Sierpiński Triangle–Based Fractal Features for Osteoporosis Assessment in Medical Imaging

Shriverdhan Pathak[1]
2025PGCSDS01
NIT Jamshedpur
2025pgcsds01@nitjsr.ac.in

Satyam Mishra[2]
2025PGCSDS06
NIT Jamshedpur
2025pgcsds06@nitjsr.ac.in

Bhumika Verma[3]
2025PGCSDS07
NIT Jamshedpur
2025pgcsds07@nitjsr.ac.in

Saurabh Verma[4]
2025PGCSDS13
NIT Jamshedpur
2025pgcsds13@nitjsr.ac.in

Amit Yadav[5]
2025PGCSDS14
NIT Jamshedpur
2025pgcsds14@nitjsr.ac.in

Sayan Mondal[6]
2025PGCSDS17
NIT Jamshedpur
2025pgcsds17@nitjsr.ac.in

Abstract

This study presents a complete machine-learning pipeline for osteoporosis classification using X-ray images and fractal-based feature engineering. Each image is first preprocessed through white-pixel suppression (≥240 intensity), square padding, and high-quality resizing to **512×512** pixels to ensure uniform structure. A geometric framework is then applied, where a regular hexagon is mapped onto the image and **level-3 Sierpiński triangles** are recursively generated to create multiple triangular masks. From each triangular region, **five statistical features**—mean, median, standard deviation, skewness, and kurtosis—are extracted, producing **391 fractal-texture features per image**. A dataset of **616 images** across three classes (Normal, Osteopenia, Osteoporosis) is prepared and converted into a wide-format feature matrix.

To evaluate the discriminative power of these fractal features, **ten different machine-learning models** are trained, including SVM - Linear, Polynomial, Sigmoid, RBF, Logistic Regression, KNN, Naive Bayes, Decision Tree, Random Forest, and Gradient Boosting. Multiple train–test splits (60/40, 70/30, 80/20, 90/10) are tested with stratified sampling, and scaling is applied where necessary. The results show that Sierpiński-triangle-based fractal features capture important trabecular texture variations and enable effective automated classification of osteoporosis severity using standard X-ray images.

Keywords

Osteoporosis, Medical Imaging, X-ray Analysis, Fractal Features, Sierpiński Triangle, Hexagonal Masking, Image Preprocessing, Feature Extraction, Machine Learning, Support Vector Machine (SVM), KNN, Decision Tree, Bone Health Classification, Trabecular Texture.

Problem Statement

Diagnosing osteoporosis usually relies on advanced imaging tools such as DEXA scanners, which are not always accessible in every healthcare environment. Although standard X-ray images contain valuable details about the internal texture of trabecular bone, these patterns cannot be interpreted accurately without computational techniques. At present, there is a clear need for an automated and lightweight system capable of extracting meaningful texture information from X-rays and accurately determining bone-health conditions.

In this study, the objective is:

To develop a computational approach that captures detailed bone-texture characteristics from ordinary X-ray images using fractal geometry—specifically Sierpiński-triangle–based feature extraction—and classifies the samples into Normal, Osteopenia, and Osteoporosis using an optimized machine-learning models.

Methodology

1. Image Preprocessing

The preprocessing pipeline ensures that all X-ray images are standardized before feature extraction. It includes the following steps:

1.1 White Pixel Removal

X-ray images often contain extremely bright (white) areas that do not contribute to bone-texture information. These pixels (intensity ≥ 240) are replaced with black to reduce noise and enhance the clarity of the trabecular region. This is achieved using a thresholding operation applied to the grayscale array of the image.

1.2 Shape Normalization (Square Padding)

Since X-ray samples have varying widths and heights, each image is padded with black pixels to form a perfect square. The longer dimension (height or width) determines the final size, and the smaller dimension is padded symmetrically on both sides. This ensures consistent geometry for mask generation.

1.3 Standardized Resizing

All square images are resized to 512 × 512 pixels using the high-quality LANCZOS resampling technique. This reduces computational variation and ensures that all fractal masks align correctly across the dataset.

2. Geometric Framework and Fractal Mask Generation

2.1 Hexagon Construction

A regular hexagon is drawn centered at the midpoint of the image. The radius is set to 0.95 × (image_size / 2) to ensure the vertices fall inside the boundaries.
Six vertices are generated using:

$(x , y) = (cx + r\cos\theta , cy + r\sin\theta)$

where θ ranges from 0 to 2π in six equal steps

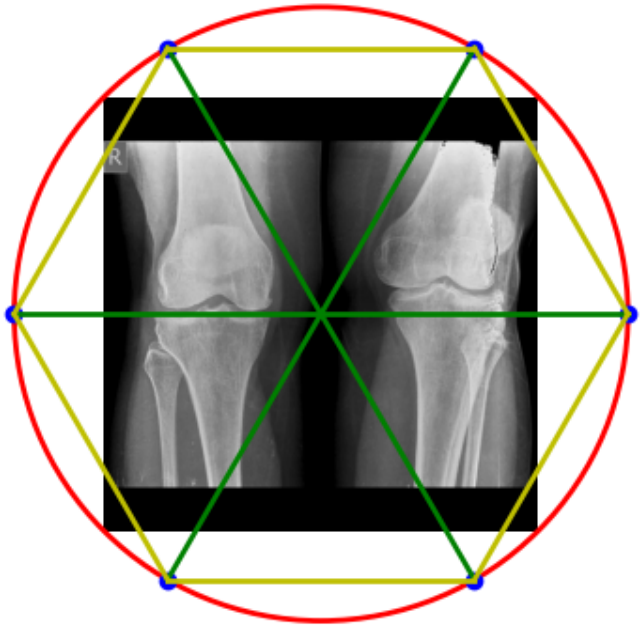


Fig. 1 : Hexagon Construction

2.2 Sierpiński Triangle Recursion (Level 3)

For each pair of adjacent hexagon vertices and the image center, a main triangle is formed. Then a recursive function generates level-3 Sierpiński triangles, using midpoint calculations:

- Level 0 → no recursion
- Level 1 → central triangle
- Level 2 → three more triangles
- Level 3 → nine fractal triangles

The code recursively computes midpoints and yields all inner triangular regions.

