**What this program is doing**

* **Goal:** Learn a yes/no rule (“Play” tennis?) from a small CSV (“Play Tennis” dataset), then use that rule to predict a new case.
* **How:** It implements the classic **ID3 algorithm**:
  1. **Load** the dataset from CSV as a list of dictionaries (each row is a dict: {"Outlook":"Sunny", ... , "Play":"No"}).
  2. **Measure impurity** with **entropy** (how mixed Yes/No is).
  3. **Choose the best question** (attribute) using **information gain** (how much entropy drops after splitting).
  4. **Split** the data on that attribute and **recurse** until leaves are pure (all Yes or all No) or you run out of attributes.
  5. **Classify** a new sample by walking the learned tree from root to leaf.
* **What it achieves:** Builds a nested-dictionary “tree” like  
  {'Outlook': {'Sunny': {'Humidity': {'High': 'No', 'Normal': 'Yes'}}, 'Overcast': 'Yes', 'Rain': {'Wind': {'Strong': 'No', 'Weak': 'Yes'}}}}  
  and predicts the label for a new weather sample.

**Line-by-line explanation**

# -\*- coding: utf-8 -\*-

# task12\_part2\_id3.py

# Simple ID3 Decision Tree Example (Play Tennis dataset)

# Compatible with Python 2.6 (no Counter used)

* Encoding header (safe for non-ASCII comments).
* Comments: this is an ID3 example, kept compatible with **Python 2.6** (so we avoid features like collections.Counter).

import csv

import math

* csv to read the dataset file.
* math for logarithms when computing entropy.

# Simple replacement for Counter

def count\_values(values):

freq = {}

for v in values:

freq[v] = freq.get(v, 0) + 1

return freq

* Tiny helper that counts occurrences of items in a list (e.g., number of “Yes” vs “No”).
* Equivalent to collections.Counter(values), but written manually for Python 2.6.

**Step 1: Load dataset**

def load\_dataset(filename):

dataset = []

with open(filename, "r") as f:

reader = csv.reader(f)

headers = next(reader) # first row is header

for row in reader:

dataset.append(dict(zip(headers, row)))

return dataset, headers

* Opens the CSV.
* reader = csv.reader(f) gives an iterator over rows.
* headers = next(reader) reads the first line (e.g., ["Outlook","Temperature","Humidity","Wind","Play"]).
* For each subsequent row it builds a dict pairing headers to values via zip, e.g.  
  {"Outlook":"Sunny","Temperature":"Hot","Humidity":"High","Wind":"Weak","Play":"No"}
* Returns:
  + dataset: list of row dicts,
  + headers: original header list (useful to know which column is target vs features).

**Step 2: Entropy**

def entropy(rows, target\_attr):

values = [row[target\_attr] for row in rows]

freq = count\_values(values)

total = len(values)

ent = 0.0

for f in freq.values():

p = float(f) / total

ent -= p \* math.log(p, 2)

return ent

* Computes **Shannon entropy** of the **target** (e.g., “Play”).
* values collects the target labels from all rows.
* freq counts how many “Yes” and “No”.
* For each class count f, compute probability p = f / total.
* Entropy formula: H = - Σ p \* log2(p).
  + H=0 means pure (all Yes or all No).
  + Higher H means more mixed.

**Step 3: Information Gain**

def info\_gain(rows, attr, target\_attr):

total\_entropy = entropy(rows, target\_attr)

values = set(row[attr] for row in rows)

weighted\_entropy = 0.0

for v in values:

subset = [row for row in rows if row[attr] == v]

weighted\_entropy += (len(subset) / len(rows)) \* entropy(subset, target\_attr)

return total\_entropy - weighted\_entropy

* **Information Gain (IG)** = how much entropy drops after splitting on attr.
* values are the distinct values of that attribute (e.g., for Outlook: {Sunny, Overcast, Rain}).
* For each value v:
  + Build subset of rows where attr == v.
  + Add the subset’s contribution: **weight** × **entropy(subset)**.
    - Weight is the fraction of rows in that subset.
* Return: IG = H(all) - Σ weight \* H(subset). Higher IG = better split.

