NOTRE DAME UNIVERSITY BANGLADESH

Machine Learning Lab Report-07

Course Code: CSE4214

Course Title: Machine Learning Lab

Lab Task Topic: Decision Tree & Confusion Matrix

Submitted by:

Name: Istiak Alam

ID: 0692230005101005

Batch: CSE-20

Submission Date: April 26, 2026

Submitted to:

A. H. M. Saiful Islam

Chairman, Dept of CSE

Notre Dame University Bangladesh

Table of Contents

1 Objective **1**

2 Dataset Description **1**

3 Decision Tree Classifier **1**

3.1 Implementing on Tennis Dataset 1
3.2 Data Preprocessing and Model Training 2
3.3 Decision Tree Model Accuracy Evaluation 3
3.4 Decision Tree - Confusion Matrix Evaluation 4
3.5 Decision Tree - Actual vs Predicted Comparison 5
3.6 Decision Tree Prediction on New Input 5

4 Decision tree from Dataset to predict as a Regressor **6**

4.1 Decision Tree Regressor - House Price Prediction 6
4.2 Decision Tree Prediction and Evaluation 7
4.3 Decision Tree Price Prediction Function 8

5 Confusion Matrix **9**

5.1 Confusion Matrix for Classification Evaluation 9
5.2 Confusion Matrix and Classification Report Evaluation 10
5.3 f1 score for “Dog” class 11
5.4 f1 score for “not a Dog” class 11

1 Objective

The objective of this experiment is to implement the Decision Tree algorithm for classification tasks. This experiment aims to build a model that can classify data into distinct categories based on feature values by learning decision rules from the dataset. It also focuses on understanding how tree-based models split data and evaluate their classification performance.

The objective of this experiment is to apply the Decision Tree algorithm for regression tasks. The goal is to predict continuous target values by learning patterns from input features using a tree-structured model. This experiment helps in understanding how decision trees can approximate numerical relationships and handle non-linear data.

The objective of this experiment is to evaluate the performance of a classification model using a confusion matrix. It aims to analyze the model's prediction results by comparing actual and predicted class labels, and to compute performance metrics such as accuracy, precision, recall, and F1-score for better assessment of the model.

2 Dataset Description

Three different datasets are used in this lab to demonstrate the application of regression algorithms.

Dataset Description: DT.csv

The `DT.csv` dataset is used for implementing Decision Tree algorithms. It contains multiple attributes representing different features of the data along with a target variable used for classification or prediction. The dataset is structured to demonstrate how a Decision Tree model splits data based on feature values to make decisions. It is suitable for understanding tree-based learning, feature importance, and rule-based classification.

Dataset Description: house_data.csv

The `house_data.csv` dataset contains real estate information used for predicting house prices. It includes several numerical and possibly categorical features such as area, number of bedrooms, location factors, and other property-related attributes. The target variable represents the house price. This dataset is used to analyze how different factors influence property prices and to apply regression techniques for accurate prediction.

3 Decision Tree Classifier

3.1 Implementing on Tennis Dataset

Explanation

This code imports essential machine learning libraries such as pandas for data handling, scikit-learn modules for building and evaluating a Decision Tree classifier. The dataset is loaded from a CSV file named `DT.csv`, which likely contains tennis-related attributes used for classification. A Decision Tree model is typically used to learn decision rules from data features and predict categorical outcomes.

[1]:
```
# Importing necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
```

```
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, confusion_matrix,␣
 ↪ConfusionMatrixDisplay
```

[2]:
```
# Load the dataset
data = pd.read_csv('DT.csv')
```

[3]:
```
data
```

Output

The output of this step is the successful loading of the dataset into a pandas DataFrame. No model training or prediction occurs yet at this stage. If printed, it would display the first few rows or structure of the dataset, confirming that the data has been read correctly and is ready for preprocessing and model training.