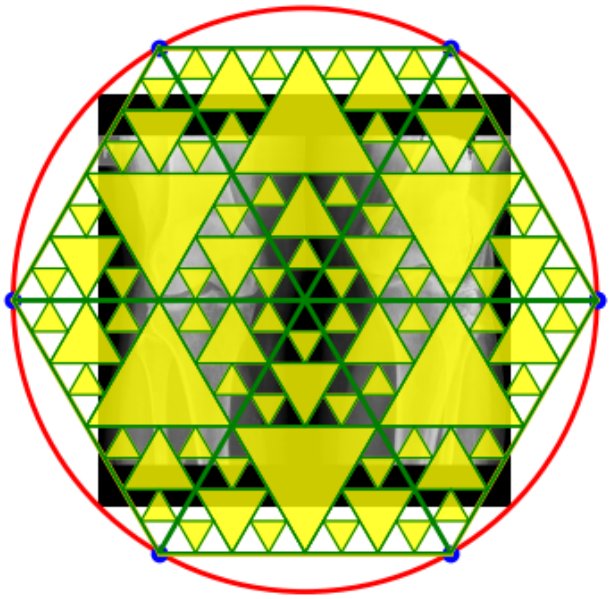


Fig. 2 : Sierpinski Triangle Recursion

2.3 Binary Mask Creation

Each Sierpiński triangle is filled to create a binary mask (True inside, False outside).

The mask size is always 512 × 512.

For each of the 6 major triangles and their recursive sub-triangles, masks are stored into a list.

A total of 78 stable masks (after recursion) are used for feature extraction.

3. Feature Extraction

3.1 Pixel Collection from Each Mask

Each mask is applied to the grayscale image, and only pixels inside the triangular regions are collected.
To avoid noise, only pixels with intensity > 0 are considered.

3.2 Statistical Measurements

For every triangular region, the following 5 statistical descriptors are computed:

- Mean
- Median
- Standard deviation
- Skewness
- Kurtosis

Custom "safe moment" functions are used to handle cases with insufficient pixels or near-zero variance.

3.3 Wide Format Dataset Creation

Features from all triangles are concatenated to produce:

- 391 features per image
- Columns named in the form → *triangle_i_mean*, *triangle_i_median*, etc.

The final dataset contains **616 rows × 391 columns**.

This dataset is exported to CSV for ML training.

4. Machine Learning Model Training

4.1 Train–Test Split

Across different experiments, the dataset is divided using **four different train–test ratios** to compare performance:

- **90/10 split**
- **80/20 split**
- **70/30 split**
- **60/40 split**

All splits use **stratified sampling** to maintain class balance.

4.2 Machine Learning Models Used

The study evaluates ten different classifiers, each trained and tested on the fractal feature dataset:

1. Support Vector Machine - Linear

2. Support Vector Machine - Sigmoid

3. Support Vector Machine - Polynomial

4. Support Vector Machine - RBF

5. Logistic Regression

6. k-Nearest Neighbors (KNN)

7. Gaussian Naive Bayes (GNB)

8. Decision Tree Classifier

9. Random Forest Classifier

10. Gradient Boosting Classifier

4.3 Model Evaluation

Each classifier is evaluated using the test split corresponding to its experiment.
Evaluation includes:

- Accuracy
- Precision, Recall, F1-score
- Confusion Matrix
- Class-wise performance for Normal, Osteopenia, and Osteoporosis

Across all trained models, Sierpiński-triangle-based fractal features demonstrated strong discriminatory capability, capturing meaningful trabecular bone-texture patterns that allow robust classification.

Solution Step Wise

1. Load dataset and configuration

- Read the feature CSV (**triangle_features_78_stable_final_file.csv**) or load images for on-the-fly feature extraction.
- Set constants : **RESIZE_DIM = (512, 512), WHITE_THRESHOLD = 240, RECURSION_LEVEL = 3.**

2. Preprocessing — white-pixel suppression

- Convert image to grayscale.
- Replace pixels with intensity ≥ **240** by **0** (black) to remove bright artifacts and background. (**remove_white_to_black** function).

3. Preprocessing — square padding & normalization

- Pad image with black pixels to make it square (**center input inside a black square**). (**pad_numpy_square_gray**).
- Resize the square image to **512×512** using LANCZOS for consistent geometry.

4. Construct geometric framework (hexagon + center)

- Compute image center **(cx, cy)** and radius ≈ **0.95 * (min(w,h)/2)**.
- Generate 6 hexagon vertices around the center (angles spaced by 60°).

5. Generate Sierpiński triangular masks (recursion)

- For each of the 6 main triangles (center + two adjacent hexagon vertices) call **generate_sierpinski_triangles(v1,v2,v3, level=3).**
- The recursive generator yields inner triangle vertex triplets (midpoint recursion).

6. Create binary masks for each subtriangle

- For every returned triangle, draw / fill polygon on a blank **512×512** mask (PIL **ImageDraw.polygon**) → boolean mask.
- Cache masks per image size to speed repeated use (**@lru_cache**). The PDF stores a stable set of masks used for feature extraction.

7. Extract pixel values inside each mask

- Apply each mask to the preprocessed image and collect non-zero pixel values (pixels > 0).
- Skip tiny regions (if nonzero pixel count < 2) or handle with safe defaults. (**extract_features** uses safe-moments).

8. Compute statistical descriptors per mask

- For each triangular region compute 5 statistics: mean, median, standard deviation, skewness, kurtosis (using **safe_moments** to avoid NaNs).

9. Assemble wide-format feature vector

- Concatenate statistics from all triangles into a single row. According to the PDF this results in 391 features per image.

- Repeat for all images to form a feature matrix; save to CSV (**triangle_features_78_stable_final_file.csv**).

10. Train–test splitting (experiments)

- Use stratified splits; the PDF experiments use multiple ratios: 90/10, 80/20, 70/30, 60/40. Choose the split(s) you want to report.

11. Scaling & model pipelines

- For scale-sensitive models (SVM, Logistic Regression, KNN) apply **StandardScaler** fitted on training set then transform test set. Tree/boosting models are trained without scaling. Use **Pipeline** for clean GridSearch/RandSearch.

12. Train & tune models (the PDF evaluates 10 models)

- Models used: SVM, Logistic Regression, KNN, Gaussian NB, Decision Tree, Random Forest, XGBoost, CatBoost, AdaBoost, Gradient Boosting.
- Example SVM tuning: test **C** in wide range (0.0001 → 100), **kernel=linear**, **class_weight** (None/“balanced”), **decision_function_shape** (ovo/ovr), use 5-fold CV / **GridSearchCV or RandomizedSearchCV**.

13. Evaluate & save results

- Compute: Accuracy, Precision / Recall / F1 (per class), Confusion Matrix, ROC/AUC (multiclass). Print training and testing reports and save best model metadata.

14. Post-analysis

- Plot learning curves, ROC curves, and error analysis per class.

