NAME: DANIYAL SAEED

SAP ID: 53937

SECTION: BS Data Science

COURSE: AI

Question no 1

A. Presentation Skills (Scores 0-100)

Step	Score Data (Sorted)	Calculation & Result
1. Find Q2 (Median)	30, 35, 76, 78, 79, 80, 81, **83**, 83, 84, 92, 92, 95, 99, 100 ($n=15$)	Q_2 position is $\frac{15+1}{2} = 8$. The 8th score is **83**.
2. Find Q1 (Lower Half Median)	*Lower Half:* 30, 35, 76, 78, 79, 80, 81 ($n=7$)	Q_1 position is $\frac{7+1}{2} = 4$. The 4th score is **78**.
3. Find Q3 (Upper Half Median)	*Upper Half:* 83, 84, 92, 92, 95, 99, 100 ($n=7$)	Q_3 position is $\frac{7+1}{2} = 4$. The 4th score is **92**.
4. Calculate IQR	$\text{IQR} = Q_3 - Q_1$	$\text{IQR} = 92 - 78 = **14**$
5. Find Boundaries	$1.5 \times \text{IQR} = 1.5 \times 14 = 21$	Lower: $78 - 21 = **57**$ Upper: $92 + 21 = **113**$
6. Identify Outliers	Check scores against the range [57, 113]	Scores below 57: **30, 35** Scores above 113: **None**
Outliers		**30, 35**

B. Development Skills (Scores 0-100)

C. Topic Strength (Scores 0-100)

Step	Score Data (Sorted)	Calculation & Result
1. Find Q2 (Median)	31, 40, 86, 86, 87, 87, 88, **89**, 90, 90, 92, 92, 95, 99, 99 ($n=15$)	$Q_2$ position is $\frac{15+1}{2} = 8$. The 8th score is **89**.
2. Find Q1 (Lower Half Median)	*Lower Half:* 31, 40, 86, 86, 87, 87, 88 ($n=7$)	$Q_1$ position is $\frac{7+1}{2} = 4$. The 4th score is **86**.
3. Find Q3 (Upper Half Median)	*Upper Half:* 90, 90, 92, 92, 95, 99, 99 ($n=7$)	$Q_3$ position is $\frac{7+1}{2} = 4$. The 4th score is **92**.
4. Calculate IQR	$\text{IQR} = Q_3 - Q_1$	$\text{IQR} = 92 - 86 = **6**$
5. Find Boundaries	$1.5 \times \text{IQR} = 1.5 \times 6 = 9$	Lower: $86 - 9 = **77**$ Upper: $92 + 9 = **101**$
6. Identify Outliers	Check scores against the range [77, 101]	Scores below 77: **31, 40** Scores above 101: **None**
Outliers		**31, 40**

Implementation of out lier

```
import numpy as np

# Define the datasets
data = {
    "Presentation Skills": [81, 79, 78, 83, 95, 100, 92, 84, 83, 76, 99, 92, 35, 30, 80],
    "Development Skills": [82, 79, 78, 82, 85, 100, 88, 84, 83, 77, 99, 95, 40, 40, 81],
    "Topic Strength": [88, 87, 86, 89, 95, 99, 92, 90, 90, 86, 99, 92, 40, 31, 87]
}

def find_outliers_iqr_clean(data_set, name):

    data_array = np.array(data_set)

    # Calculate Q1 (25th) and Q3 (75th)
    Q1 = np.percentile(data_array, 25)
    Q3 = np.percentile(data_array, 75)

    # Calculate IQR and Fences
    IQR = Q3 - Q1
    lower_fence = Q1 - (1.5 * IQR)
    upper_fence = Q3 + (1.5 * IQR)

    # Identify Outliers
    outliers = data_array[(data_array < lower_fence) | (data_array > upper_fence)]

    # Print only the final results without emojis
    print(f"--- {name} Results ---")

    if len(outliers) > 0:
        print(f"Outliers: {list(outliers)}")
    else:
        print("No Outliers Detected.")
    print("-" * (len(name) + 16))

# Run the function for all three datasets
for name, data_set in data.items():
    find_outliers_iqr_clean(data_set, name)
    print("\n")
```