[3]:

	Day	Outlook	Temp	Humidity	Wind	PlayTennis	Unnamed: 6
0	D1	sunny	hot	high	weak	No	NaN
1	D2	sunny	hot	high	strong	No	NaN
2	D3	overcast	hot	high	weak	Yes	NaN
3	D4	rain	mild	high	weak	Yes	NaN
4	D5	rain	cold	normal	weak	Yes	NaN
5	D6	rain	cold	normal	strong	No	NaN
6	D7	overcast	cold	normal	strong	Yes	NaN
7	D8	sunny	mild	high	weak	No	NaN
8	D9	sunny	cold	normal	weak	Yes	NaN
9	D10	rain	mild	normal	weak	Yes	NaN
10	D11	sunny	mild	normal	strong	Yes	NaN
11	D12	overcast	mild	high	strong	Yes	NaN
12	D13	overcast	hot	normal	weak	Yes	NaN
13	D14	rain	mild	high	strong	No	NaN
14	D15	sunny	hot	normal	weak	No	NaN

3.2 Data Preprocessing and Model Training

Explanation

This code performs preprocessing and model training for a Decision Tree classifier. First, categorical variables in the dataset are converted into numeric form using `LabelEncoder`, since machine learning models cannot directly process text data. Each column (except identifiers like `Day`) is encoded and stored.

Next, the dataset is split into feature variables `X` (Outlook, Temperature, Humidity, Wind) and target variable `y` (PlayTennis). The data is then divided into training and testing sets using an 80-20 split.

A `DecisionTreeClassifier` is trained using the training data, and predictions are made on the test set.

[4]:
```
# Encoding categorical variables
label_encoders = {}
for column in data.columns[1:]: # Ignore 'Day' as it's an identifier
    le = LabelEncoder()
    data[column] = le.fit_transform(data[column])
```

```
    label_encoders[column] = le
```

[5]:
```
# Separating features and target variable
X = data[['Outlook', 'Temp', 'Humidity', 'Wind']] # Feature columns
y = data['PlayTennis'] # Target column
```

[6]:
```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

[7]:
```
# Train the Decision Tree classifier
model = DecisionTreeClassifier(random_state=42)
model.fit(X_train, y_train)
# Make predictions on the test set
y_pred = model.predict(X_test)
```

[8]:
```
y_pred
```

[8]:
```
array([1, 1, 0])
```

[9]:
```
y_test
```

Output

The output includes the trained model and predicted values (`y_pred`) for the test dataset. The comparison between `y_pred` and actual values (`y_test`) shows how well the model performs. If printed, both arrays will display encoded class labels (e.g., 0 and 1), which can later be evaluated using accuracy or confusion matrix.

[9]:
```
9     1
11    1
0     0
Name: PlayTennis, dtype: int64
```

3.3 Decision Tree Model Accuracy Evaluation

Explanation

This code calculates the performance of the trained Decision Tree classifier by comparing the predicted labels (`y_pred`) with the actual test labels (`y_test`). The function `accuracy_score` from scikit-learn is used to compute the proportion of correctly classified instances. The result is stored in the variable `accuracy` and then printed.

[10]:
```
# Calculate and print accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Model accuracy:", accuracy)
```

Output

The output displays the model's accuracy as a numeric value between 0 and 1 (or 0% to 100%).

