Question 1 : What is Information Gain, and how is it used in Decision Trees?

Answer:

Information Gain (IG) is a metric used in decision trees to measure how well a given feature splits a dataset into target classes. It is based on the concept of entropy, which measures the impurity or randomness in the dataset.

Entropy Formula:

$$Entropy(S) = -\Sigma_{i=1}^{C} = pilog2(pi)$$

Where:

- pi is the probability of class i in dataset S.
- c is the total number of classes
- Information Gain Formula:

$$IG(S, A) = Entropy(S) - \sum_{v \in A} \frac{|S_v|}{|s|} \times Entropy(S_v)$$

Where:

- A Is a feature
- Sv is the subset of S where feature A has value v

Use in Decision Trees:

- At each node, the algorithm calculates IG for all features.
- The feature with the highest Information Gain is selected to split the dataset.
- This process continues recursively to build the tree.

Question 2: What is the difference between Gini Impurity and Entropy? Hint: Directly compares the two main impurity measures, highlighting strengths, weaknesses, and appropriate use cases.

Answer:

Definitions/formulas

- Gini Impurity (G):

$$G = 1 - \sum_{i=1}^{k} p_i^2$$

It measures the probability of misclassifying a randomly chosen sample if it were labeled according to the class distribution in the node.

- Entropy (H):

$$H = -\sum_{i=1}^{k} pi \log_2 p_i$$

It measures the expected information (in bits) required to identify the class of a randomly chosen sample.

Range

- Gini: $0 \leq G \leq 1 - \frac{1}{k}$ (for binary classes $0 \leq G \leq 0.5$).
- Entropy: $0 \leq H \leq log_2\, k$ (for binary classes$0 \leq H \leq 1$).

Behavioral comparison

- Both measure impurity; both are 0 when the node is pure.
- Gini is a quadratic function of class probabilities (computationally cheaper — no logs).
- Entropy grows more slowly near pure distributions but penalizes mid-range uncertainty more subtly.

When they differ practically

- For many splits they produce similar rankings of candidate splits (i.e., they often choose the same feature), but small differences can appear in edge cases.
- Gini tends to prefer larger, more balanced partitions slightly (because of quadratic form), while entropy is a bit more sensitive to distribution differences.

Computation speed

- Gini is faster (no logarithms). This is why CART (Classification And Regression Trees) uses Gini by default.

Use cases / algorithm defaults

Gini

- Gini impurity — used by CART (and hence scikit-learn's DecisionTreeClassifier default). while using the (Classification and Regression Trees) algorithm.
- while working with large datasets where speed matters more than slight gains in accuracy.

Entropy

- Entropy — used by ID3/C4.5 family and sometimes chosen when interpreting splits in information-theory terms is desirable.
- while handling datasets where class distribution is highly imbalanced or where accuracy is more important than speed

Strengths and weaknesses

- Gini:
    - Faster to compute.
    - Often produces shallower trees (practical advantage).
    - − Slight bias towards features with more categories if not controlled.
- Entropy:
    - Theoretically grounded in information theory; good for interpreting information gain.
    - − Slightly slower to compute; may produce different splits in some datasets.

Recommendation:

Use Gini for performance and typically similar results. Use Entropy when you want information-theoretic interpretability or when experimenting shows entropy gives better validation performance for your dataset.

Question 3:What is Pre-Pruning in Decision Trees?

Answer:

Pre-pruning (also called early stopping) is the practice of halting the growth of a decision tree before it perfectly fits (or grows deep on) the training data, using predefined constraints. The goal is to prevent overfitting by limiting complexity during training.

Common pre-pruning hyperparameters (scikit-learn names included):

- max_depth — maximum depth of the tree.
- min_samples_split — minimum number of samples required to split an internal node.
- min_samples_leaf — minimum number of samples required to be at a leaf node.
- max_leaf_nodes — maximum number of leaf nodes.
- min_impurity_decrease — a split will be made only if it decreases impurity by at least this threshold.
- ccp_alpha — complexity parameter used for Minimal Cost-Complexity Pruning (but note: ccp_alpha is applied in a post-pruning style in scikit-learn via pruning after full growth).

How pre-pruning works in practice

- The tree stops splitting further if:
    - Maximum depth is reached (max_depth)
    - Minimum number of samples to split is not met (min_samples_split)
    - Minimum samples in a leaf node is small (min_samples_leaf)
    - Information Gain or Gini decrease is below a threshold.
    - During tree expansion, if a potential split would violate any pre-pruning condition (e.g., would create leaves with fewer than min_samples_leaf), the split is not performed and the node becomes a leaf.
    - This keeps the tree simpler and reduces variance at the cost of possibly increasing bias.

