from sklearn.preprocessing import OneHotEncoder

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from collections import Counter

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, adjusted\_rand\_score

class DecisionTree:

def \_\_init\_\_(self, max\_depth=None, min\_samples\_split=2, criterion='entropy'):

self.max\_depth = max\_depth

self.min\_samples\_split = min\_samples\_split

self.criterion = criterion

self.tree = None

self.feature\_importances = None

def entropy(self, y):

counts = np.bincount(y)

probabilities = counts / len(y)

return -np.sum([p \* np.log2(p) for p in probabilities if p > 0])

# Считаем количество объектов для каждого класса. Формат - [0,0,1,2,1,2,0]

# суммируем вероятности. p - каждая итерация в полученном массиве 'probabilities'.

def gini(self, y):

counts = np.bincount(y)

probabilities = counts / len(y)

return 1 - np.sum(probabilities \*\* 2)

def information\_gain(self, y, left\_indices, right\_indices):

if self.criterion == 'entropy':

impurity\_func = self.entropy

else:

impurity\_func = self.gini

parent\_impurity = impurity\_func(y)

left\_impurity = impurity\_func(y[left\_indices])

right\_impurity = impurity\_func(y[right\_indices])

n, n\_left, n\_right = len(y), len(left\_indices), len(right\_indices)

weighted\_impurity = (n\_left / n) \* left\_impurity + (n\_right / n) \* right\_impurity

return parent\_impurity - weighted\_impurity

def custom\_1(self, y, left\_indices, right\_indices):

N = y\_oh.sum()

left = y\_oh[left\_indices]

right = y\_oh[right\_indices]

p\_1 = left.sum() / N

p\_2 = right.sum() / N

num\_classes = y\_oh.shape[1]

sum\_total = 0

epsilon = 1e-10

for l in range(num\_classes):

p\_1l = left[:, l].sum() / N

p\_2l = right[:, l].sum() / N

p\_l = p\_1l + p\_2l

b = 1

# eps. для стабильности вычислений

# denominator\_1 = max(p\_1 \* b\*\*2, epsilon)

# denominator\_2 = max(p\_2 \* b\*\*2, epsilon)

sum\_total += ((p\_1l - p\_1 \* p\_l)\*\*2) / p\_1 \* b\*\*2

sum\_total += ((p\_2l - p\_2 \* p\_l)\*\*2) / p\_2 \* b\*\*2

return N \* sum\_total

def custom\_2(self, y, left\_indices, right\_indices):

N = y\_oh.sum()

left = y\_oh[left\_indices]

right = y\_oh[right\_indices]

p\_1 = left.sum() / N

p\_2 = right.sum() / N

num\_classes = y\_oh.shape[1]

sum\_total = 0

epsilon = 1e-10

for l in range(num\_classes):

p\_1l = left[:, l].sum() / N

p\_2l = right[:, l].sum() / N

p\_l = p\_1l + p\_2l

b = np.sqrt(p\_l)

# eps. для стабильности вычислений

# denominator\_1 = max(p\_1 \* b\*\*2, epsilon)

# denominator\_2 = max(p\_2 \* b\*\*2, epsilon)

sum\_total += ((p\_1l - p\_1 \* p\_l)\*\*2) / p\_1 \* b\*\*2

sum\_total += ((p\_2l - p\_2 \* p\_l)\*\*2) / p\_2 \* b\*\*2

return N \* sum\_total

def custom\_3(self, y, left\_indices, right\_indices):

N = y\_oh.sum()

left = y\_oh[left\_indices]

right = y\_oh[right\_indices]

p\_1 = left.sum() / N

p\_2 = right.sum() / N

num\_classes = y\_oh.shape[1]

sum\_total = 0

epsilon = 1e-10

for l in range(num\_classes):

p\_1l = left[:, l].sum() / N

p\_2l = right[:, l].sum() / N

p\_l = p\_1l + p\_2l

b = np.sqrt(p\_l\*(1 - p\_l))

# eps. для стабильности вычислений

denominator\_1 = max(p\_1 \* b\*\*2, epsilon)

denominator\_2 = max(p\_2 \* b\*\*2, epsilon)

sum\_total += ((p\_1l - p\_1 \* p\_l)\*\*2) / denominator\_1

sum\_total += ((p\_2l - p\_2 \* p\_l)\*\*2) / denominator\_2

return N \* sum\_total

def custom\_4(self, y, left\_indices, right\_indices):

N = y\_oh.sum()

left = y\_oh[left\_indices]

right = y\_oh[right\_indices]

p\_1 = left.sum() / N

p\_2 = right.sum() / N

num\_classes = y\_oh.shape[1]

sum\_total = 0

epsilon = 1e-10

for l in range(num\_classes):

p\_1l = left[:, l].sum() / N

p\_2l = right[:, l].sum() / N

p\_l = p\_1l + p\_2l

b = p\_l

# eps. для стабильности вычислений

# denominator\_1 = max(p\_1 \* b\*\*2, epsilon)

# denominator\_2 = max(p\_2 \* b\*\*2, epsilon)

sum\_total += ((p\_1l - p\_1 \* p\_l)\*\*2) / p\_1 \* b\*\*2

sum\_total += ((p\_2l - p\_2 \* p\_l)\*\*2) / p\_2 \* b\*\*2

return N \* sum\_total

def custom\_5(self, y, left\_indices, right\_indices):