```
Model accuracy: 1.0
```

3.4 Decision Tree - Confusion Matrix Evaluation

Explanation

This code evaluates the performance of the trained Decision Tree classifier using a confusion matrix. The function `confusion_matrix(y_test, y_pred)` compares the actual labels (`y_test`) with the predicted labels (`y_pred`) and produces a matrix showing correct and incorrect classifications. Each row represents actual classes, while each column represents predicted classes. Additionally, `ConfusionMatrixDisplay.from_predictions()` visualizes this matrix with proper class labels using the encoded target variable `PlayTennis`.

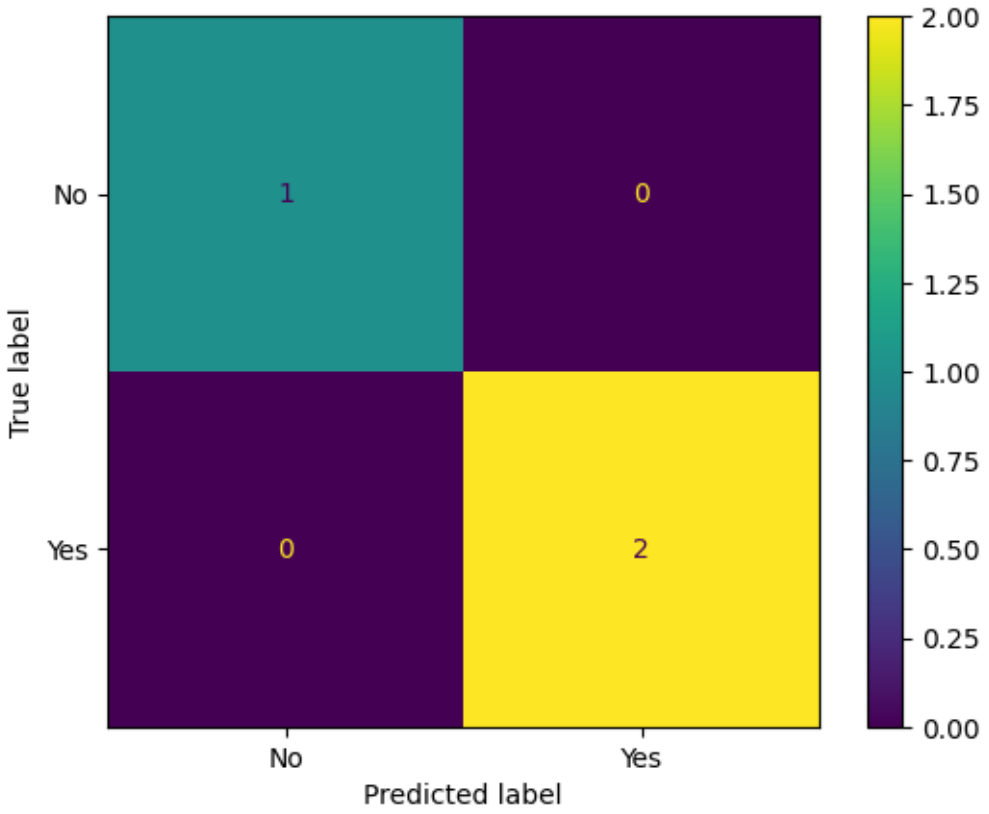
[11]:
```
# Display confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:")
print(conf_matrix)
ConfusionMatrixDisplay.from_predictions(y_test, y_pred,␣
 ↪display_labels=label_encoders['PlayTennis'].classes_)
```

Output

The output includes a printed confusion matrix in numerical form, showing how many predictions were correct and where misclassifications occurred.

```
Confusion Matrix:
[[1 0]
 [0 2]]
```

[11]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at
0x7f2c94464e50>
```



3.5 Decision Tree - Actual vs Predicted Comparison

Explanation

This code compares the actual target values with the predicted values generated by the Decision Tree classifier. It iterates through the test dataset using `y_test` (true labels) and `y_pred` (model predictions). Since the labels were originally encoded using `LabelEncoder`, the code converts them back to their original categorical form (e.g., Yes/No for PlayTennis) using `inverse_transform` for better interpretability.

[12]:
```
# Show actual vs predicted values for each test sample
print("\nActual vs Predicted:")
for actual, predicted in zip(y_test, y_pred):
    print(f"Actual: {label_encoders['PlayTennis'].
 ↪inverse_transform([actual])[0]}, "f"Predicted:␣
 ↪{label_encoders['PlayTennis'].inverse_transform([predicted])[0]}")
```

Output

The output prints a list of test samples showing side-by-side comparison of actual vs predicted values. Each line displays the true class (e.g., PlayTennis = Yes/No) and the model's predicted class for that same instance. This helps evaluate how accurately the Decision Tree classifier is performing on unseen data.

```
Actual vs Predicted:
Actual: Yes, Predicted: Yes
Actual: Yes, Predicted: Yes
Actual: No, Predicted: No
```

3.6 Decision Tree Prediction on New Input

Explanation

This code prepares a new input sample for prediction using a trained Decision Tree model. Since machine learning models require numerical input, categorical values such as `Outlook`, `Temp`, `Humidity`, and `Wind` are first converted into encoded numeric form using previously fitted `LabelEncoder` objects. The input corresponds to a condition: rain, mild temperature, high humidity, and strong wind. After preprocessing, the model predicts whether tennis should be played based on learned patterns from the dataset. Finally, the predicted numeric output is converted back into a readable label using inverse transformation.