Question no 2

Code for Knn

```
TRAIN_SAMPLES = [
[1, 350, 22, "Wheat"], [2, 600, 30, "Rice"], [1, 400, 24, "Wheat"],
[2, 750, 32, "Rice"], [1, 370, 23, "Wheat"], [2, 780, 33, "Rice"],
[1, 420, 25, "Wheat"], [2, 700, 31, "Rice"], [1, 390, 22, "Wheat"],
[2, 770, 30, "Rice"]
]TEST_SAMPLES = [
{"name": "Test Sample 1", "data": [1, 360, 22]},
{"name": "Test Sample 2", "data": [2, 720, 31]}
]
#Preprocessing Min-Max Normalization
def normalize_data(data):
soil_types = [d[0] for d in TRAIN_SAMPLES]
rainfall = [d[1] for d in TRAIN_SAMPLES]
```

```
temp = [d[2] for d in TRAIN_SAMPLES]
min_vals = [min(soil_types), min(rainfall), min(temp)]
max_vals = [max(soil_types), max(rainfall), max(temp)]
#Store the normalization values
global MIN_MAX_VALUES
MIN_MAX_VALUES = {"min": min_vals, "max": max_vals}
normalized_point = []
for i in range(len(data)):
feature = data[i]
min_v = min_vals[i]
max_v = max_vals[i]
if max_v == min_v:
normalized_point.append(0.0)
else:
normalized_value = (feature - min_v) / (max_v - min_v)
normalized_point.append(normalized_value)return normalized_point
NORMALIZED_TRAIN_SAMPLES = []
for sample in TRAIN_SAMPLES:
features = sample[:3] # [Soil, Rain, Temp]
crop = sample[3]
normalized_features = normalize_data(features)
NORMALIZED_TRAIN_SAMPLES.append(normalized_features + [crop])
# Distance Calculation
def euclidean_distance(point1, point2):
squared_sum = 0
for i in range(len(point1)):
squared_sum += (point1[i] - point2[i]) ** 2
return squared_sum ** 0.5 # Square root of the sum
def predict_knn(test_point, k=3):
test_point_norm = normalize_data(test_point)
distances = []
for train_sample in NORMALIZED_TRAIN_SAMPLES:
train_features = train_sample[:3]
```

```
train_crop = train_sample[3]
dist = euclidean_distance(test_point_norm, train_features)
distances.append((dist, train_crop)) # Store (distance, label) tuple
# Sort the distances and take the top K (K=3)
distances.sort(key=lambda x: x[0])
k_nearest = distances[:k]vote_count = {}
for dist, crop in k_nearest:
vote_count[crop] = vote_count.get(crop, 0) + 1
prediction = max(vote_count, key=vote_count.get)
return prediction, k_nearest, test_point_norm
print("--- K-NN (K=3) Crop Recommendation ---")
for test_sample in TEST_SAMPLES:
test_data = test_sample["data"]
test_name = test_sample["name"]
prediction, neighbors, norm_data = predict_knn(test_data, k=3)
print(f"\n{test_name} ({test_data}):")
print(f" Normalized Data: {norm_data}")
print(f" Top 3 Neighbors:")
# Print neighbors for dry-run verification
for dist, crop in neighbors:
print(f" - Crop: {crop}, Distance: {dist:.4f}")
print(f"\n Final Prediction: {prediction}")
print("-" * 30)
```