N = y\_oh.sum()

left = y\_oh[left\_indices]

right = y\_oh[right\_indices]

p\_1 = left.sum() / N

p\_2 = right.sum() / N

num\_classes = y\_oh.shape[1]

sum\_total = 0

epsilon = 1e-10

for l in range(num\_classes):

p\_1l = left[:, l].sum() / N

p\_2l = right[:, l].sum() / N

p\_l = p\_1l + p\_2l

b = p\_l\*\*2

# eps. для стабильности вычислений

# denominator\_1 = max(p\_1 \* b\*\*2, epsilon)

# denominator\_2 = max(p\_2 \* b\*\*2, epsilon)

sum\_total += ((p\_1l - p\_1 \* p\_l)\*\*2) / p\_1 \* b\*\*2

sum\_total += ((p\_2l - p\_2 \* p\_l)\*\*2) / p\_2 \* b\*\*2

return N \* sum\_total

def custom\_6(self, y, left\_indices, right\_indices):

N = y\_oh.sum()

left = y\_oh[left\_indices]

right = y\_oh[right\_indices]

p\_1 = left.sum() / N

p\_2 = right.sum() / N

num\_classes = y\_oh.shape[1]

sum\_total = 0

epsilon = 1e-10

for l in range(num\_classes):

p\_1l = left[:, l].sum() / N

p\_2l = right[:, l].sum() / N

p\_l = p\_1l + p\_2l

b = np.log(max(p\_l, epsilon))

# eps. для стабильности вычислений

denominator\_1 = max(p\_1 \* b\*\*2, epsilon)

denominator\_2 = max(p\_2 \* b\*\*2, epsilon)

sum\_total += ((p\_1l - p\_1 \* p\_l)\*\*2) / denominator\_1

sum\_total += ((p\_2l - p\_2 \* p\_l)\*\*2) / denominator\_2

return N \* sum\_total

def custom\_7(self, y, left\_indices, right\_indices):

N = y\_oh.sum()

left = y\_oh[left\_indices]

right = y\_oh[right\_indices]

p\_1 = left.sum() / N

p\_2 = right.sum() / N

num\_classes = y\_oh.shape[1]

sum\_total = 0

epsilon = 1e-10

for l in range(num\_classes):

p\_1l = left[:, l].sum() / N

p\_2l = right[:, l].sum() / N

p\_l = p\_1l + p\_2l

b = (-p\_l)\*np.log(max(p\_l, epsilon))

# eps. для стабильности вычислений

denominator\_1 = max(p\_1 \* b\*\*2, epsilon)

denominator\_2 = max(p\_2 \* b\*\*2, epsilon)

sum\_total += ((p\_1l - p\_1 \* p\_l)\*\*2) / denominator\_1

sum\_total += ((p\_2l - p\_2 \* p\_l)\*\*2) / denominator\_2

return N \* sum\_total

# Функция находит наиболее частый элемент в массиве y (метки классов).

def most\_common\_label(self, y):

return Counter(y).most\_common(1)[0][0]

def find\_best\_split(self, X, y, num\_features):

best\_gain = 0

best\_split = None

for feature\_index in range(num\_features):

thresholds = np.unique(X[:, feature\_index])

for threshold in thresholds:

left\_indices = np.where(X[:, feature\_index] <= threshold)[0]

right\_indices = np.where(X[:, feature\_index] > threshold)[0]

if len(left\_indices) == 0 or len(right\_indices) == 0:

continue

if self.criterion == 'custom\_1':

gain = self.custom\_1(y, left\_indices, right\_indices)

elif self.criterion == 'custom\_2':

gain = self.custom\_2(y, left\_indices, right\_indices)

elif self.criterion == 'custom\_3':

gain = self.custom\_3(y, left\_indices, right\_indices)

elif self.criterion == 'custom\_4':

gain = self.custom\_4(y, left\_indices, right\_indices)

elif self.criterion == 'custom\_5':

gain = self.custom\_5(y, left\_indices, right\_indices)

elif self.criterion == 'custom\_6':

gain = self.custom\_6(y, left\_indices, right\_indices)

elif self.criterion == 'custom\_7':

gain = self.custom\_7(y, left\_indices, right\_indices)

else:

gain = self.information\_gain(y, left\_indices, right\_indices)

if gain > best\_gain:

best\_gain = gain

best\_split = {

'feature\_index': feature\_index,

'threshold': threshold,

'left\_indices': left\_indices,

'right\_indices': right\_indices

}

return best\_split

def fit(self, X, y, y\_oh):

num\_features = X.shape[1]

self.feature\_importances = np.zeros(num\_features)

self.tree = self.grow\_tree(X, y, y\_oh, depth=0)

total = self.feature\_importances.sum()

if total > 0:

self.feature\_importances /= total

def grow\_tree(self, X, y, y\_oh, depth):

num\_samples, num\_features = X.shape

num\_classes = len(set(y))

if depth == self.max\_depth or num\_classes == 1 or num\_samples < self.min\_samples\_split:

return self.most\_common\_label(y)

if self.criterion == 'custom':

best\_split = self.find\_best\_split(X, y\_oh, num\_features)

else:

best\_split = self.find\_best\_split(X, y, num\_features)

if best\_split is None:

return self.most\_common\_label(y)

left\_indices, right\_indices = best\_split['left\_indices'], best\_split['right\_indices']