Output

The model outputs a single classification result indicating whether playing tennis is suitable under the given weather conditions. For the input {rain, mild, high, strong}, the printed output will typically show either **Play Tennis = Yes** or **Play Tennis = No**, depending on the patterns learned during training. This confirms that the model can generalize and make decisions on unseen data.

[13]:
```
# Example prediction for new input {rain, mild, high, strong}
input_data = pd.DataFrame({
    'Outlook': [label_encoders['Outlook'].transform(['rain'])[0]],
    'Temp': [label_encoders['Temp'].transform(['mild'])[0]],
    'Humidity': [label_encoders['Humidity'].transform(['high'])[0]],
    'Wind': [label_encoders['Wind'].transform(['strong'])[0]]
```

```
})
```

[14]:
```
# Predict whether to play tennis
prediction = model.predict(input_data)
result = label_encoders['PlayTennis'].inverse_transform(prediction)
print("\nPrediction for input {rain, mild, high, strong}: Play Tennis =",␣
 ↪result[0])
```

```
Prediction for input {rain, mild, high, strong}: Play Tennis = No
```

4 Decision tree from Dataset to predict as a Regressor

4.1 Decision Tree Regressor - House Price Prediction

Explanation

This code implements a Decision Tree Regressor model to predict house prices using a dataset named `house_data_DT.csv`. First, necessary libraries are imported, including pandas for data handling, train_test_split for splitting the dataset, and DecisionTreeRegressor for regression modeling. The dataset is then loaded and cleaned by stripping whitespace from column names and validating the presence of the `Location` column.

The `Location` feature is converted into numerical form using manual encoding (Suburb = 0, City = 1). Missing or invalid values are checked, and any incomplete rows are removed to ensure data consistency. After preprocessing, the dataset is split into features (Rooms, Area, Location) and target variable (Price). Finally, the data is divided into training and testing sets, and a Decision Tree Regressor is trained using the training data.

[15]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error, r2_score
```

[16]:
```
# Load the dataset
data = pd.read_csv('house_data_DT.csv')
```

[17]:
```
 # Ensure all column names are stripped of whitespace
data.columns = data.columns.str.strip()
# Check if 'Location' exists
if 'Location' not in data.columns:
    raise KeyError("'Location' column is missing!")

# Convert all values in the 'Location' column to strings and strip whitespace
data['Location'] = data['Location'].astype(str).str.strip()

# Encode the 'Location' column
data['Location'] = data['Location'].map({'Suburb': 0, 'City': 1})

# Check for missing or unmapped values
if data['Location'].isnull().any():
```

```
    print("Warning: Some rows in 'Location' have unmapped or invalid values!
 ↪")
    print(data[data['Location'].isnull()])
    data = data.dropna() # Drop rows with unmapped values if necessary
```

[18]:
```
# Define features (X) and target (y)
X = data[['Rooms', 'Area', 'Location']]
y = data['Price']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
```

[19]:
```
# Train the Decision Tree Regressor
regressor = DecisionTreeRegressor(random_state=42)
regressor.fit(X_train, y_train)
```

Output

The output of this stage includes a cleaned and preprocessed dataset with no missing or invalid values, followed by successful training of the Decision Tree Regressor model. No prediction is shown yet at this step, but the model is now ready to estimate house prices based on input features in later stages.

[19]:
```
DecisionTreeRegressor(random_state=42)
```

4.2 Decision Tree Prediction and Evaluation

Explanation

This code uses a trained Decision Tree regressor model to make predictions on the test dataset (`X_test`). The predicted values (`y_pred`) are then compared with the actual target values (`y_test`) to evaluate model performance. Two evaluation metrics are used: Mean Squared Error (MSE), which measures the average squared difference between actual and predicted values, and R^2 Score, which indicates how well the model explains the variance in the data.

[20]:
```
# Make predictions on the test set
y_pred = regressor.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
print(f"R^2 Score: {r2}")
```

Output

The output displays two numerical evaluation results. The Mean Squared Error (MSE) shows how much error the model is making on average (lower is better). The R^2 Score indicates the goodness of fit, where a value closer to 1 means the model is performing well in predicting the target variable. These results help assess the accuracy and reliability of the Decision Tree model.

