```python
class Node:
    def __init__(self):
        self.feature = None
        self.threshold = None
        self.left = None
        self.right = None
        self.class_label = None
        self.is_leaf = False
```

کلاس Node(گره)

کلاس Node ساختار داده ای است که هر گره از درخت تصمیم را نمایش میدهد. این کلاس شامل 6 ویژگی اصلی است که مشخص میکند آیا گره یک برگ (پایانی) است یا یک گره تصمیم گیری که باید داده ها را تقسیم کند.

ویژگی های کلاس:

- feature: شماره ویژگی ای که برای تقسیم داده ها در این گره استفاده میشود. در گره های برگ مقدار None دارد.
- threshold: مقدار آستانه ای که داده ها بر اساس آن به دو دسته چپ و راست تقسیم میشوند. اگر مقدار ویژگی کمتر یا مساوی آستانه باشد، به سمت چپ می رود و در غیر این صورت به سمت راست.
- left و right: اشاره گرهایی به فرزندان چپ و راست این گره هستند. در گره های برگ مقدار None دارند.
- class_label: برچسب کلاسی که این گره به آن تعلق دارد. فقط در گره های برگ مقداردهی میشود و نشان دهنده پیش بینی نهایی است.
- is_leaf: یک متغیر Boolean که مشخص میکند آیا این گره برگ است یا خیر. اگر True باشد، دیگر تقسیمی انجام نمیشود.

در ابتدای ساخت هر گره، تمام ویژگی ها به مقدار None تنظیم می‌شوند و is_leaf برابر False است. سپس در فرآیند ساخت درخت، این مقادیر به صورت مناسب پر میشوند.

```python
class DecisionTreeC45:

    def __init__(self, max_depth=None, min_samples_split=2):
        """
        Parameters:
        - max_depth: Maximum depth of the tree (None = no limit)
        - min_samples_split: Minimum number of samples required to split a node
        """
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.root = None
        self.n_classes = None
        self.n_features = None


    def entropy(self, y):
        # Count occurrences of each class
        unique_classes, counts = np.unique(y, return_counts=True)

        # Calculate probability of each class
        probabilities = counts / len(y)

        # Calculate entropy
        entropy_value = 0
        for p in probabilities:
            if p > 0:  # Avoid log(0)
                entropy_value -= p * math.log2(p)

        return entropy_value
```

این کد ادامه هم دارد که در فایل اصلی هست اما در اینجا فقط تیکه اول آورده شده است :

1. تابع __init__ (سازنده کلاس)

این تابع پارامترهای اولیه درخت را مقداردهی میکند. max_depth **حداکثر** عمق درخت را مشخص میکند که از رشد بیش از حد درخت و **overfitting** جلوگیری میکند. min_samples_split حداقل تعداد نمونه های لازم برای تقسیم یک گره را تعیین میکنه تا از تقسیم های بی‌معنا در داده های کم جلوگیری کند. متغیرهایroot ، n_classes و n_features در ابتدا None هستند و بعد از آموزش مدل پر میشوند.

---

2. تابع entropy

آنتروپی معیاری برای سنجش میزان ناخالصی یا بی نظمی یا عدم قطعیت در داده ها است. این تابع ابتدا تعداد هر کلاس را شمارش کرده و احتمال هر کلاس را محاسبه میکند. سپس با استفاده از فرمول آنتروپی شانون میزان **عدم قطعیت** را محاسبه می‌کند. اگر تمام داده ها از یک کلاس باشند، آنتروپی صفر است و اگر به طور مساوی توزیع شده باشند، آنتروپی حداکثر خواهد بود. شرط $p > 0$ برای جلوگیری از محاسبه لگاریتم صفر است.

---

3. تابع information_gain_ratio

این تابع نسبت **بهره اطلاعات** را محاسبه میکند که بهبودی نسبت به بهره اطلاعات ساده در الگوریتم C4.5 است. ابتدا بهره اطلاعات را از تفاضل آنتروپی والد و میانگین وزن دار آنتروپی فرزندان به دست می آورد. سپس اطلاعات تقسیم را محاسبه میکند که نشان دهنده میزان تقسیم داده ها است. در نهایت، نسبت این دو مقدار را برمیگرداند. استفاده از نسبت بهره اطلاعات به جای بهره اطلاعات ساده، از انتخاب ویژگی هایی که مقادیر زیادی دارند جلوگیری میکند و باعث ایجاد درخت متوازن تر میشود.