⚠ **Python 2 note:** In len(subset) / len(rows), integer division could occur in Python 2.6.  
To be mathematically correct on 2.x, prefer:

weighted\_entropy += (float(len(subset)) / len(rows)) \* entropy(subset, target\_attr)

(Your run still produced a sensible tree, but this cast avoids subtle errors.)

**Step 4: Build the tree (recursive ID3)**

def id3(rows, attrs, target\_attr):

values = [row[target\_attr] for row in rows]

if values.count(values[0]) == len(values):

return values[0] # all same

if not attrs:

freq = count\_values(values)

return max(freq, key=freq.get) # majority vote

* **Base case 1:** If all targets are the same, return that class (make a leaf).
* **Base case 2:** If you’ve run out of attributes to split on, return the **majority class** (tie-breaker leaf).

# Choose best attribute

gains = [(attr, info\_gain(rows, attr, target\_attr)) for attr in attrs]

best\_attr = max(gains, key=lambda x: x[1])[0]

tree = {best\_attr: {}}

* Compute **information gain** for every remaining attribute.
* Pick the best\_attr (highest gain).
* Start building the node: a nested dict where the key is the attribute name, and its value is another dict that will hold branches for each attribute value.

for v in set(row[best\_attr] for row in rows):

subset = [row for row in rows if row[best\_attr] == v]

new\_attrs = [a for a in attrs if a != best\_attr]

tree[best\_attr][v] = id3(subset, new\_attrs, target\_attr)

return tree

* For each **branch value** v of best\_attr:
  + Filter the rows to that branch’s subset.
  + Recurse on that subset with the **reduced attribute list** (we don’t reuse best\_attr below).
  + The recursive call returns either:
    - another dict (internal node), or
    - a string label (leaf), which we store at tree[best\_attr][v].
* Return the constructed (sub)tree.

**Step 5: Classify (walk the tree)**

def classify(tree, sample):

if not isinstance(tree, dict):

return tree

attr = list(tree.keys())[0]

if sample[attr] in tree[attr]:

return classify(tree[attr][sample[attr]], sample)

else:

return "Unknown"

* If tree is **not** a dict, it’s a **leaf** (class label). Return it.
* Otherwise, this is an internal node:
  + Extract the attribute at this node (e.g., “Outlook”).
  + Look up the sample’s value for that attribute (e.g., “Sunny”).
  + If that value exists as a branch, **recurse** into that subtree.
  + If the value wasn’t seen during training, return "Unknown" (a safe fallback).

**Main: glue everything together**

if \_\_name\_\_ == "\_\_main\_\_":

dataset, headers = load\_dataset("/home/cloudera/play\_tennis.csv")

target\_attr = "Play"

attrs = [a for a in headers if a != target\_attr]

* Load the CSV into dataset.
* Define the **target** column ("Play").
* Build the **feature list** (every header except the target).

print("Building ID3 Decision Tree...")

tree = id3(dataset, attrs, target\_attr)

print("Decision Tree:", tree)

* Train the tree with ID3 and print the learned structure.  
  (On Python 2.x you’ll see tuple formatting like ('Decision Tree:', {...}).)

# Test with a new sample

sample = {"Outlook": "Sunny", "Temperature": "Cool", "Humidity": "High", "Wind": "Weak"}

prediction = classify(tree, sample)

print("New Sample:", sample)

print("Predicted Class:", prediction)

* Build a **new** example (not from the CSV).
* **Classify** it using the tree.
* Print both the sample and the prediction.

**Data structures at a glance**

* **Row:** {"Outlook":"Sunny","Temperature":"Hot","Humidity":"High","Wind":"Weak","Play":"No"}
* **Tree:** nested dict, e.g.
* {
* 'Outlook': {
* 'Sunny': {
* 'Humidity': {'High': 'No', 'Normal': 'Yes'}
* },
* 'Overcast': 'Yes',
* 'Rain': {
* 'Wind': {'Strong': 'No', 'Weak': 'Yes'}
* }
* }
* }