```
Mean Squared Error: 2500000000.0
R^2 Score: -10.11111111111111
```

4.3 Decision Tree Price Prediction Function

Explanation

This code defines a function `predict_price()` that takes user inputs such as number of rooms, area, and location to predict a house price using a trained regression model (referred to as `regressor`). The categorical variable `location` is manually encoded into numerical form (Suburb = 0, City = 1) to match the model's training format. The inputs are then converted into a pandas DataFrame and passed into the model's `predict()` function to generate the estimated price.

[21]:
```
# Function to predict price based on user input
def predict_price(rooms, area, location):
    # Encode the location input to match training data encoding
    location_encoded = 0 if location.lower() == 'suburb' else 1

    # Prepare the input data
    input_data = pd.DataFrame({
    'Rooms': [rooms],
    'Area': [area],
    'Location': [location_encoded]
    })

# Make the prediction
    predicted_price = regressor.predict(input_data)
    print("\nPredicted Price for input (Rooms: {}, Area: {}, Location: {}):␣
 ↪${:,.2f}".format(rooms, area, location, predicted_price[0]))

# Example prediction
predict_price(rooms=3, area=75, location='Suburb')
predict_price(rooms=4, area=95, location='City')
```

Output

The function prints the predicted house price for given input values. For example, when called with (Rooms=3, Area=75, Location='Suburb'), it outputs a lower estimated price compared to (Rooms=4, Area=95, Location='City'), as larger area and city location typically increase property value. The output is displayed in a formatted currency style showing the predicted price for each test case.

```
Predicted Price for input (Rooms: 3, Area: 75, Location: Suburb): $320,000.00

Predicted Price for input (Rooms: 4, Area: 95, Location: City): $450,000.00
```

5 Confusion Matrix

5.1 Confusion Matrix for Classification Evaluation

Explanation

This code demonstrates how to evaluate a classification model using a confusion matrix. It imports necessary libraries including pandas, matplotlib, seaborn, and scikit-learn metrics. A custom function `plot_confusion_matrix()` is defined to visualize the confusion matrix using a heatmap. The actual labels (`truth`) and predicted labels (`prediction`) are defined for a binary classification problem (Dog vs Not a dog). The confusion matrix is then computed using `confusion_matrix()` from scikit-learn, which compares actual vs predicted values and summarizes correct and incorrect classifications.

[22]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix , classification_report
```

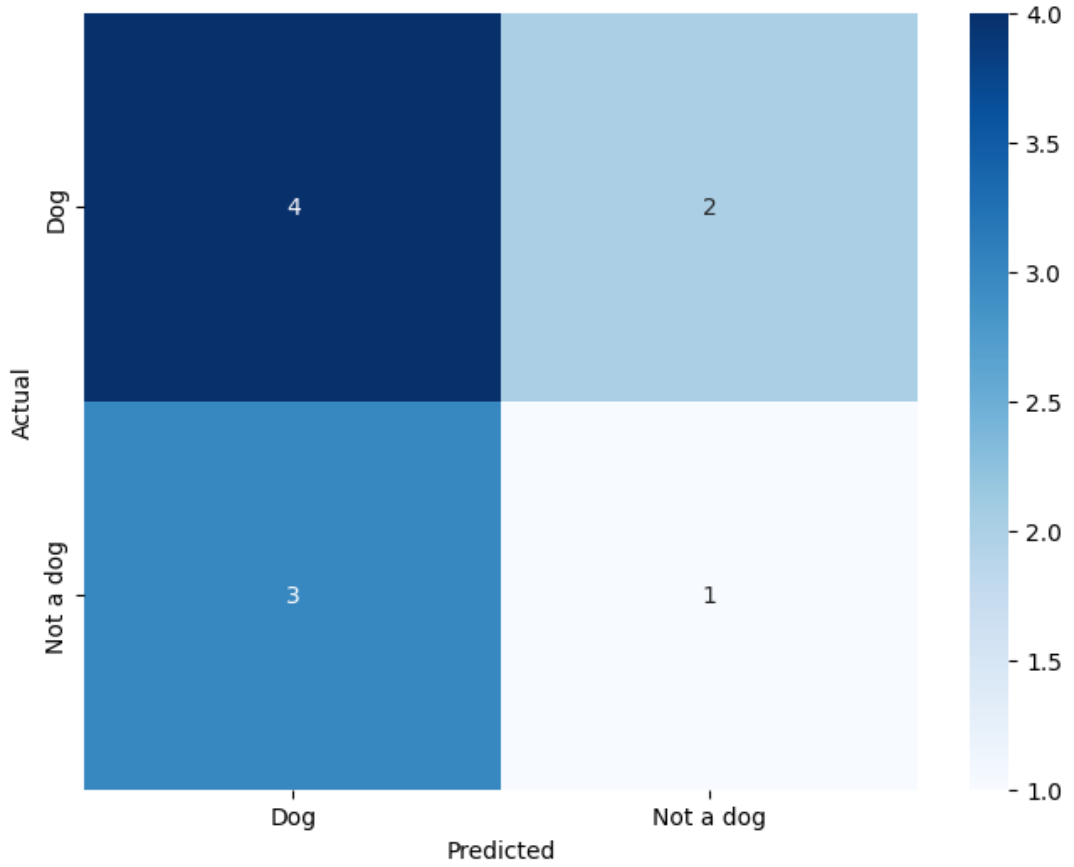
[23]:
```
def plot_confusion_matrix(conf_matrix, class_names):
    df_cm = pd.DataFrame(conf_matrix, index=class_names, columns=class_names)
    plt.figure(figsize=(8, 6))
    sns.heatmap(df_cm, annot=True, fmt='d', cmap='Blues')
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.show()
```