---

4. تابع find_best_split

این تابع وظیفه یافتن بهترین تقسیم برای یک گره را بر عهده دارد. برای هر ویژگی، تمام آستانه های ممکن را امتحان میکند. آستانه ها به صورت میانگین دو مقدار متوالی از مقادیر منحصر به فرد ویژگی انتخاب میشوند. برای هر آستانه، داده ها به دو بخش چپ و راست تقسیم شده و نسبت بهره اطلاعات محاسبه میشود. تقسیمی که بیشترین نسبت بهره اطلاعات را دارد، به عنوان بهترین تقسیم انتخاب و شماره ویژگی، آستانه و نسبت بهره اطلاعات آن برگردانده میشود.

---

5. تابع build_tree

این تابع به صورت بازگشتی درخت را میسازد .ابتدا چهار شرط توقف را بررسی میکند: اگر تمام نمونه ها از یک کلاس باشند، اگر تعداد نمونه ها کمتر از min_samples_split باشد، اگر به حداکثر عمق رسیده باشیم، یا اگر تقسیم خوبی پیدا نشود. در هر یک از این حالات، یک گره برگ با برچسب کلاس اکثریت ایجاد میشود. در غیر این صورت، بهترین تقسیم یافته شده و یک گره تصمیم ساخته میشود. سپس داده ها تقسیم شده و تابع به صورت بازگشتی برای هر دو زیرمجموعه چپ و راست فراخوانی می‌شود تا فرزندان ایجاد شوند.

---

6. تابع fit

این تابع مدل را آموزش میدهد .ابتدا تعداد کلاس ها و ویژگی ها را از داده های ورودی استخراج کرده و ذخیره میکند. سپس با فراخوانی تابع build_tree ، فرآیند ساخت درخت را شروع کرده و ریشه درخت را در متغیر root ذخیره میکند. این تابع خود شیء را برمیگرداند که امکان استفاده زنجیره ای متدها را فراهم میکند.

---

7. تابع predict_single

این تابع پیش بینی برای یک نمونه واحد را انجام میدهد. از ریشه درخت شروع کرده و به صورت بازگشتی در درخت حرکت میکند. اگر به گره برگ برسد، برچسب کلاس آن گره را برمیگرداند. در غیر این صورت، بر اساس مقدار ویژگی مورد نظر و آستانه، تصمیم میگیرد که به سمت چپ یا راست حرکت کند و تابع را برای فرزند مناسب فراخوانی میکند.

---

8. تابع predict

این تابع پیش بینی برای چندین نمونه را به صورت همزمان انجام میدهد. برای هر نمونه در ماتریس ورودی X ، تابع predict_single را فراخوانی کرده و پیش بینی را دریافت میکند. سپس تمام پیش بینی ها را در یک آرایه NumPy جمع آوری کرده و برمیگرداند.

```
# Load Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

print("Iris Dataset Overview")
print(f"\nNumber of samples: {len(X)}")
print(f"Number of features: {X.shape[1]}")
print(f"Number of classes: {len(iris.target_names)}")
print(f"Class names: {iris.target_names}")
print(f"\nFeature names:")
for i, name in enumerate(iris.feature_names, 1):
    print(f"  {i}. {name}")
```

این بخش از کد مسئول بارگذاری مجموعه داده Iris و نمایش اطلاعات اولیه آن است. ابتدا با استفاده از تابع load_iris از کتابخانه sklearn ، داده های Iris بارگذاری میشوند. متغیر X شامل ویژگی های ورودی است و متغیر y شامل برچسب کلاس هر نمونه میباشد.

سپس اطلاعات کلی داده نمایش داده میشود که شامل تعداد کل نمونه ها (150نمونه)، تعداد ویژگی ها (4 ویژگی)، و تعداد کلاس ها (3 کلاس) است. نام کلاس ها شامل سه نوع گل Setosa ، Versicolor و Virginica است. در انتها، نام تمام ویژگی ها به همراه شماره شان نمایش داده میشود تا مشخص شود هر ستون داده به کدام ویژگی تعلق دارد.

```
# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

print("\nData split:")
print(f"  Train: {len(X_train)} samples")
print(f"  Test:  {len(X_test)} samples")
```

این بخش مسئول تقسیم داده ها به دو دسته آموزش و آزمون است. از تابع train_test_split برای این منظور استفاده میشود که داده ها را به صورت تصادفی تقسیم میکند. پارامتر test_size=0.2 مشخص میکند که 20 درصد از داده ها (30 نمونه) برای آزمون و 80 درصد باقیمانده (120 نمونه) برای آموزش در نظر گرفته شوند.

