append((best_split["dataset_right"], right_subtree,␣
↪depth + 1))
                else:
                    node.value = self.calculate_leaf_value(Y)

        return self.root

    def get_best_split(self, dataset: np.ndarray, num_samples: int,␣
↪num_features: int) -> dict:
        """Find the best split based on information gain.

        Args:
            dataset (np.ndarray): The dataset to find the best split.
            num_samples (int): The number of samples in the dataset.
            num_features (int): The number of features in the dataset.

        Returns:
            dict: A dictionary containing the best split information.

        """
        best_split = {}
        max_info_gain = -float("inf")

        for feature_index in range(num_features):
            feature_values = dataset[:, feature_index]
            possible_thresholds = np.unique(feature_values)

            for threshold in possible_thresholds:
                dataset_left, dataset_right = self.split(dataset,␣
↪feature_index, threshold)

                if len(dataset_left) > 0 and len(dataset_right) > 0:
                    y, left_y, right_y = dataset[:, -1], dataset_left[:, -1],␣
↪dataset_right[:, -1]
                    curr_info_gain = self.information_gain(y, left_y, right_y,␣
↪"gini")
```

```
                    if curr_info_gain > max_info_gain:
                        best_split["feature_index"] = feature_index
                        best_split["threshold"] = threshold
                        best_split["dataset_left"] = dataset_left
                        best_split["dataset_right"] = dataset_right
                        best_split["info_gain"] = curr_info_gain
                        max_info_gain = curr_info_gain

        return best_split

    def split(self, dataset: np.ndarray, feature_index: int, threshold: float)␣
↪-> Tuple[np.ndarray, np.ndarray]:
        """Split the dataset based on a given feature and threshold.

        Args:
            dataset (np.ndarray): The dataset to split.
            feature_index (int): The index of the feature to use for splitting.
            threshold (float): The threshold value for splitting the feature.

        Returns:
            Tuple[np.ndarray, np.ndarray]: A tuple containing the left and␣
↪right split datasets.

        """
        dataset_left = dataset[dataset[:, feature_index] <= threshold]
        dataset_right = dataset[dataset[:, feature_index] > threshold]
        return dataset_left, dataset_right

    def information_gain(self, parent: np.ndarray, l_child: np.ndarray, r_child:
↪ np.ndarray, mode: str = "entropy") -> float:
        """Compute the information gain based on Gini impurity or entropy.

        Args:
            parent (np.ndarray): The labels of the parent node.
            l_child (np.ndarray): The labels of the left child node.
            r_child (np.ndarray): The labels of the right child node.
            mode (str): The impurity measure to use ('entropy' or 'gini').

        Returns:
            float: The computed information gain.

        """
        weight_l = len(l_child) / len(parent)
        weight_r = len(r_child) / len(parent)

        if mode == "gini":
```

```
            gain = self.gini_index(parent) - (weight_l * self.
↪gini_index(l_child) + weight_r * self.gini_index(r_child))
        else:
            gain = self.entropy(parent) - (weight_l * self.entropy(l_child) +␣
↪weight_r * self.entropy(r_child))

        return gain

    def entropy(self, y: np.ndarray) -> float:
        """Compute the entropy for a given set of labels.

        Args:
            y (np.ndarray): The array of labels.

        Returns:
            float: The computed entropy.

        """
        class_labels, class_counts = np.unique(y, return_counts=True)
        entropy = -np.sum((class_counts / len(y)) * np.log2(class_counts /␣
↪len(y)))
        return entropy

    def gini_index(self, y: np.ndarray) -> float:
        """Compute the Gini index for a given set of labels.

        Args:
            y (np.ndarray): The array of labels.

        Returns:
            float: The computed Gini index.

        """
        class_labels, class_counts = np.unique(y, return_counts=True)
        gini = 1.0 - np.sum((class_counts / len(y)) ** 2)
        return gini

    def calculate_leaf_value(self, Y: np.ndarray) -> int:
        """Compute the leaf node value (predicted class label).

        Args:
            Y (np.ndarray): The array of labels.

        Returns:
            int: The predicted class label (majority class).

        """
```

```
        return np.argmax(np.bincount(Y.astype(int)))

    def fit(self, X: np.ndarray, Y: np.ndarray) -> None:
        """Train the decision tree classifier.

        Args:
            X (np.ndarray): The feature matrix.
            Y (np.ndarray): The target labels.

        """
        dataset = np.concatenate((X, Y.reshape(-1, 1)), axis=1)
        self.build_tree(dataset)

    def predict(self, X: np.ndarray) -> List[int]:
        """Predict the class labels for new input data.

        Args:
            X (np.ndarray): The input feature matrix.

        Returns:
            List[int]: The predicted class labels.

        """
        predictions = [self.make_prediction(x, self.root) for x in X]
        return predictions

    def make_prediction(self, x: np.ndarray, tree: Node) -> int:
        """Make a prediction for a single data point using the decision tree.

        Args:
            x (np.ndarray): The input data point (features).
            tree (Node): The root node of the decision tree.

        Returns:
            int: The predicted class label.

        """
        if tree.value is not None:
            return tree.value

        feature_val = x[tree.feature_index]

        if feature_val <= tree.threshold:
            return self.make_prediction(x, tree.left)
        else:
            return self.make_prediction(x, tree.right)
```

1.8 Fit the model

[ ]:
```
classifier = DecisionTreeClassifier(min_samples_split=3, max_depth=3)
classifier.fit(X_train,Y_train)
# classifier.print_tree()
```

1.9 Evaluating the model

[ ]:
```
Y_pred = classifier.predict(X_test)
from sklearn.metrics import accuracy_score
print(f"Accuracy: {round(accuracy_score(Y_test, Y_pred), 2)}")
```

```
Accuracy: 0.93
```

1.10 Conclusion

Understanding the mathematical foundations behind decision trees, including entropy, Gini index, information gain, and accuracy, is essential for building and interpreting machine learning models effectively. By using this approach into my `DecisionTreeClassifier` implementation, I was able to achieve an impressive accuracy of 93% on Iris dataset.