    Advantages

    - Faster training time
    - Produces simpler and more interpretable trees

- Controls overfitting by reducing tree complexity.
- Less computational time and memory than growing a large tree and post-pruning.
- Direct control over model complexity.

Disadvantages

- Choosing hyperparameters incorrectly can lead to underfitting.
- It might stop growth too early, preventing the discovery of useful, fine-grained patterns.

Question 4:Write a Python program to train a Decision Tree Classifier using Gini

Impurity as the criterion and print the feature importances (practical). Hint: Use criterion='gini' in DecisionTreeClassifier and access .feature_importances_. (Include your Python code and output in the code box below.)

```
# decision_tree_gini_feature_importance.py
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report

# 1. Load data
iris = load_iris()
X, y = iris.data, iris.target
feature_names = iris.feature_names
target_names = iris.target_names
```

```
# 2. Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# 3. Train Decision Tree with Gini
clf = DecisionTreeClassifier(criterion='gini', random_state=42, max_depth=None)
clf.fit(X_train, y_train)

# 4. Predict & evaluate
y_pred = clf.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification report:\n", classification_report(y_test, y_pred, target_names=target_names))

# 5. Feature importances
importances = clf.feature_importances_
print("\nFeature importances (feature_name: importance):")
for name, score in zip(feature_names, importances):
    print(f"{name}: {score:.4f}")

# 6. (Optional) Sort and display
order = np.argsort(importances)[::-1]
print("\nFeatures ranked by importance:")
for idx in order:
    print(f"{feature_names[idx]}: {importances[idx]:.4f}")
```

"""Interpretation of .feature_importances_

Feature importance values show the normalized total reduction of the criterion (Gini impurity) brought by that feature. Larger values → more important feature."""

Output:

Accuracy: 0.9333333333333333

Classification report:

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	10
versicolor	0.90	0.90	0.90	10
virginica	0.90	0.90	0.90	10
accuracy			0.93	30
macro avg	0.93	0.93	0.93	30
weighted avg	0.93	0.93	0.93	30

Feature importances (feature_name: importance):

sepal length (cm): 0.0062

sepal width (cm): 0.0292

petal length (cm): 0.5586

petal width (cm): 0.4060

```
Features ranked by importance:

petal length (cm): 0.5586

petal width (cm): 0.4060

sepal width (cm): 0.0292

sepal length (cm): 0.0062

'Interpretation of .feature_importances_\nFeature importance values
show the normalized total reduction of the criterion (Gini impurity)
brought by that feature. Larger values → more important feature.'
```

Question 5: What is a Support Vector Machine (SVM)?

Answer:

Definition.

A Support Vector Machine (SVM) is a supervised machine learning algorithm used primarily for classification (and can be adapted for regression). It finds an optimal separating hyperplane between classes by maximizing the margin — the distance between the hyperplane and the nearest data points from each class (called support vectors).

Key concepts

- Hyperplane: In d-dimensional space, a (d−1)-dimensional hyperplane separates the classes.
- Margin: Distance between the hyperplane and the closest points. SVM seeks to maximize this margin.
- Support Vectors: Points lying closest to the decision boundary; they determine the position of the hyperplane.
- Hard-margin vs Soft-margin:
    - Hard-margin SVM requires perfectly separable data.
    - Soft-margin SVM introduces slack variables and parameter C to allow some misclassification while controlling regularization.

Objective (primal form, linear separable):

- Find $\boldsymbol{\omega}$ and $\boldsymbol{b}$ to minimize $\frac{1}{2}\left|\left|\omega\right|\right|^2$ subject to $\boldsymbol{y_i}(\boldsymbol{\omega^T x_1} + \boldsymbol{b}) \geq \mathbf{1}$ for all i.

Strengths

- Effective in high-dimensional spaces.
- Uses a subset of training points (support vectors), so memory efficient.
- Robust to overfitting with appropriate regularization (C parameter).

Weaknesses

- Choice of kernel and hyperparameters critically affects performance.

Scalability:

- training time can be high for very large datasets (though linear SVMs and approximations exist)

Question 6: What is the Kernel Trick in SVM?