پارامتر random_state=42 برای تکرارپذیری نتایج استفاده میشود تا در هر بار اجرای برنامه، تقسیم بندی یکسانی انجام شود. پارامتر stratify=y نیز تضمین میکند که نسبت کلاس ها در هر دو مجموعه آموزش و آزمون برابر با نسبت آنها در داده اصلی باشد، به این ترتیب از توزیع نابرابر کلاس ها جلوگیری میشود.

نتیجه این تقسیم، چهار متغیر است :X_train (ویژگی‌های آموزش) ، X_test(ویژگی‌های آزمون)، y_train(برچسب‌های آموزش)، و y_test(برچسب‌های آزمون). این تقسیم بندی برای ارزیابی صحیح عملکرد مدل ضروری است تا بتوانیم قدرت تعمیم دهی درخت را روی داده های جدید بسنجیم.

```
# Create and train the model
model = DecisionTreeC45(max_depth=10, min_samples_split=2)
model.fit(X_train, y_train)

print("Model trained successfully!")
```

در این مرحله، یک نمونه از کلاس DecisionTreeC45 ساخته میشود و پارامترهای آن تنظیم میگردد. پارامتر max_depth=10 حداکثر عمق درخت را به 10 سطح محدود میکند که از پیچیدگی بیش از حد درخت و overfitting جلوگیری میکند. پارامتر min_samples_split=2 مشخص میکند که حداقل دو نمونه برای تقسیم یک گره لازم است.

سپس با فراخوانی متد fit و دادن داده های آموزشی (X_train و y_train) به آن، فرآیند آموزش مدل آغاز میشود. در این مرحله، درخت تصمیم به صورت بازگشتی ساخته میشود و بهترین تقسیم ها برای هر گره بر اساس نسبت بهره اطلاعات محاسبه و اعمال میشوند. این فرآیند تا رسیدن به شرایط توقف (مثل رسیدن به حداکثر عمق یا خالص شدن گره ها)ادامه می یابد.

پس از پایان آموزش، ساختار کامل درخت در متغیر model ذخیره میشود و آماده استفاده برای پیش بینی روی داده های جدید است.

```
# Make predictions
y_pred_train = model.predict(X_train)
y_pred_test = model.predict(X_test)

# Calculate accuracy
accuracy_train = np.mean(y_pred_train == y_train)
accuracy_test = np.mean(y_pred_test == y_test)

print("Prediction Results")
print(f"\nTrain Accuracy: {accuracy_train:.4f} ({accuracy_train*100:.2f}%)")
print(f"Test Accuracy:  {accuracy_test:.4f} ({accuracy_test*100:.2f}%)")
```

در این بخش، مدل آموزش دیده برای پیش بینی برچسب کلاس ها استفاده میشود. ابتدا با فراخوانی متد predict روی داده های آموزش(X_train) ، پیش بینی‌های مدل روی همان داده هایی که با آن‌ها آموزش داده شده در متغیر y_pred_train ذخیره میشود. سپس همین کار برای داده های آزمون (X_test) انجام شده و نتایج در y_pred_test قرار میگیرد.

برای ارزیابی عملکرد مدل، معیار دقت (Accuracy) محاسبه میشود. دقت با مقایسه پیش بینی‌ها با برچسب های واقعی و محاسبه درصد پیش بینی های صحیح به دست می آید. از تابع np.mean برای شمارش تعداد پیش بینی های درست استفاده شده است.

نتایج نشان میدهد که مدل روی داده های آموزش دقت بالایی (معمولاً نزدیک به 100%) دارد که نشان دهنده یادگیری خوب الگو ها است .دقت روی داده های آزمون معیار مهم تری است چون قدرت تعمیم دهی مدل روی داده های جدید و دیده نشده را نشان میدهد. تفاوت قابل قبول بین این دو دقت نشان میدهد که مدل دچار overfitting شدید نشده است.

```python
def calculate_metrics(y_true, y_pred, n_classes, class_names):
    """
    Calculate evaluation metrics for each class
    """
    metrics = {}

    for class_idx in range(n_classes):
        # True Positives, False Positives, False Negatives, True Negatives
        tp = np.sum((y_pred == class_idx) & (y_true == class_idx))
        fp = np.sum((y_pred == class_idx) & (y_true != class_idx))
        fn = np.sum((y_pred != class_idx) & (y_true == class_idx))
        tn = np.sum((y_pred != class_idx) & (y_true != class_idx))

        # Calculate metrics
        precision = tp / (tp + fp) if (tp + fp) > 0 else 0
        recall = tp / (tp + fn) if (tp + fn) > 0 else 0
        f1 = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

        metrics[class_names[class_idx]] = {
            'TP': tp, 'FP': fp, 'FN': fn, 'TN': tn,
            'Precision': precision,
            'Recall': recall,
            'F1-Score': f1
        }

    return metrics

# Calculate metrics
metrics_test = calculate_metrics(y_test, y_pred_test, len(iris.target_names), iris.target_names)

print("Detailed Evaluation Metrics (Test Set)")

for class_name in iris.target_names:
    m = metrics_test[class_name]
    print(f"\nClass: {class_name}")
    print(f"   Precision: {m['Precision']:.4f}")
    print(f"   Recall:    {m['Recall']:.4f}")
    print(f"   F1-Score:  {m['F1-Score']:.4f}")

# Average metrics
avg_precision = np.mean([metrics_test[cn]['Precision'] for cn in iris.target_names])
avg_recall = np.mean([metrics_test[cn]['Recall'] for cn in iris.target_names])
avg_f1 = np.mean([metrics_test[cn]['F1-Score'] for cn in iris.target_names])

print("\nAverage Metrics (Macro Average):")
print(f"   Precision: {avg_precision:.4f}")
print(f"   Recall:    {avg_recall:.4f}")
print(f"   F1-Score:  {avg_f1:.4f}")
```