Feature Extraction Code for Single Image

```
from PIL import Image

import numpy as np

import matplotlib.pyplot as plt


# --- Sierpinski Plotting Functions ---

def get_midpoint(p1, p2):

    """Calculates the midpoint of a line segment."""

    return ((p1[0] + p2[0]) / 2, (p1[1] + p2[1]) / 2)

def plot_sierpinski(v1, v2, v3, ax, level):

    """Recursively plots Sierpinski triangles."""

    if level == 0:

        return

  # Draw the triangle for the current level # meaning

  ax.plot([v1[0], v2[0], v3[0], v1[0]], [v1[1], v2[1], v3[1], v1[1]], color='green', linewidth=0.8)

  # Calculate midpoints

  m1 = get_midpoint(v1, v2)

  m2 = get_midpoint(v2, v3)

  m3 = get_midpoint(v1, v3)

  # Fill the central, "necessary" triangle at every level > 0

  ax.fill([m1[0], m2[0], m3[0],m1[0]], [m1[1], m2[1], m3[1],m1[1]], color='yellow', alpha=0.8)

  ax.plot([m1[0], m2[0], m3[0], m1[0]], [m1[1], m2[1], m3[1], m1[1]], color='green',linewidth=0.8)
```

```
# Recursively call for the 3 outer triangles
plot_sierpinski(v1, m1, m3, ax, level - 1)
plot_sierpinski(m1, v2, m2, ax, level - 1)
plot_sierpinski(m3, m2, v3, ax, level - 1)
# Load squared grayscale image
img = Image.open("final_square.jpg").convert("L")
arr = np.array(img)
h, w = arr.shape
assert h == w, "Image must be square!"
# Center of image
cx, cy = w // 2, h // 2
# Radius = half of diagonal
radius = int(np.sqrt(2) * w / 2)
# Plot image
fig, ax = plt.subplots()
ax.imshow(arr, cmap="gray")
# Draw circle with exact radius
circle = plt.Circle((cx, cy), radius, color="red", fill=False, linewidth=2)
ax.add_patch(circle)
# Angles for 6 points (radians)
angles = np.linspace(0, 2*np.pi, 7)

# Compute coordinates of the 6 points
points = [(cx + radius*np.cos(a), cy + radius*np.sin(a)) for a in angles]
```

```
# Plot the 6 points
for (x, y) in points:
    ax.plot(x, y, 'bo', markersize=6)   # blue dots
# Draw lines from center to each point
for (x, y) in points:
    ax.plot([cx, x], [cy, y], 'g-', linewidth=2)
for i in range(len(points) - 1):
    x1, y1 = points[i]
    x2, y2 = points[(i+1) % len(points)]  # wrap around to connect last→first
    ax.plot([x1, x2], [y1, y2], 'y-', linewidth=2)  #  hexagon edges
 # Expand axis limits so circle is visible
 ax.set_xlim(0 - w*0.3, w + w*0.3)
 ax.set_ylim(h + h*0.3, 0 - h*0.3)  # inverted y for image display
 ax.set_aspect('equal')
 plt.axis("off")
 # --- New Loop to Plot the Sierpinski Triangles ---
 center_point = (cx, cy)
 recursion_level = 3 # You can change this level to get more or fewer triangles


 for i in range(len(points)):
     p1 = points[i]
     p2 = points[(i + 1) % len(points)]


  # Call the recursive plotting function for each of the 6 main triangles
  plot_sierpinski(center_point, p1, p2, ax, recursion_level)
```

```
 # Show the final plot
 plt.show()
```