Question no 3

K-Nearest Centroid (kNC) - Predicts based on the closest class average.

```
TRAIN_SAMPLES = [
[1, 350, 22, "Wheat"], [2, 600, 30, "Rice"], [1, 400, 24, "Wheat"],
[2, 750, 32, "Rice"], [1, 370, 23, "Wheat"], [2, 780, 33, "Rice"],
[1, 420, 25, "Wheat"], [2, 700, 31, "Rice"], [1, 390, 22, "Wheat"],
[2, 770, 30, "Rice"]
]
```

```
TEST_SAMPLES = [
{"name": "Test Sample 1", "data": [1, 360, 22]},
{"name": "Test Sample 2", "data": [2, 720, 31]}
]
def get_min_max_values(data):
soil = [d[0] for d in data]
rain = [d[1] for d in data]
temp = [d[2] for d in data]
return [min(soil), min(rain), min(temp)], \
[max(soil), max(rain), max(temp)]
def normalize(data, min_vals, max_vals):
norm_point = []
for i in range(len(data)):
min_v = min_vals[i]
max_v = max_vals[i]
value = data[i]
if max_v == min_v:
norm_point.append(0.0)
else:norm_point.append((value - min_v) / (max_v - min_v))
return norm_point
def euclidean_distance(p1, p2):
return sum([(p1[i] - p2[i]) ** 2 for i in range(len(p1))]) ** 0.5
def calculate_centroids(train_samples):
class_data = {"Wheat": [], "Rice": []}
for sample in train_samples:
class_data[sample[3]].append(sample[:3])
centroids = {}
for crop, feature_lists in class_data.items():
if not feature_lists: continue
num_samples = len(feature_lists)
sum_features = [sum(f[i] for f in feature_lists) for i in range(3)]
centroids[crop] = [s / num_samples for s in sum_features]
return centroids
```

```
def predict_knc(test_point):
min_vals, max_vals = get_min_max_values(TRAIN_SAMPLES)
centroids = calculate_centroids(TRAIN_SAMPLES)
norm_test = normalize(test_point, min_vals, max_vals)
distances = {}
for crop, centroid in centroids.items():
norm_centroid = normalize(centroid, min_vals, max_vals)
distances[crop] = euclidean_distance(norm_test, norm_centroid)
prediction = min(distances, key=distances.get)
return prediction, distancesprint("--- K-Nearest CENTROID (kNC)
Results ---")
for ts in TEST_SAMPLES:
prediction, distances = predict_knc(ts["data"])
print(f"\n{ts['name']} ({ts['data']}):")
for crop, dist in distances.items():
print(f" Distance to {crop} Centroid: {dist:.4f}")
print(f" Predicted Crop: {prediction}")
print("-" * 40)
```

KND implementation

```
import numpy as np
TRAIN_SAMPLES = np.array([
[1, 350, 22, "Wheat"], [2, 600, 30, "Rice"], [1, 400, 24, "Wheat"],
[2, 750, 32, "Rice"], [1, 370, 23, "Wheat"], [2, 780, 33, "Rice"],
[1, 420, 25, "Wheat"], [2, 700, 31, "Rice"], [1, 390, 22, "Wheat"],
[2, 770, 30, "Rice"]
], dtype=object)
TEST_SAMPLES = [
{"name": "Test Sample 1", "data": np.array([1, 360, 22], dtype=float)},
{"name": "Test Sample 2", "data": np.array([2, 720, 31], dtype=float)}
]
class KNNClassifier:
def __init__(self, k=3):
```

```
self.k = k
self.X_train = None
self.y_train = None
self.min_vals = Noneself.max_vals = None
self.range_vals = None
def fit(self, X, y):
self.X_train = X.astype(float)
self.y_train = y
self.min_vals = np.min(self.X_train, axis=0)
self.max_vals = np.max(self.X_train, axis=0)
self.range_vals = self.max_vals - self.min_vals
self.range_vals[self.range_vals == 0] = 1
self.X_train_norm = (self.X_train - self.min_vals) / self.range_vals
def _normalize_point(self, x):
return (x - self.min_vals) / self.range_vals
def predict(self, test_point):
if self.X_train is None:
raise Exception("Model must be fitted before prediction.")
norm_test = self._normalize_point(test_point)
distances = np.sqrt(np.sum((self.X_train_norm - norm_test)**2,
axis=1))
distance_label_pairs = sorted(zip(distances, self.y_train),
key=lambda x: x[0])
k_nearest = distance_label_pairs[:self.k]
neighbor_labels = [label for dist, label in k_nearest]vote_count = {}
for label in neighbor_labels:
vote_count[label] = vote_count.get(label, 0) + 1
prediction = max(vote_count, key=vote_count.get)
return prediction, k_nearest
if __name__ == "__main__":
X = TRAIN_SAMPLES[:, 0:3].astype(float)
y = TRAIN_SAMPLES[:, 3]
classifier = KNNClassifier(k=3)
```

```
classifier.fit(X, y)
print("--- KNN Classifier (Decent Code, K=3) Results ---")
for ts in TEST_SAMPLES:
prediction, neighbors = classifier.predict(ts["data"])
print(f"\n{ts['name']} ({ts['data']}):")
print(f" Top {classifier.k} Neighbors:")
for dist, crop in neighbors:
print(f" - Crop: {crop}, Distance: {dist:.4f}")
print(f" Predicted Crop: {prediction}")
print("-" * 40)
```