[24]:
```
truth = ["Dog","Not a dog","Dog","Dog", "Dog", "Not a dog", "Not a dog",␣
 ↪"Dog", "Dog", "Not a dog"]
prediction = ["Dog","Dog", "Dog","Not a dog","Dog", "Not a dog", "Dog", "Not␣
 ↪a dog", "Dog", "Dog"]
```

[25]:
```
from sklearn.metrics import confusion_matrix
# Assuming 'truth' and 'prediction' are defined earlier in your code
cm = confusion_matrix(truth, prediction)
plot_confusion_matrix(cm, ["Dog", "Not a dog"])
```

Output

The output is a 2×2 confusion matrix heatmap showing the classification performance. The diagonal values represent correctly classified samples (true positives and true negatives), while the off-diagonal values represent misclassifications. The heatmap visually highlights how many ”Dog” and ”Not a dog” instances were correctly or incorrectly predicted by the model, making it easier to evaluate accuracy and error patterns.



5.2 Confusion Matrix and Classification Report Evaluation

Explanation

This code evaluates a classification model using the `classification_report` function, which provides precision, recall, and F1-score for each class by comparing the true labels (`truth`) with the predicted labels (`prediction`).

Additionally, the F1-score is manually computed for two classes: "Dog" and "not a Dog" using the harmonic mean formula:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

For the "Dog" class, precision is 0.57 and recall is 0.67. For the "not a Dog" class, precision is 0.33 and recall is 0.25. These calculations help measure the balance between precision and recall for each class.

Output

The output includes a full classification report showing precision, recall, F1-score, and support for each class. It indicates how well the model distinguishes between "Dog" and "not a Dog" instances. The manually computed F1-scores further confirm model performance: the "Dog" class achieves a moderate F1-score (0.61), while the "not a Dog" class shows a low F1-score (0.29), indicating weaker predictive performance for that class.

[26]:
```
print(classification_report(truth, prediction))
```

```
              precision    recall  f1-score   support

         Dog       0.57      0.67      0.62         6
   Not a dog       0.33      0.25      0.29         4

    accuracy                           0.50        10
   macro avg       0.45      0.46      0.45        10
weighted avg       0.48      0.50      0.48        10
```

5.3 f1 score for “Dog” class

[27]:
```
2*(0.57*0.67/(0.57+0.67))
```

[27]: 0.6159677419354839

5.4 f1 score for “not a Dog” class

[28]:
```
2*(0.33*0.25/(0.33+0.25))
```

[28]: 0.2844827586206896