Sierpinski Triangle CSV Code

```
import os
import numpy as np
import pandas as pd
from PIL import Image, ImageDraw
from scipy import stats
from functools import lru_cache
from concurrent.futures import ProcessPoolExecutor, as_completed
from tqdm import tqdm # ADDED: For a helpful progress bar


def safe_moments(arr):
   if len(arr) < 2 or np.std(arr) < 1e-8:  # Too few values or almost constant
      return {"skewness": 0.0, "kurtosis": 0.0}
   return {
      "skewness": float(stats.skew(arr)),
      "kurtosis": float(stats.kurtosis(arr))
   }
# --- Configuration ---
RESIZE_DIM = (512, 512) # ADDED: Define a standard size for all images
```

```
# --- Helper Functions ---

def remove_white_to_black(img, threshold=240):

    """Replace white (>=threshold) with black in grayscale image."""

    arr = np.array(img)

    arr[arr >= threshold] = 0

    return Image.fromarray(arr)

def pad_numpy_square_gray(img):

    """Pad grayscale image with black to make it square."""

    arr = np.array(img)

    h, w = arr.shape

    size = max(h, w)

    new_arr = np.zeros((size, size), dtype=np.uint8)

    y_offset = (size - h) // 2

    x_offset = (size - w) // 2

    new_arr[y_offset:y_offset+h, x_offset:x_offset+w] = arr

    return Image.fromarray(new_arr)

def get_midpoint(p1, p2):

    """Calculates the midpoint between two points."""

    return ((p1[0] + p2[0]) / 2, (p1[1] + p2[1]) / 2)

def generate_sierpinski_triangles(v1, v2, v3, level):

    """

    Recursively generates the vertices of the 'hole' triangles.

    This function is a generator that yields the coordinates.

    """
```

```
 if level <= 0:
        return
    m1 = get_midpoint(v1, v2)
    m2 = get_midpoint(v2, v3)
    m3 = get_midpoint(v1, v3)
    yield (m1, m2, m3)
    yield from generate_sierpinski_triangles(v1, m1, m3, level - 1)
    yield from generate_sierpinski_triangles(m1, v2, m2, level - 1)
    yield from generate_sierpinski_triangles(m3, m2, v3, level - 1)
# --- Caching and Processing Functions ---
def extract_features(sub_triangle_pixels):
    """Compute stats from pixels."""
    nonzero_pixels = sub_triangle_pixels[sub_triangle_pixels > 0]
    if nonzero_pixels.size < 2:
        return {
            "mean": 0,
            "median": 0,
            "std_dev": 0,
            "skewness": np.nan,
            "kurtosis": np.nan
        }
    nonzero_pixels = nonzero_pixels.astype(np.float64)
    # Use safe moments
    moments = safe_moments(nonzero_pixels)
```

```
    return {
        "mean": float(np.mean(nonzero_pixels)),
        "median": float(np.median(nonzero_pixels)),
        "std_dev": float(np.std(nonzero_pixels)),
        "skewness": moments["skewness"],
        "kurtosis": moments["kurtosis"]
    }
@lru_cache(maxsize=5)
def get_triangle_masks(w, h, recursion_level=3):
    """Precompute masks for the 'hole' triangles for a given size."""
    cx, cy = w // 2, h // 2
    # Adjust radius slightly to ensure vertices are within the image bounds
    radius = min(w, h) // 2 * 0.95
    # Vertices of a regular hexagon
    main_points = [
        (cx + radius * np.cos(angle), cy + radius * np.sin(angle))
        for angle in np.linspace(0, 2 * np.pi, 7)
    ]
    center_point = (cx, cy)
    all_masks = []
    for i in range(6):
        p1, p2 = main_points[i], main_points[i + 1]
        sierpinski_verts = list(generate_sierpinski_triangles(center_point, p1, p2,
level=recursion_level))
        mask_set = []
```

```
 for vertices in sierpinski_verts:
        mask = Image.new("L", (w, h), 0)
        draw = ImageDraw.Draw(mask)
        # CHANGED: Simplified polygon drawing call. PIL can handle a list of tuples.
        draw.polygon(vertices, fill=1)
        mask_set.append(np.array(mask, dtype=bool))
    all_masks.append(mask_set)
return all_masks
def process_single_image(args):
    img_path, class_label, label_map = args
    try:
    img = Image.open(img_path).convert("L")
    no_white = remove_white_to_black(img, threshold=240)
    square_img = pad_numpy_square_gray(no_white)
    # ADDED: Standardize image size. This is the key fix.#check this what it does
    resized_img = square_img.resize(RESIZE_DIM, Image.Resampling.LANCZOS)
    arr = np.array(resized_img)
    h, w = arr.shape
    # Now, w and h will be constant (e.g., 512), so the cache will be hit every time.
    masks = get_triangle_masks(w, h, recursion_level=3)
    row_features = {}
    triangle_count = 0
    for mask_set in masks:
        for mask_arr in mask_set:
            sub_triangle_pixels = arr[mask_arr]
```

```
            features = extract_features(sub_triangle_pixels)
            for fname, val in features.items():
                if np.isnan(val) or np.isinf(val):
                    row_features[f"triangle_{triangle_count + 1}_{fname}"] = 0
                else:
                    row_features[f"triangle_{triangle_count + 1}_{fname}"] = val
            triangle_count += 1
        row_features["label"] = label_map.get(class_label, -1)
        return row_features
    except Exception as e:
        print(f"❌ Error processing {img_path}: {e}")
        return None # Return None on error
# --- Main Dataset Processing ---
def process_dataset_wide(root_folder, output_csv, num_workers=4):
    label_map = {"Normal": 0, "Osteoporosis": 1, "Osteopenia": 2}
    tasks = []
    for class_label in os.listdir(root_folder):
        class_folder = os.path.join(root_folder, class_label)
        if os.path.isdir(class_folder) and class_label in label_map:
            for img_name in os.listdir(class_folder):
                if img_name.lower().endswith(('.png', '.jpg', '.jpeg')):
                    img_path = os.path.join(class_folder, img_name)
                    tasks.append((img_path, class_label, label_map))
    all_rows = []
    # Use ProcessPoolExecutor within a 'with' statement for proper cleanup
    with ProcessPoolExecutor(max_workers=num_workers) as executor:
```

```
    # ADDED: Use tqdm to show a progress bar
        futures = [executor.submit(process_single_image, task) for task in tasks]
        for f in tqdm(as_completed(futures), total=len(tasks), desc="Processing Images"):
            result = f.result()
            if result: # Check if the result is not None
                all_rows.append(result)
    if not all_rows:
        print("⚠️ No features were extracted. Check dataset path and image files.")
        return
    df = pd.DataFrame(all_rows)
    df.to_csv(output_csv, index=False)
    print(f"\n✅ Wide-format features saved to {output_csv}")
# --- Execution ---
# ADDED: Essential guard for multiprocessing
if __name__ == "__main__":
    root_folder = "dataset"
    output_csv = "triangle_features_78_stable_final_file.csv"
    # It's good practice to use os.cpu_count() or a slightly lower number
    workers = max(1, os.cpu_count() - 1 if os.cpu_count() else 1)
    process_dataset_wide(root_folder, output_csv, num_workers=workers)
```

Table 1 : Performance comparison of classical ML Models (80 : 20) for three-class bone density classification

Sr. No.	ML Model	Class	Accuracy	Precision	Recall	F1 Score
1.	SVM - Linear	Normal Osteoporosis Osteopenia	68%	81% 61% 70%	57% 82% 61%	67% 70% 66%
2.	SVM - Sigmoid	Normal Osteoporosis Osteopenia	70%	76% 64% 76%	66% 80% 61%	71% 71% 68%
3.	SVM - Polynomial	Normal Osteoporosis Osteopenia	68%	79% 64% 60%	68% 80% 48%	73% 71% 54%
4.	SVM - RBF	Normal Osteoporosis Osteopenia	69%	93% 61% 64%	57% 86% 58%	70% 71% 61%
5.	KNN	Normal Osteoporosis Osteopenia	64%	67% 60% 66%	66% 63% 61%	67% 61% 63%
6.	Random Forest	Normal Osteoporosis Osteopenia	60%	71% 62% 49%	50% 61% 71%	59% 62% 58%
7.	Logistic Regression	Normal Osteoporosis Osteopenia	63%	68% 62% 55%	73% 69% 39%	70% 65% 45%
8.	Decision Tree	Normal Osteoporosis Osteopenia	60%	59% 60% 61%	75% 55% 45%	66% 57% 52%
9.	Gradient Boosting	Normal Osteoporosis Osteopenia	68%	81% 62% 65%	59% 75% 69%	68% 68% 67%
10.	Gaussian Naive Bayes	Normal Osteoporosis Osteopenia	52%	46% 48% 70%	73% 27% 61%	56% 34% 66%

Table 2 : Performance comparison of classical ML Models (90 : 10) for three-class bone density classification