Question no 4

ID3 implementation in python

```
import numpy as np
import math
import json
DATASET = {
'Outlook': ['Sunny', 'Sunny', 'Overcast', 'Rain', 'Rain', 'Rain',
'Overcast', 'Sunny', 'Sunny', 'Rain', 'Sunny', 'Overcast', 'Overcast', 'Rain'],
'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Cool', 'Cool', 'Cool', 'Mild',
'Cool', 'Mild', 'Mild', 'Mild', 'Hot', 'Mild'],
'Humidity': ['High', 'High', 'High', 'High', 'Normal', 'Normal',
'Normal', 'High', 'Normal', 'Normal', 'Normal', 'High', 'Normal', 'High'],
'Wind': ['Weak', 'Strong', 'Weak', 'Weak', 'Weak', 'Strong', 'Strong',
'Weak', 'Weak', 'Weak', 'Strong', 'Strong', 'Weak', 'Strong'],
'PlayTennis': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes',
'Yes', 'Yes', 'Yes', 'No']
}
RECORDS = []
feature_names = list(DATASET.keys())[:-1]
target_name = list(DATASET.keys())[-1]
for i in range(len(DATASET['Outlook'])):
```

```
record = {name: DATASET[name][i] for name in feature_names}
record[target_name] = DATASET[target_name][i]
RECORDS.append(record)
def calculate_entropy(data):
if not data:
return 0.0target_values = [d[target_name] for d in data]
unique_classes, counts = np.unique(target_values,
return_counts=True)
probabilities = counts / len(target_values)
entropy = -np.sum(probabilities * np.log2(probabilities))
return entropy
def calculate_information_gain(data, feature):
parent_entropy = calculate_entropy(data)
feature_values = [d[feature] for d in data]
unique_values = np.unique(feature_values)
total_samples = len(data)
weighted_entropy = 0
for value in unique_values:
subset = [d for d in data if d[feature] == value]
weight = len(subset) / total_samples
subset_entropy = calculate_entropy(subset)
weighted_entropy += weight * subset_entropy
information_gain = parent_entropy - weighted_entropy
return information_gain
def get_majority_class(data):
target_values = [d[target_name] for d in data]
unique, counts = np.unique(target_values, return_counts=True)
majority_index = np.argmax(counts)
return unique[majority_index]
def build_tree(data, features):
target_values = [d[target_name] for d in data]
unique_targets = np.unique(target_values)
if len(unique_targets) == 1:return unique_targets[0]
if not features:
return get_majority_class(data)
best_feature = None
max_gain = -1.0
for feature in features:
gain = calculate_information_gain(data, feature)
if gain > max_gain:
max_gain = gain
best_feature = feature
if max_gain == 0:
return get_majority_class(data)
tree = {best_feature: {}}
best_feature_values = np.unique([d[best_feature] for d in data])
remaining_features = [f for f in features if f != best_feature]
for value in best_feature_values:
subset = [d for d in data if d[best_feature] == value]
subtree = build_tree(subset, remaining_features)
tree[best_feature][value] = subtree
return tree
def print_tree(tree):
return json.dumps(tree, indent=4)
def predict(tree, sample):if not isinstance(tree, dict):
return tree
root_feature = list(tree.keys())[0]
feature_value = sample.get(root_feature)
if feature_value in tree[root_feature]:
next_tree = tree[root_feature][feature_value]
return predict(next_tree, sample)
else:
return "Unseen Value: Majority Class (Yes/No)"
if __name__ == '__main__':
decision_tree = build_tree(RECORDS, feature_names)
print("--- ID3 Decision Tree (Play Tennis) - NumPy Optimized ---")
print("\nDecision Tree Structure (JSON Output):")
print(print_tree(decision_tree))
print("\n--- Dry Run Predictions ---")
test_sample_1 = {'Outlook': 'Sunny', 'Temperature': 'Mild',
'Humidity': 'High', 'Wind': 'Weak'}
pred_1 = predict(decision_tree, test_sample_1)
print(f"Sample 1: {test_sample_1} -> Prediction: {pred_1}")
test_sample_2 = {'Outlook': 'Rain', 'Temperature': 'Cool', 'Humidity':
'Normal', 'Wind': 'Weak'}
pred_2 = predict(decision_tree, test_sample_2)
print(f"Sample 2: {test_sample_2} -> Prediction: {pred_2}")test_sample_3
= {'Outlook': 'Overcast', 'Temperature': 'Hot',
'Humidity': 'High', 'Wind': 'Strong'}
pred_3 = predict(decision_tree, test_sample_3)
print(f"Sample 3: {test_sample_3} -> Prediction: {pred_3}")
print("-" * 50)
```

