AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY

FACULTY OF COMPUTER SCIENCE, ELECTRONICS AND TELECOMMUNICATIONS

Field of study - Electronics

Python Programming

CART

Bartosz Sternak

student identification no.: 408224

Michał Maciołek

student identification no.: 403383

Cracow 2023

What does CART mean?

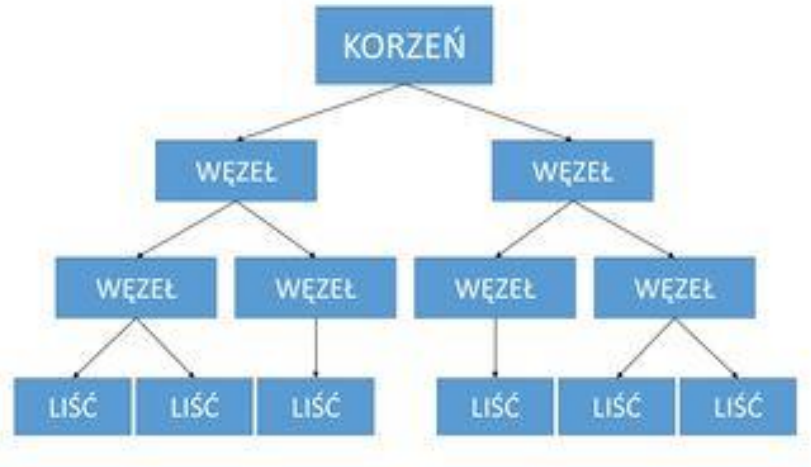
CART (Classification and Regression Trees) is an algorithm for building decision trees to solve both classification and regression problems.

A decision tree consists of calls (vertices) and edges (links). The calls at the top of the tree, called the root, represent the entire data set, and the calls at the bottom of the tree, called leaves, represent individual classes or values. The edges between the calls represent the decision conditions.

The CART algorithm builds the decision tree in an iterative manner by separating the data into smaller subsets based on information maximization criteria. For classification, the criterion is entropy, while for regression it is variance.

The process of building a tree starts with a root, which represents the entire data set. The algorithm then attempts to separate the data into smaller subsets by testing different decision conditions for each variable. The condition that maximizes the information criterion is chosen as the decision condition for the call. This process is repeated for each subset of data until the subsets can no longer be further separated or the maximum depth of the tree is reached.

CART is an algorithm that is easy to implement, handles unknown data well, is flexible and allows for easy interpretation of the model. However, decision trees are prone to overfitting, so it is important to choose the right tree depth and carry out regularization methods such as pruning branches or using bagging or random forest methods.