# Вычисляем прирост информации для подсчета важности признаков

if self.criterion == 'custom\_1':

gain = self.custom\_1(y, left\_indices, right\_indices)

elif self.criterion == 'custom\_2':

gain = self.custom\_2(y, left\_indices, right\_indices)

elif self.criterion == 'custom\_3':

gain = self.custom\_3(y, left\_indices, right\_indices)

elif self.criterion == 'custom\_4':

gain = self.custom\_4(y, left\_indices, right\_indices)

elif self.criterion == 'custom\_5':

gain = self.custom\_5(y, left\_indices, right\_indices)

elif self.criterion == 'custom\_6':

gain = self.custom\_6(y, left\_indices, right\_indices)

elif self.criterion == 'custom\_7':

gain = self.custom\_7(y, left\_indices, right\_indices)

else:

gain = self.information\_gain(y, left\_indices, right\_indices)

# Сохраняем вклад этого признака в важность

self.feature\_importances[best\_split['feature\_index']] += gain

left\_subtree = self.grow\_tree(X[left\_indices], y[left\_indices], y\_oh[left\_indices], depth + 1)

right\_subtree = self.grow\_tree(X[right\_indices], y[right\_indices], y\_oh[right\_indices], depth + 1)

return {

'feature\_index': best\_split['feature\_index'],

'threshold': best\_split['threshold'],

'left': left\_subtree,

'right': right\_subtree

}

def predict(self, X):

return np.array([self.\_traverse\_tree(x, self.tree) for x in X])

def \_traverse\_tree(self, x, node):

if isinstance(node, dict):

if x[node['feature\_index']] <= node['threshold']:

return self.\_traverse\_tree(x, node['left'])

else:

return self.\_traverse\_tree(x, node['right'])

return node

def plot\_tree(self, accuracy, precision, recall, f1, ari, importances, title=None, tree=None, feature\_names=None, class\_names=None):

if tree is None:

tree = self.tree

fig = plt.figure(figsize=(10, 5))

fig.patch.set\_edgecolor('red')

fig.patch.set\_linewidth(2)

fig.patch.set\_facecolor('white') # по желанию

self.\_plot\_subtree(tree, 0.5, 1.0, 0.5, 0.1, feature\_names, class\_names)

plt.axis("off")

# Adding the legend with metrics

plt.figtext(0.12, 0.01, f"Accuracy: {accuracy\*100:.2f}%\nPrecision: {precision\*100:.2f}%\nRecall: {recall\*100:.2f}%\nF1 Score: {f1\*100:.2f}%\nARI: {ari\*100:.2f}%",

ha="left", va="top", fontsize=10, bbox=dict(facecolor="white", edgecolor="black", boxstyle="round,pad=0.5"))

plt.figtext(0.90, 0.01, importances, ha="right", va="top", fontsize=10, bbox=dict(facecolor="white", edgecolor="black", boxstyle="round,pad=0.5"))

plt.title(title)

plt.show()

def \_plot\_subtree(self, node, x, y, dx, dy, feature\_names, class\_names):

if isinstance(node, dict):

# Node: Plot the decision

feature\_index = node["feature\_index"]

threshold = node["threshold"]

if feature\_names is not None:

feature\_name = feature\_names[feature\_index]

else:

feature\_name = f"Feature {feature\_index}"

node\_text = f"{feature\_name} <= {threshold:.2f}"

plt.text(x, y, node\_text, ha="center", va="center", bbox=dict(boxstyle="round", fc="white"))

# Plot left subtree

self.\_plot\_subtree(node["left"], x - dx, y - dy, dx / 2, dy, feature\_names, class\_names)

plt.plot([x, x - dx], [y, y - dy], 'k-')

# Plot right subtree

self.\_plot\_subtree(node["right"], x + dx, y - dy, dx / 2, dy, feature\_names, class\_names)

plt.plot([x, x + dx], [y, y - dy], 'k-')

else:

# Leaf: Plot the class label

if class\_names is not None:

leaf\_text = class\_names[node]

else:

leaf\_text = f"Class {node}"

plt.text(x, y, leaf\_text, ha="center", va="center", bbox=dict(boxstyle="round", fc="yellow"))

def plot\_feature\_importances(self, feature\_names=None):

if self.feature\_importances is None:

print("Модель не обучена. Нет важностей признаков.")

return

if feature\_names is None:

feature\_names = [f"Feature {i}" for i in range(len(self.feature\_importances))]

importances = pd.DataFrame({

'Feature': feature\_names,

'Importance': self.feature\_importances

}).sort\_values(by='Importance', ascending=False)

return importances.head(10)