Question no 5

ID3 Decision Tree

```
def log2(x):
if x <= 0:
return 0
return __import__('math').log2(x)
DATASET_RECORDS = [
{'Age': 'young', 'Has_Job': 'false', 'Own_House': 'false',
'Credit_Rating': 'fair', 'Class': 'No'},
{'Age': 'young', 'Has_Job': 'false', 'Own_House': 'false',
'Credit_Rating': 'good', 'Class': 'No'},
{'Age': 'young', 'Has_Job': 'true', 'Own_House': 'false',
'Credit_Rating': 'good', 'Class': 'Yes'},
{'Age': 'young', 'Has_Job': 'true', 'Own_House': 'false',
'Credit_Rating': 'fair', 'Class': 'Yes'},
{'Age': 'young', 'Has_Job': 'false', 'Own_House': 'false',
'Credit_Rating': 'fair', 'Class': 'No'},
{'Age': 'middle', 'Has_Job': 'false', 'Own_House': 'false',
'Credit_Rating': 'fair', 'Class': 'No'},
{'Age': 'middle', 'Has_Job': 'false', 'Own_House': 'false',
'Credit_Rating': 'good', 'Class': 'No'},{'Age': 'middle', 'Has_Job': 'true',
'Own_House': 'true',
'Credit_Rating': 'good', 'Class': 'Yes'},
{'Age': 'middle', 'Has_Job': 'false', 'Own_House': 'true',
'Credit_Rating': 'excellent', 'Class': 'Yes'},
{'Age': 'middle', 'Has_Job': 'true', 'Own_House': 'true',
'Credit_Rating': 'excellent', 'Class': 'Yes'},
{'Age': 'old', 'Has_Job': 'false', 'Own_House': 'true', 'Credit_Rating':
'excellent', 'Class': 'Yes'},
{'Age': 'old', 'Has_Job': 'false', 'Own_House': 'true', 'Credit_Rating':
'good', 'Class': 'Yes'},
{'Age': 'old', 'Has_Job': 'true', 'Own_House': 'false', 'Credit_Rating':
'good', 'Class': 'Yes'},
{'Age': 'old', 'Has_Job': 'true', 'Own_House': 'false', 'Credit_Rating':
'excellent', 'Class': 'Yes'},
{'Age': 'old', 'Has_Job': 'false', 'Own_House': 'false', 'Credit_Rating':
'fair', 'Class': 'No'}
]
RECORDS = DATASET_RECORDS
feature_names = ['Age', 'Has_Job', 'Own_House', 'Credit_Rating']
target_name = 'Class'
def get_counts(data, attribute_name):
counts = {}
values = [d[attribute_name] for d in data]
for value in values:
counts[value] = counts.get(value, 0) + 1
return counts
```

```
def calculate_entropy(data):
if not data:
return 0.0target_counts = get_counts(data, target_name)
total_samples = len(data)
entropy = 0.0
for count in target_counts.values():
probability = count / total_samples
if probability > 0:
entropy -= probability * log2(probability)
return entropy
def calculate_information_gain(data, feature):
parent_entropy = calculate_entropy(data)
feature_counts = get_counts(data, feature)
total_samples = len(data)
weighted_entropy = 0
for value, count in feature_counts.items():
subset = [d for d in data if d[feature] == value]
weight = count / total_samples
subset_entropy = calculate_entropy(subset)
weighted_entropy += weight * subset_entropy
information_gain = parent_entropy - weighted_entropy