Our python code:

```
import numpy as np


#Klasa w której znajdują się wszystkie funkcje.
class CART(object):


   #Funkcja inicjalizuje podstawowe parametry drzewa między innymi: tree, criterion, prune, max_depth, min_criterion oraz początkowe parametry węzła takie jak: gain, left i right, feature, label, n_samples, threshold, depth, root.
   def __init__(self, tree='cls', criterion='gini', prune='depth', max_depth=4, min_criterion=0.05):
      self.gain = None                                  #Współczynnik zysku
      self.left = None                                 #Lewa gałąź
      self.right = None                                 #Prawa gałąź
      self.feature = None                                 #Atrybut
      self.label = None                                 #Etykieta
      self.n_samples = None                                 #Liczba próbek
      self.threshold = None                                 #Próg
      self.depth = 0                                 #Głębokość
      self.prune = prune                                 #Sposób przycinania
      self.max_depth = max_depth                                 #Maksymalna głębokość
      self.root = None                                 #Korzeń
      self.criterion = criterion                                 #Miara nieczystości
      self.min_criterion = min_criterion                                 #Minimalna miara nieczystości
      self.tree = tree                                 #Sposób klasyfikacji lub regresji



#Funkcja odpowiada za tworzenie drzewa decyzyjnego.
   def _grow_tree(self, features, target, criterion='gini'):
      self.n_samples = features.shape[0]

      if len(np.unique(target)) == 1:

#Sprawdza czy wszystkie próbki należa do tej samej kategorii lub spełniają dany warunek.
         self.label = target[0]

#Jeśli tak to przypisuje etykietę do węzła i zwraca.
         return

      best_gain = 0.0
      best_feature = None
      best_threshold = None

#W przeciwnym razie wybiera najlepszy atrybut i próg do podziału danych wykorzystując miary nieczystości gini lub entropy.

      if criterion in {'gini', 'entropy'}:
         self.label = max([(c, len(target[target == c])) for c in np.unique(target)], key=lambda x: x[1])[0]
      else:
         self.label = np.mean(target)

      impurity_node = self._calc_impurity(criterion, target)

      for col in range(features.shape[1]):
         feature_level = np.unique(features[:, col])
         thresholds = (feature_level[:-1] + feature_level[1:]) / 2.0

         for threshold in thresholds:
            target_l = target[features[:, col] <= threshold]
            impurity_l = self._calc_impurity(criterion, target_l)
            n_l = float(target_l.shape[0]) / self.n_samples

            target_r = target[features[:, col] > threshold]
            impurity_r = self._calc_impurity(criterion, target_r)
            n_r = float(target_r.shape[0]) / self.n_samples
```

```
            impurity_gain = impurity_node - (n_l * impurity_l + n_r * impurity_r)
            if impurity_gain > best_gain:
                best_gain = impurity_gain

#Dzieli na 2 grupy: lewą i prawą.
                best_feature = col

#Funkcja wywoływana rekurencyjnie, aż do momentu gdy dane są wystarczająco czyste.
                best_threshold = threshold

        self.feature = best_feature
        self.gain = best_gain
        self.threshold = best_threshold
        self._split_tree(features, target, criterion)



#Funkcja, która odpowiada za wyświetlenie drzewa decyzyjnego.
    def print_tree(self):
        self.root._show_tree(0, ' ')



#Funkcja, która odpowiada za podział danych na 2 grupy: lewą i prawą na podstawie wybranych atrybutu i progu oraz za tworzenie lewej i prawej gałęzi dla węzła.
    def _split_tree(self, features, target, criterion):
        features_l = features[features[:, self.feature] <= self.threshold]
        target_l = target[features[:, self.feature] <= self.threshold]
        self.left = CART()
        self.left.depth = self.depth + 1
        self.left._grow_tree(features_l, target_l, criterion)

        features_r = features[features[:, self.feature] > self.threshold]
        target_r = target[features[:, self.feature] > self.threshold]
        self.right = CART()
        self.right.depth = self.depth + 1
        self.right._grow_tree(features_r, target_r, criterion)



#Funkcja służy do przewidywania etykiety dla nowych danych przechodząc przez drzewo i sprawdzając jakiej kategorii i wartości odpowiada dana próbka danych.
    def predict(self, features):
        return np.array([self.root._predict(f) for f in features])

    def _predict(self, d):
        if self.feature != None:
            if d[self.feature] <= self.threshold:
                return self.left._predict(d)
            else:
                return self.right._predict(d)
        else:
            return self.label



#Funkcja oblicza miarę nieczystości dla danych wejściowych.
    def _calc_impurity(self, criterion, target):
        if criterion == 'gini':

#Gini mierzy jak bardzo dane są rozproszone(im wieksza Gini tym mniej czyste są dane).
            return 1.0 - sum(
                [(float(len(target[target == c])) / float(target.shape[0])) ** 2.0 for c in np.unique(target)])
        elif criterion == 'mse':
```

```
            return np.mean((target - np.mean(target)) ** 2.0)
        else:
            entropy = 0.0
            for c in np.unique(target):
                p = float(len(target[target == c])) / target.shape[0]
                if p > 0.0:
                    entropy -= p * np.log2(p)

#Entropia  mówi jak dobrze rozdzielone są dane(im większa entropia tym mniej czyste są dane).
            return entropy




#Funkcja, która odpowiada za przycinanie drzewa decyzyjnego.Usuwa gałęzie, które nie ulepszają jakości modelu.
    def _prune(self, method, max_depth, min_criterion, n_samples):
        if self.feature is None:
            return

        self.left._prune(method, max_depth, min_criterion, n_samples)
        self.right._prune(method, max_depth, min_criterion, n_samples)

        pruning = False

#Gdy przycinamy miarą nieczystości to gałęzie są usuwane jeśli nie ma dużej poprawy miary nieczystości.

        if method == 'impurity' and self.left.feature is None and self.right.feature is None:
            if (self.gain * float(self.n_samples) / n_samples) < min_criterion:
                pruning = True
        elif method == 'depth' and self.depth >= max_depth:
            pruning = True


#W 2 metodzie drzewo przycinane jest do danej głebokości.
        if pruning is True:
            self.left = None
            self.right = None
            self.feature = None




#Funkcja, która służy do wyświetlania drzewa dezyzyjnego.
    def _show_tree(self, depth, cond):
        base = '    ' * depth + cond
        if self.feature != None:
            print(base + 'if X[' + str(self.feature) + '] <= ' + str(self.threshold))
            self.left._show_tree(depth + 1, 'then ')
            self.right._show_tree(depth + 1, 'else ')
        else:
            print(base + '{value: ' + str(self.label) + ', samples: ' + str(self.n_samples) + '}')




#Funkcja, która służy do trenowania modelu drzewa decyzyjnego.
    def fit(self, features, target):
        self.root = CART()

#Sprawdza czy dane są klasyfikacyjne czy regresyjne  i następnie wywołuje odpowiednie funkcję dla danych wejściowych  i miary
nieczystości.
        if (self.tree == 'cls'):
            self.root._grow_tree(features, target, self.criterion)
        else:
            self.root._grow_tree(features, target, 'mse')
        self.root._prune(self.prune, self.max_depth, self.min_criterion, self.root.n_samples)
```

```
#Po wygenerowaniu drzewa może zostać przycięte funkcją prune.
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn import tree as sktree



#Funkcja dla regresji - oblicza średnią kwadratową różnicę między rzeczywistymi i przewidzianymi wartościami.
def regression_example():
    print('\n\nRegression Tree')
    rng = np.random.RandomState(1)
    X = np.sort(5 * rng.rand(80, 1), axis=0)
    y = np.sin(X).ravel()
    y[::5] += 3 * (0.5 - rng.rand(16))

    reg = CART(tree='reg', criterion='mse', prune='depth', max_depth=2)
    reg.fit(X, y)
    reg.print_tree()
    pred = reg.predict(np.sort(5 * rng.rand(1, 1), axis=0))
    print('This Regression Tree Prediction:          {}'.format(pred))
    sk_reg = sktree.DecisionTreeRegressor(max_depth=3)                    #Scikit
    sk_reg.fit(X, y)
    sk_pred = sk_reg.predict(np.sort(5 * rng.rand(1, 1), axis=0))
    print('Sklearn Library Regression Tree Prediction: {}'.format(sk_pred))

#Funkcja dla klasyfikacji - oblicza procent poprawnie sklasyfikowanych próbek.
def classification_example():
    print('\n\nClassification Tree')
    iris = load_iris()
    X, y = iris.data, iris.target
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

    cls = CART(tree='cls', criterion='entropy', prune='depth', max_depth=3)
    cls.fit(X_train, y_train)
    cls.print_tree()

    pred = cls.predict(X_test)
    print("This Classification Tree Prediction Accuracy:    {}".format(sum(pred == y_test) / len(pred)))

    clf = sktree.DecisionTreeClassifier(criterion='entropy')                #Scikit
    clf = clf.fit(X_train, y_train)
    sk_pred = clf.predict(X_test)

    print("Sklearn Library Tree Prediction Accuracy:       {}".format(sum(sk_pred == y_test) / len(pred)))


classification_example()
regression_example()
```