Answer:

Definition

- The **Kernel Trick** is a method used in Support Vector Machines (SVMs) and other kernelized algorithms. It allows them to implicitly map data into a higher-dimensional feature space without explicitly computing the coordinates of the data in that space. This enables the algorithm to find a linear separating hyperplane in the higher-dimensional space, which corresponds to a non-linear decision boundary in the original lower-dimensional space.xj^{x_i}

How it works

- Instead of transforming the data points x_i and x_j into the higher-dimensional space (x_i) and (x_j) and then computing their dot product , the kernel trick uses a **kernel function** $K(x_{i\,,}\, x_j)$ that directly computes the dot product $\phi(x_i) \cdot \phi\big(x_j\big)$in the higher-dimensional space:

  $K(x_{i\,,}\, x_j) = \phi(x_i) \cdot \phi\big(x_j\big)$

This avoids the computationally expensive explicit mapping to the high-dimensional space, especially when the dimensionality is very large or even infinite.

Mercer's Theorem

- For a function $\boldsymbol{K}(\boldsymbol{x}, \boldsymbol{x}')$ to be a valid kernel, it must produce a positive semi-definite Gram matrix for all finite sets of inputs. This theorem ensures that the kernel function

corresponds to an inner product in some feature space, allowing the kernel trick to work.

Common Kernel Functions

- **Linear Kernel:** $K(x_i, x_j) = x_i \cdot x_j$ (This is equivalent to no kernel trick, just a linear SVM)
- **Polynomial Kernel:** $K(x_i, x_j) = (\gamma x_i \cdot x_j + r)^d$, where γ , r , and d parameters.
- **Radial Basis Function (RBF) Kernel:** $K(x_i, x_j) = \exp\left(-\gamma \|x_i - x_j\|^2\right)$, where γ is a parameter. This is one of the most commonly used kernels.
- **Sigmoid Kernel:** $K(x_i, x_j) = \tanh(\gamma x_i \cdot x_j + r)$, where γ and r are parameters.

Advantages

- Allows SVMs to learn non-linear decision boundaries.
- Avoids the computational cost of explicitly mapping data to high dimensions.
- Enables working with infinite-dimensional feature spaces (e.g., with the RBF kernel).

Disadvantages

- Choosing the right kernel and its parameters can be challenging and often requires experimentation (e.g., using cross-validation).
- Can still be computationally expensive for very large datasets, even with the kernel trick.

Question 7: Write a Python program to train two SVM classifiers with Linear and RBF kernels on the Wine dataset, then compare their accuracies. Hint:Use SVC(kernel='linear') and SVC(kernel='rbf'), then compare accuracy scores after fitting on the same dataset. (Include your Python code and output in the code box below.)

Answer:

```
# svm_wine_compare.py

from sklearn.datasets import load_wine

from sklearn.svm import SVC

from sklearn.model_selection import train_test_split, GridSearchCV
```

```
from sklearn.metrics import accuracy_score, classification_report
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Load
data = load_wine()
X, y = data.data, data.target

# Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Scaling is recommended for SVMs
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Linear SVM
svm_linear = SVC(kernel='linear', C=1.0, random_state=42)
svm_linear.fit(X_train_scaled, y_train)
y_pred_lin = svm_linear.predict(X_test_scaled)
acc_lin = accuracy_score(y_test, y_pred_lin)

# RBF SVM (default gamma='scale')
svm_rbf = SVC(kernel='rbf', C=1.0, gamma='scale', random_state=42)
svm_rbf.fit(X_train_scaled, y_train)
```

```
y_pred_rbf = svm_rbf.predict(X_test_scaled)
acc_rbf = accuracy_score(y_test, y_pred_rbf)

print(f"Linear SVM accuracy: {acc_lin:.4f}")
print(f"RBF SVM accuracy:    {acc_rbf:.4f}")
print("\nClassification report for RBF SVM:\n",
classification_report(y_test, y_pred_rbf))
```

output:

Linear SVM accuracy: 0.9444

RBF SVM accuracy: 0.9722

Classification report for RBF SVM:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	12
1	0.93	1.00	0.97	14
2	1.00	0.90	0.95	10
accuracy			0.97	36
macro avg	0.98	0.97	0.97	36
weighted avg	0.97	0.97	0.97	36

Question 8: What is the Naïve Bayes classifier, and why is it called "Naïve"?

Answer:

Naïve Bayes (definition)

Naïve Bayes classifiers are a family of probabilistic classifiers based on Bayes' theorem with the (naïve) assumption that the features are conditionally independent given the target class.it is mainly used for classification tasks

Bayes' theorem (for classification):

$$\boldsymbol{P_{(y|x)} = \frac{P_{(x|y)} P_{(y)}}{P(x)}}$$

For classification, we seek the class that maximizes. Because it $P(x|y) = \Pi_i P(x_i|y)$is constant across classes, we use:

$$\hat{y} = argmax_y P(y) \Pi_{i=1}^{d} P(x_i|y)$$

Where $x = (x, \dots, x_d)$.

Why "naïve"?