Sr. No.	ML Model	Class	Accuracy	Precision	Recall	F1 Score
1.	SVM - Linear	Normal Osteoporosis Osteopenia	42%	40% 38% 50%	45% 36% 44%	43% 37% 47%
2.	SVM - Sigmoid	Normal Osteoporosis Osteopenia	69%	81% 59% 79%	59% 79% 69%	68% 68% 73%
3.	SVM - Polynomial	Normal Osteoporosis Osteopenia	61%	69% 58% 60%	50% 75% 56%	58% 65% 58%
4.	SVM - RBF	Normal Osteoporosis Osteopenia	69%	92% 60% 73%	50% 88% 69%	65% 71% 71%
5.	KNN	Normal Osteoporosis Osteopenia	65%	60% 61% 79%	55% 71% 69%	57% 65% 73%
6.	Random Forest	Normal Osteoporosis Osteopenia	68%	86% 61% 65%	55% 79% 69%	67% 69% 67%
7.	Logistic Regression	Normal Osteoporosis Osteopenia	58%	62% 56% 56%	59% 58% 56%	60% 57% 56%
8.	Decision Tree	Normal Osteoporosis Osteopenia	60%	62% 59% 57%	59% 67% 50%	60% 63% 53%
9.	Gradient Boosting	Normal Osteoporosis Osteopenia	42%	43% 40% 43%	59% 45% 17%	50% 43% 24%
10.	Gaussian Naive Bayes	Normal Osteoporosis Osteopenia	53%	44% 50% 79%	64% 33% 69%	52% 40% 73%

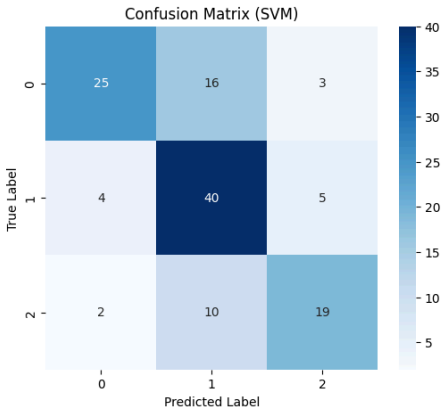
Table 3 : Performance comparison of classical ML Models (70 : 30) for three-class bone density classification

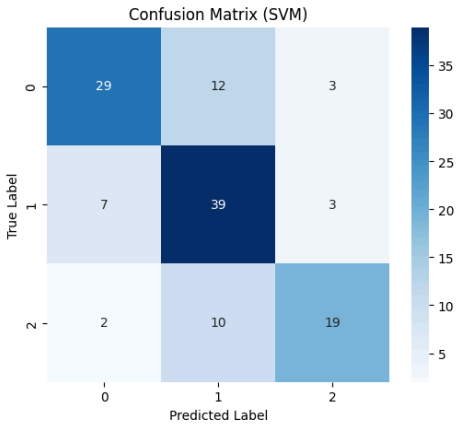
Sr. No.	ML Model	Class	Accuracy	Precision	Recall	F1 Score
1.	SVM - Linear	Normal Osteoporosis Osteopenia	40%	46% 42% 32%	48% 35% 36%	47% 38% 34%
2.	SVM - Sigmoid	Normal Osteoporosis Osteopenia	64%	70% 59% 64%	65% 66% 59%	68% 62% 61%
3.	SVM - Polynomial	Normal Osteoporosis Osteopenia	63%	75% 59% 55%	68% 70% 46%	71% 64% 50%
4.	SVM - RBF	Normal Osteoporosis Osteopenia	67%	85% 59% 65%	62% 81% 52%	72% 68% 58%
5.	KNN	Normal Osteoporosis Osteopenia	64%	71% 60% 60%	67% 66% 57%	69% 63% 58%
6.	Random Forest	Normal Osteoporosis Osteopenia	69%	81% 62% 63%	73% 75% 52%	77% 68% 57%
7.	Logistic Regression	Normal Osteoporosis Osteopenia	62%	65% 62% 55%	73% 62% 46%	69% 62% 50%
8.	Decision Tree	Normal Osteoporosis Osteopenia	57%	54% 59% 61%	68% 56% 41%	60% 57% 49%
9.	Gradient Boosting	Normal Osteoporosis Osteopenia	63%	71% 59% 61%	67% 70% 48%	69% 64% 54%
10.	Gaussian Naive Bayes	Normal Osteoporosis Osteopenia	54%	48% 50% 71%	67% 38% 59%	56% 43% 64%

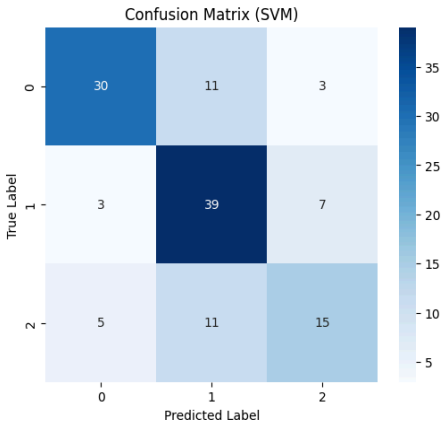
Table 4 : Performance comparison of classical ML Models (60 : 40) for three-class bone density classification

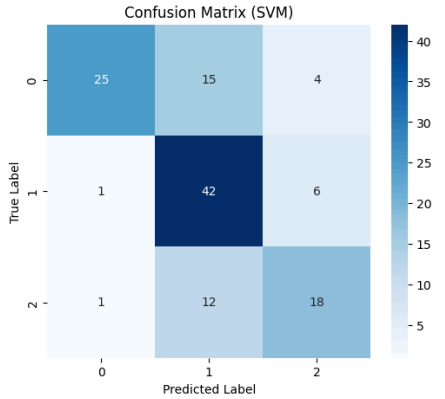
Sr. No.	ML Model	Class	Accuracy	Precision	Recall	F1 Score
1.	SVM - Linear	Normal Osteoporosis Osteopenia	38%	36% 37% 40%	42% 33% 39%	39% 35% 40%
2.	SVM - Sigmoid	Normal Osteoporosis Osteopenia	65%	75% 58% 67%	59% 73% 60%	66% 65% 63%
3.	SVM - Polynomial	Normal Osteoporosis Osteopenia	63%	76% 58% 57%	62% 73% 48%	69% 65% 52%
4.	SVM - RBF	Normal Osteoporosis Osteopenia	67%	88% 59% 67%	57% 84% 56%	69% 69% 61%
5.	KNN	Normal Osteoporosis Osteopenia	62%	63% 59% 64%	66% 61% 56%	64% 60% 60%
6.	Random Forest	Normal Osteoporosis Osteopenia	66%	82% 59% 62%	64% 75% 55%	72% 66% 58%
7.	Logistic Regression	Normal Osteoporosis Osteopenia	59%	65% 57% 55%	62% 61% 52%	64% 59% 53%
8.	Decision Tree	Normal Osteoporosis Osteopenia	54%	57% 52% 50%	70% 49% 37%	63% 51% 43%
9.	Gradient Boosting	Normal Osteoporosis Osteopenia	64%	76% 59% 60%	64% 70% 56%	69% 64% 58%
10.	Gaussian Naive Bayes	Normal Osteoporosis Osteopenia	56%	52% 54% 67%	62% 48% 60%	57% 51% 63%