return information_gain
def get_majority_class(data):
target_counts = get_counts(data, target_name)
majority_class = None
max_count = -1
for cls, count in target_counts.items():
if count > max_count:
max_count = countmajority_class = cls
return majority_class
def build_tree(data, features):
target_counts = get_counts(data, target_name)
unique_targets = list(target_counts.keys())
```

```
if len(unique_targets) == 1:
return unique_targets[0]
if not features:
return get_majority_class(data)
best_feature = None
max_gain = -1.0
for feature in features:
gain = calculate_information_gain(data, feature)
if gain > max_gain:
max_gain = gain
best_feature = feature
if max_gain == 0:
return get_majority_class(data)
tree = {best_feature: {}}
best_feature_values = list(get_counts(data, best_feature).keys())
remaining_features = [f for f in features if f != best_feature]
for value in best_feature_values:
subset = [d for d in data if d[best_feature] == value]
subtree = build_tree(subset, remaining_features)tree[best_feature][value]
= subtree
return tree
def print_tree(tree, indent=''):
if not isinstance(tree, dict):
print(f": {tree}")
return
root_feature = list(tree.keys())[0]
print(f"{indent}-> {root_feature}")
for value, subtree in tree[root_feature].items():
print(f"{indent} {root_feature} = {value}", end="")
if isinstance(subtree, dict):
print()
print_tree(subtree, indent + ' ')
else:
```

```
print(f": {subtree}")
def predict(tree, sample):
if not isinstance(tree, dict):
return tree
root_feature = list(tree.keys())[0]
feature_value = sample.get(root_feature)
if feature_value in tree[root_feature]:
next_tree = tree[root_feature][feature_value]
return predict(next_tree, sample)
else:
return "Unknown"if __name__ == '__main__':
decision_tree = build_tree(RECORDS, feature_names)
print("--- ID3 Decision Tree (Question #5 Dataset) - NO IMPORTS -
--")
print("\nDecision Tree Structure (Custom Output):")
print_tree(decision_tree)
print("\n--- Dry Run Predictions ---")
test_sample_1 = {'Age': 'young', 'Has_Job': 'true', 'Own_House':
'false', 'Credit_Rating': 'fair'}
pred_1 = predict(decision_tree, test_sample_1)
print(f"Sample 1: {test_sample_1} -> Prediction: {pred_1}")
test_sample_2 = {'Age': 'middle', 'Has_Job': 'false', 'Own_House':
'false', 'Credit_Rating': 'fair'}
pred_2 = predict(decision_tree, test_sample_2)
print(f"Sample 2: {test_sample_2} -> Prediction: {pred_2}")
test_sample_3 = {'Age': 'old', 'Has_Job': 'true', 'Own_House': 'true',
'Credit_Rating': 'excellent'}
pred_3 = predict(decision_tree, test_sample_3)
print(f"Sample 3: {test_sample_3} -> Prediction: {pred_3}")
print("-" * 50)
```