این بخش شامل تابع calculate_metrics است که معیارهای ارزیابی دقیق تر را برای هر کلاس به صورت جداگانه محاسبه می‌کند. این تابع چهار معیار اساسی را برای هر کلاس استخراج میکند True Positives (TP) :که تعداد پیش‌بینی‌های صحیح مثبت، False Positives (FP) که تعداد پیش‌بینی‌های نادرست مثبت، False Negatives (FN) که تعداد نمونه‌های مثبتی که به اشتباه منفی پیش بینی شده اند، و True Negatives (TN) که تعداد پیش‌بینی‌های صحیح منفی است.

سپس بر اساس این چهار معیار، سه معیار مهم محاسبه میشود Precision (دقت): که نشان میدهد از نمونه هایی که مدل به عنوان یک کلاس خاص پیش بینی کرده، چه درصدی واقعاً از آن کلاس هستند. Recall (بازخوانی) که نشان میدهد از کل نمونه

های واقعی یک کلاس، چه درصدی به درستی شناسایی شده اند. F1-Score که میانگین هارمونیک Precision و Recall است و تعادل بین این دو معیار را نشان میدهد.

در انتها، میانگین این معیارها برای همه کلاس ها (Macro Average) محاسبه و نمایش داده میشود.

```
def confusion_matrix(y_true, y_pred, n_classes):
    """
    Calculate Confusion Matrix
    """
    cm = np.zeros((n_classes, n_classes), dtype=int)

    for i in range(len(y_true)):
        actual = y_true[i]
        predicted = y_pred[i]
        cm[actual][predicted] += 1

    return cm

# Calculate Confusion Matrix
cm = confusion_matrix(y_test, y_pred_test, len(iris.target_names))

print("Confusion Matrix (Test Set)")
print("\n" + " "*27 + "Predicted")
header = " "*15
for name in iris.target_names:
    header += f"{name:<12}"
print(header)
print("-"*60)

for i, class_name in enumerate(iris.target_names):
    row = f"Actual {class_name:<10}"
    for j in range(len(iris.target_names)):
        row += f"{cm[i, j]:<12}"
    print(row)
```

تابع confusion_matrix یک جدول دوبعدی به ابعاد n×n ایجاد میکند که در آن n تعداد کلاس ها است. این ماتریس نشان میدهد که مدل چگونه کلاس ها را با یکدیگر اشتباه میگیرد. ابتدا یک ماتریس صفر ایجاد شده و سپس با پیمایش همه نمونه های آزمون، برای هر نمونه برچسب واقعی و پیش بینی شده استخراج میشود و درایه متناظر در ماتریس یک واحد افزایش می یابد.

در نمایش ماتریس، سطرها نشان دهنده کلاس واقعی و ستون ها نشان دهنده کلاس پیش بینی شده هستند. عناصر روی قطر اصلی این ماتریس نشان دهنده پیش بینی های صحیح هستند، در حالی که عناصر خارج از قطر اصلی خطاهای طبقه بندی را نمایش میدهند.

در نتیجه نمایش داده شده، مشاهده میشود که کلاس Setosa به طور کامل تشخیص داده شده و هیچ خطایی ندارد. ولی بین کلاس های Versicolor و Virginica تعدادی اشتباه رخ داده که نشان میدهد این دو کلاس ویژگی های مشابه تری دارند و تفکیک آنها دشوارتر است.