- It assumes that all features are independent of each other, which is rarely true in real-world data. This assumption is simple (naïve) but makes the computation very fast and effective.
- The classifier assumes conditional independence between features x_i given the class y, i.e., $P(x|y) = \Pi_i P(x_i|y)$. This assumption is often false in practice (features are frequently correlated), hence the term "naïve".

Despite the naive assumption, why it works well:

- Works well in many practical tasks (especially text classification/spam detection) because it estimates class-conditional probability distributions robustly and requires relatively few parameters.
- Extremely fast to train and predict; works well with high-dimensional data.
- Performs well with small training datasets and noisy data.

Limitations

- If feature dependence is strong and critical to class discrimination, performance may suffer.

- Requires appropriate probability models for features (Gaussian for continuous, Multinomial for counts, Bernoulli for binary features)

Question 9: Explain the differences between Gaussian Naïve Bayes, Multinomial Naïve Bayes, and Bernoulli Naïve Bayes

Answer:

Overview / Intuition

Naïve Bayes variants differ primarily in the assumed distribution for the feature likelihood $P(x_i|y)$. Choose the variant that matches the nature of your features.

Gaussian Naïve Bayes

- **Use when:** Features are continuous and roughly normally distributed within each class.
- **Model:** For each feature and class, model $P(x_i|y)$ as a Gaussian:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_{i,y}^2}} e \sup\left(-\frac{(x_i - \mu_i, y)^2}{2\sigma_{i,y}^2}\right)$$

where μ_i, y and $\sigma_{i,y}^2$ are estimated from training data.

- **Typical use:** Numeric features like height, weight, lab measurements.

Multinomial Naïve Bayes

- **Use when:** Features are discrete counts (e.g., word counts in text classification — bag-of-words).
- **Model:** Class-conditional probability of counts; each class has a probability distribution over features (word probabilities). The likelihood of the feature vector is multinomial.
- **Smoothing:** Usually uses Laplace (add-one) smoothing to handle zero counts.
- **Typical use:** Document classification, where x_i is count of word *i.*

Bernoulli Naïve Bayes

- **Use when:** Features are binary (0/1), representing presence/absence of a feature (e.g., whether a word appears in a document).

- **Model:** Each feature modeled as a Bernoulli trial with probability $P(x_i = 1|y)$.
- **Typical use:** Binary feature vectors, e.g., "term present or not" in text.

Choice Summary:

- If features are real-valued and roughly normal → GaussianNB.
- If features are counts (frequency of terms) → MultinomialNB.
- If features are binary (present/absent) → BernoulliNB.

Question 10: Breast Cancer Dataset

Write a Python program to train a Gaussian Naïve Bayes classifier on the Breast Cancer dataset and evaluate accuracy. Hint:Use GaussianNB() from sklearn.naive_bayes and the Breast Cancer dataset from sklearn.datasets. (Include your Python code and output in the code box below.)

Answer:

```
# gnb_breast_cancer.py

from sklearn.datasets import load_breast_cancer

from sklearn.model_selection import train_test_split, cross_val_score

from sklearn.naive_bayes import GaussianNB

from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

from sklearn.preprocessing import StandardScaler

import numpy as np


# 1. Load data

data = load_breast_cancer()

X, y = data.data, data.target

feature_names = data.feature_names

target_names = data.target_names
```

```
# 2. Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# 3. Scaling (not strictly required for GaussianNB, but sometimes helps)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 4. Train GaussianNB
gnb = GaussianNB()
gnb.fit(X_train_scaled, y_train)

# 5. Predict & evaluate
y_pred = gnb.predict(X_test_scaled)
acc = accuracy_score(y_test, y_pred)
print(f"Test accuracy: {acc:.4f}")
print("\nClassification report:\n", classification_report(y_test, y_pred, target_names=target_names))
print("\nConfusion matrix:\n", confusion_matrix(y_test, y_pred))



"""Interpretation
```

```
classification_report shows per-class precision, recall (sensitivity),
and F1-score — important for medical datasets where sensitivity
(recall for the positive class) often matters"""


output:

Test accuracy: 0.9298


Classification report:
               precision    recall  f1-score   support

   malignant       0.90      0.90      0.90        42
      benign       0.94      0.94      0.94        72

    accuracy                           0.93       114
   macro avg       0.92      0.92      0.92       114
weighted avg       0.93      0.93      0.93       114



Confusion matrix:
 [[38  4]
 [ 4 68]]
'Interpretation\n\nclassification_report shows per-class precision,
recall (sensitivity), and F1-score — important for medical datasets
\nwhere sensitivity (recall for the positive class) often matters'
```