Question no 6

```
# This script calculates Precision, Recall, F1-Score, and Overall
Accuracy
# from scratch for two confusion matrices, as requested.
# --- DATA DEFINITION ---
# Matrix 1: Binary Classification (Cancerous vs. Normal)
# Row 0: True Cancerous (P) | Row 1: True Normal (N)
# Column 0: Predicted Cancerous | Column 1: Predicted Normal
MATRIX_1_LABELS = ['Cancerous', 'Normal']
MATRIX_1 = [
[762, 15], # 762 TP, 15 FN
[10, 93] # 10 FP, 93 TN
]
# Matrix 2: Multi-Class Classification (7 classes)
# Rows are True Labels, Columns are Predicted Labels
MATRIX_2_LABELS = ['N', 'PB', 'UDH', 'ADH', 'FEA', 'DCIS', 'IC']
MATRIX_2 = [
[87, 5, 4, 3, 1, 3, 0],
[12, 121, 5, 3, 3, 2, 1],
[4, 2, 75, 6, 1, 3, 2],
[7, 0, 12, 79, 2, 13, 1],
[1, 3, 7, 5, 148, 4, 0],
[1, 7, 9, 6, 4, 125, 1],
[0, 1, 3, 2, 1, 5, 101]
]
# --- CORE CALCULATION FUNCTIONS (NO LIBRARIES) ---
def safe_divide(numerator, denominator):
"""Handles division by zero."""
return numerator / denominator if denominator else 0.0def
calculate_metrics_for_class(matrix, class_index):
"""
```

```
Calculates TP, FP, FN, Precision, Recall, and F1-Score for a single
class (one-vs-all approach).
"""
N = len(matrix)
TP = matrix[class_index][class_index]
# FN (False Negatives) = Sum of all elements in the true class row,
excluding TP
FN = sum(matrix[class_index]) - TP
# FP (False Positives) = Sum of all elements in the predicted class
column, excluding TP
FP = sum(matrix[i][class_index] for i in range(N)) - TP
# Precision: TP / (TP + FP)
precision = safe_divide(TP, TP + FP)
# Recall: TP / (TP + FN)
recall = safe_divide(TP, TP + FN)
# F1-Score: 2 * (Precision * Recall) / (Precision + Recall)
f1_score = safe_divide(2 * precision * recall, precision + recall)
return TP, FP, FN, precision, recall, f1_score
def calculate_accuracy(matrix):
"""
Calculates Overall Accuracy: (Sum of Diagonals) / (Total Samples)
"""
N = len(matrix)total_samples = sum(sum(row) for row in matrix)
correct_predictions = sum(matrix[i][i] for i in range(N)) # Sum of
diagonal (TPs)
return safe_divide(correct_predictions, total_samples),
total_samples
# --- DRY RUN AND OUTPUT ---
if __name__ == '__main__':
# --------------------------------------------------------------------
#
MATRIX 1 (BINARY)
# --------------------------------------------------------------------
```

```
print("==============================================
==============================")
print("STEP-BY-STEP DRY RUN: MATRIX 1 (Cancerous/Normal)")
print("==============================================
==============================")
accuracy_m1, total_m1 = calculate_accuracy(MATRIX_1)
print(f"\n1. Overall Accuracy Calculation:")
print(f" Correct Predictions (TP+TN): {MATRIX_1[0][0]} +
{MATRIX_1[1][1]} = 855")
print(f" Total Samples: {total_m1}")
print(f" Accuracy: 855 / {total_m1} = {accuracy_m1:.4f}")
print("\n2. Metrics for 'Cancerous' Class (Positive Class):")
TP, FP, FN, P, R, F1 = calculate_metrics_for_class(MATRIX_1, 0) #
Index 0 is 'Cancerous'print(f" - True Positives (TP): {TP}")