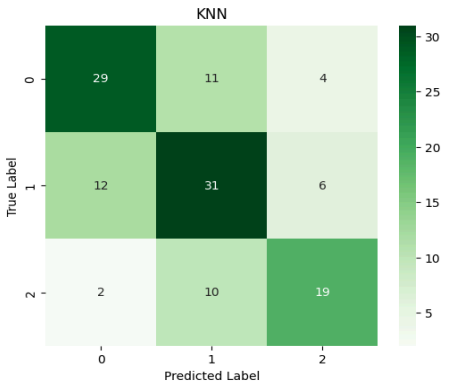
Confusion Matrix for Classical ML Models (80 : 20)

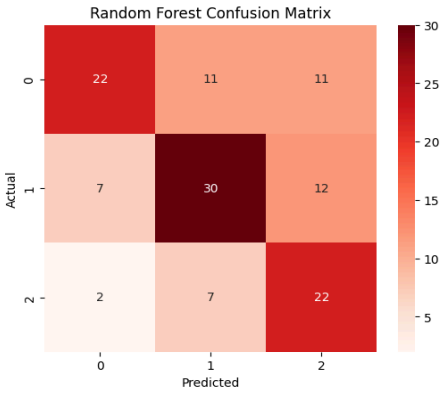


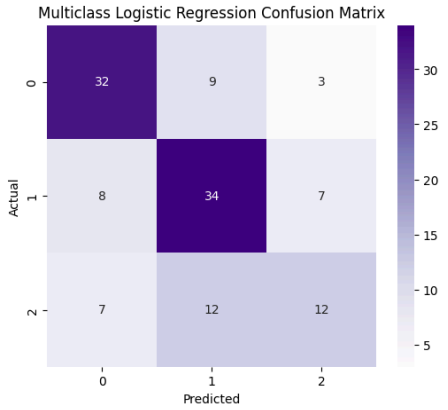


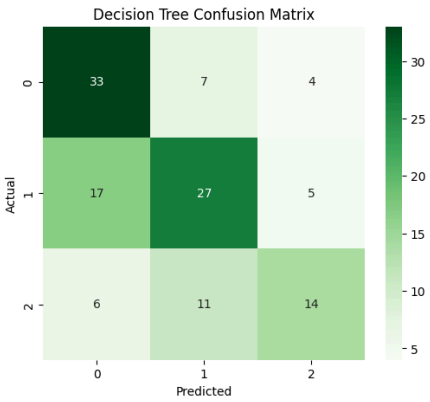


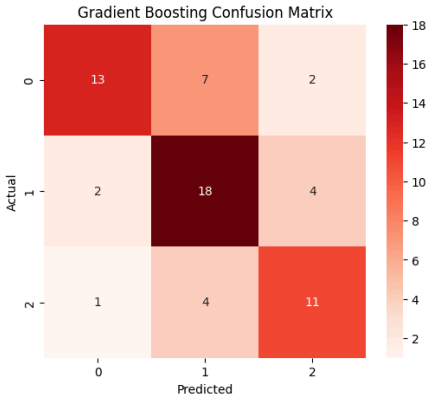


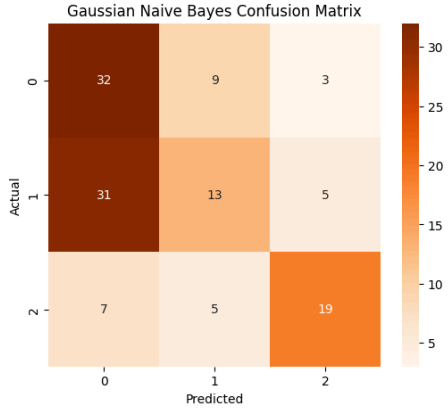




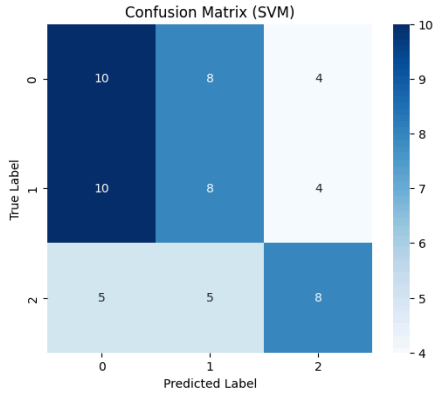


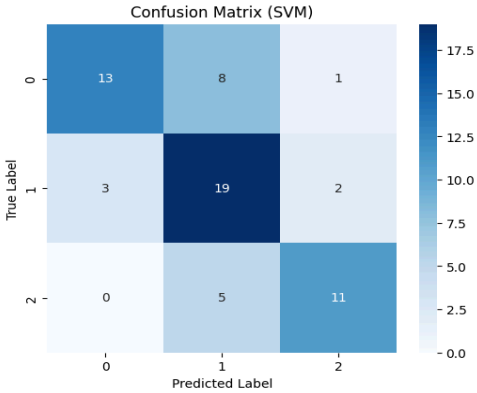


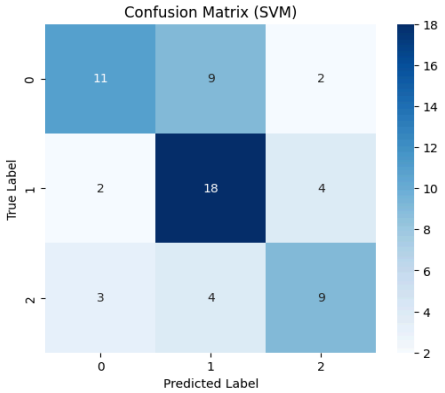


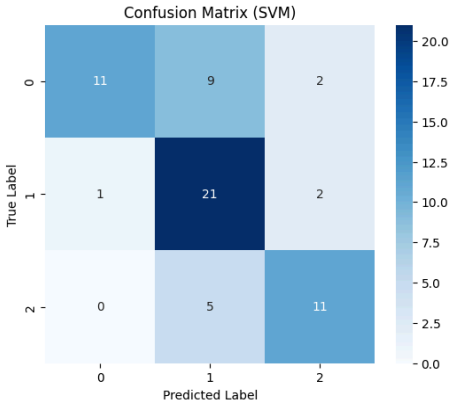


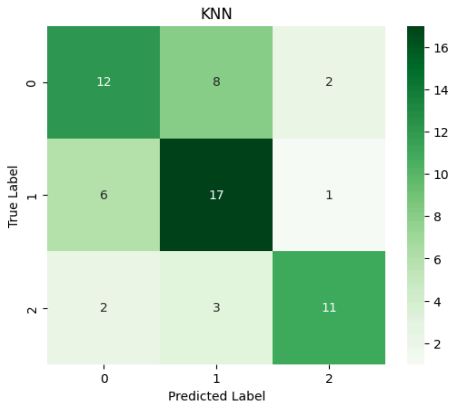
Confusion Matrix for Classical ML Models (90 : 10)

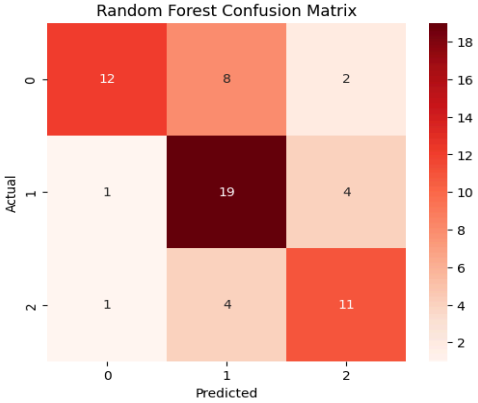


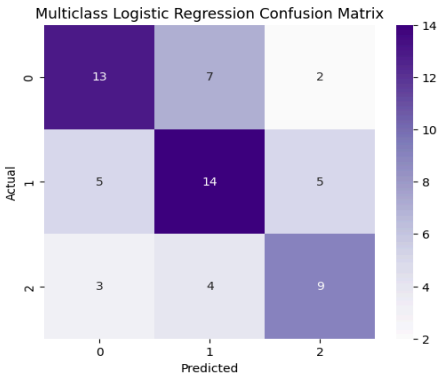


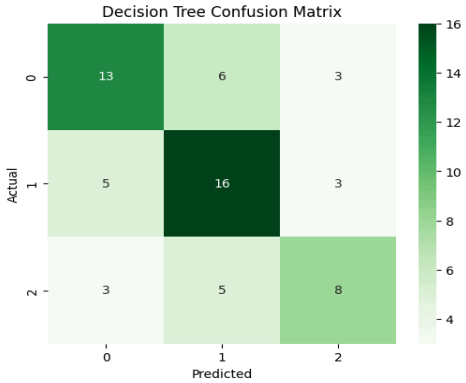


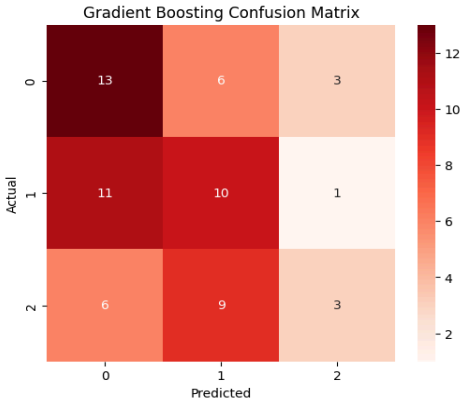


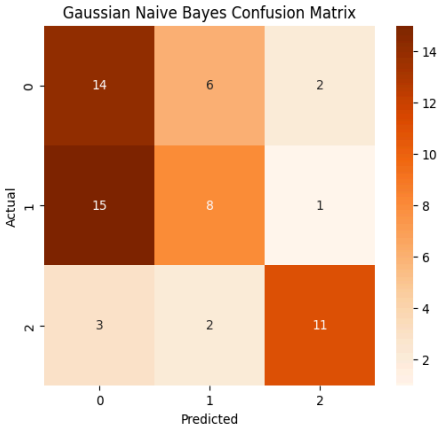




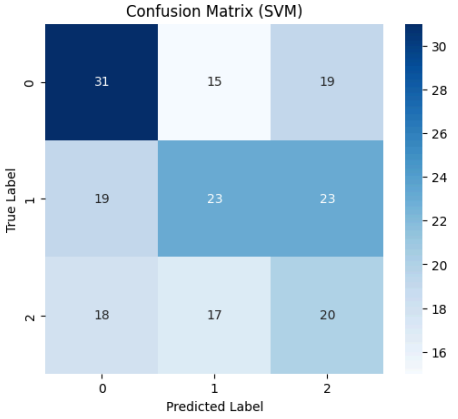


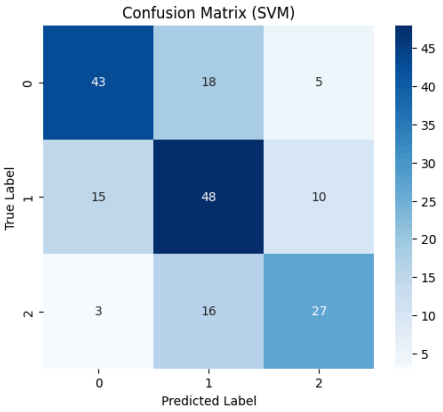


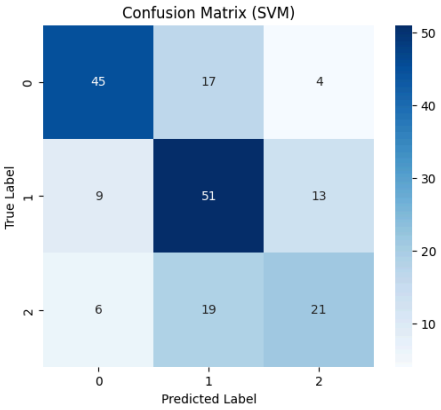


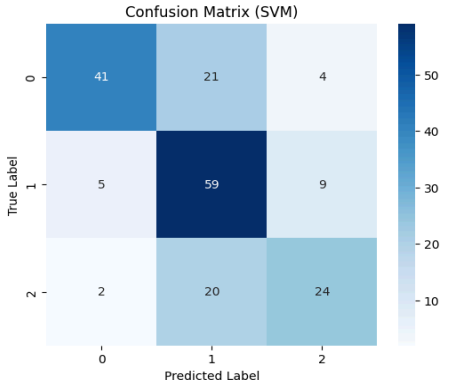


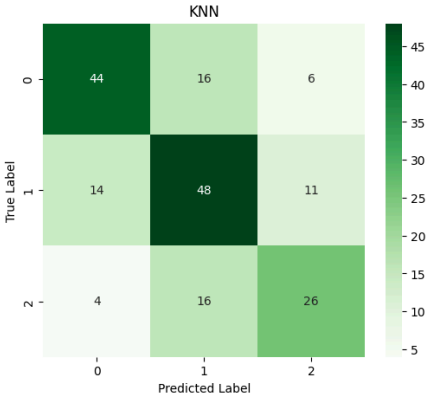
Confusion Matrix for Classical ML Models (70 : 30)