print(f" - False Positives (FP): {FP} (10)")
print(f" - False Negatives (FN): {FN} (15)")
print(f" - Precision: {TP} / ({TP} + {FP}) = 762 / 772 = {P:.4f}")
print(f" - Recall: {TP} / ({TP} + {FN}) = 762 / 777 = {R:.4f}")
print(f" - F1-Score: 2 * ({P:.4f} * {R:.4f}) / ({P:.4f} + {R:.4f}) = {F1:.4f}")
print("\n3. Metrics for 'Normal' Class (Negative Class):")
TP_N, FP_N, FN_N, P_N, R_N, F1_N =
calculate_metrics_for_class(MATRIX_1, 1) # Index 1 is 'Normal'
print(f" - True Positives (TP): {TP_N}")
print(f" - False Positives (FP): {FP_N} (15)")
print(f" - False Negatives (FN): {FN_N} (10)")
print(f" - Precision: {TP_N} / ({TP_N} + {FP_N}) = 93 / 108 =
{P_N:.4f}")
print(f" - Recall: {TP_N} / ({TP_N} + {FN_N}) = 93 / 103 = {R_N:.4f}")
print(f" - F1-Score: 2 * ({P_N:.4f} * {R_N:.4f}) / ({P_N:.4f} + {R_N:.4f})
= {F1_N:.4f}")
# ------------------------------------------------------------------
#
MATRIX 2 (MULTI-CLASS)
```

```
# -----------------------------------------------------------------
print("\n\n==========================================
==============================")
print("STEP-BY-STEP DRY RUN: MATRIX 2 (Multi-Class)")
print("==========================================
=============================")accuracy_m2, total_m2 =
calculate_accuracy(MATRIX_2)
print(f"\n1. Overall Accuracy Calculation:")
print(f" Correct Predictions (Diagonal Sum): 87 + 121 + 75 + 79 +
148 + 125 + 101 = 736")
print(f" Total Samples: {total_m2}")
print(f" Accuracy: 736 / {total_m2} = {accuracy_m2:.4f}")
print("\n2. Metrics for Each Class (One-vs-All Approach):")
results = {}
for i, label in enumerate(MATRIX_2_LABELS):
TP, FP, FN, P, R, F1 = calculate_metrics_for_class(MATRIX_2, i)
results[label] = {'TP': TP, 'FP': FP, 'FN': FN, 'P': P, 'R': R, 'F1': F1}
print(f"\n--- Class: {label} ---")
print(f" - TP: {TP} | FP: {FP} | FN: {FN}")
print(f" - Precision: {P:.4f}")
print(f" - Recall: {R:.4f}")
print(f" - F1-Score: {F1:.4f}")
print("\n==========================================
=============================")
print("SUMMARY OF RESULTS")
print("==========================================
============================")
print(f"| {'Class':<10} | {'Precision':<10} | {'Recall':<10} | {'F1-
Score':<10} |")
print("-" * 47)# Print Matrix 1 Summary
P1, R1, F1_1 = results['Cancerous']['P'], results['Cancerous']['R'],
results['Cancerous']['F1']
print(f"| {'Cancerous':<10} | {P1:<10.4f} | {R1:<10.4f} | {F1_1:<10.4f}
|")
P2, R2, F1_2 = results['Normal']['P'], results['Normal']['R'],
results['Normal']['F1']
print(f"| {'Normal':<10} | {P2:<10.4f} | {R2:<10.4f} | {F1_2:<10.4f} |")
print("-" * 47)
print(f"Overall Accuracy (M1): {accuracy_m1:.4f}")
print("\n\n| {'Class':<10} | {'Precision':<10} | {'Recall':<10} | {'F1-
Score':<10} |")
print("-" * 47)
# Print Matrix 2 Summary
for label in MATRIX_2_LABELS:
res = results[label]
print(f"| {label:<10} | {res['P']:<10.4f} | {res['R']:<10.4f} |
{res['F1']:<10.4f} |")
print("-" * 47)
print(f"Overall Accuracy (M2): {accuracy_m2:.4f}")
```