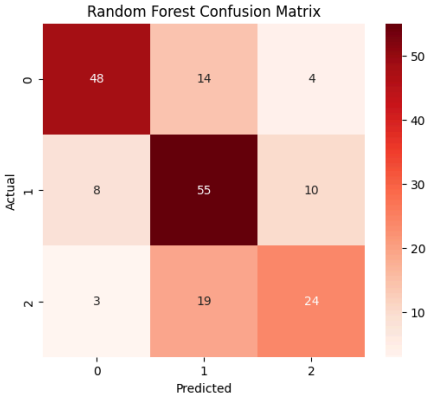


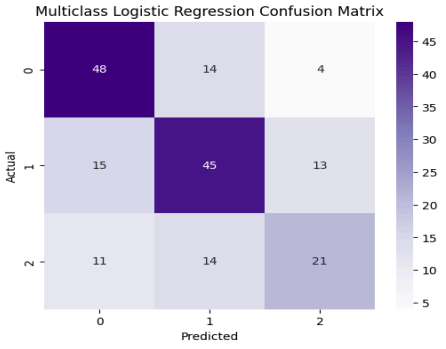


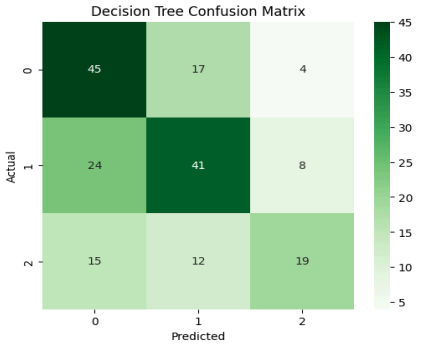


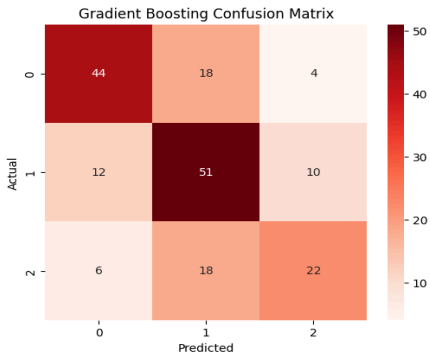


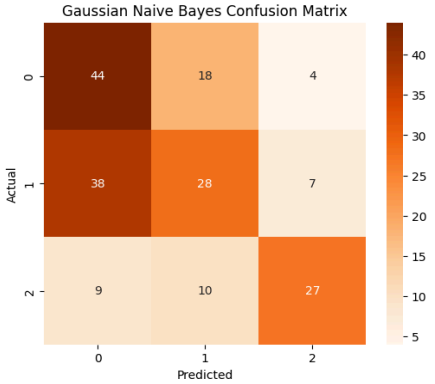




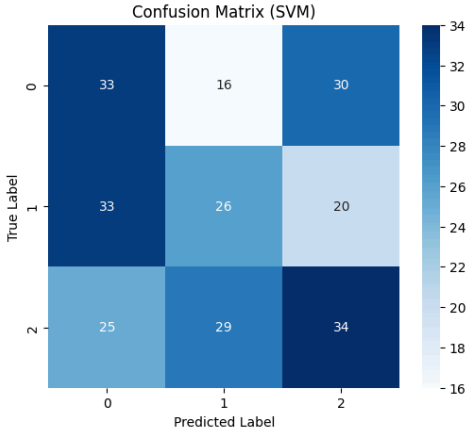


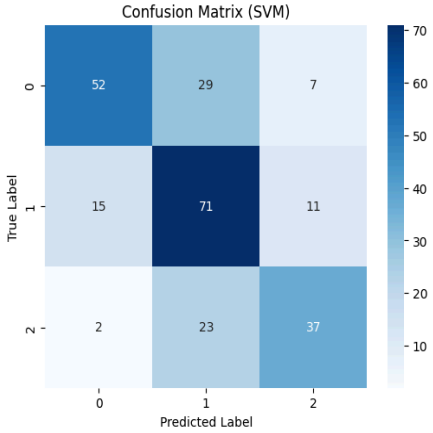


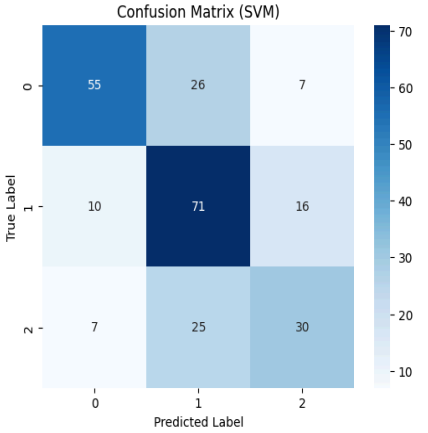


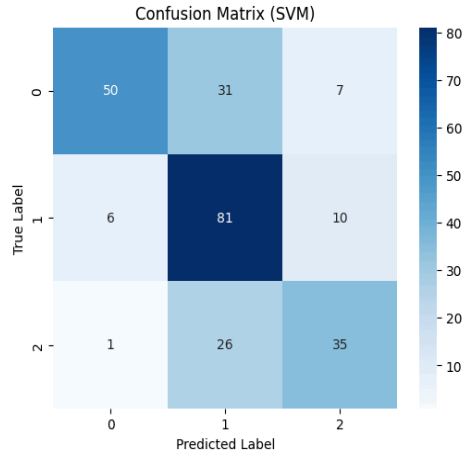


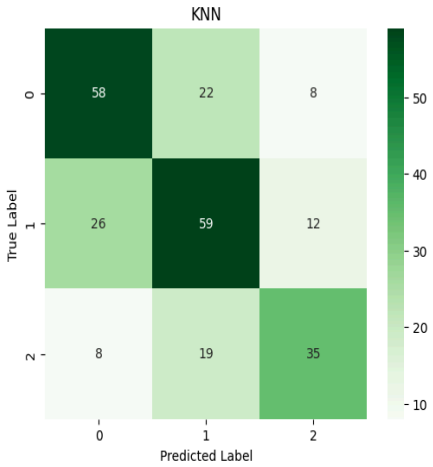
Confusion Matrix for Classical ML Models (60 : 40)

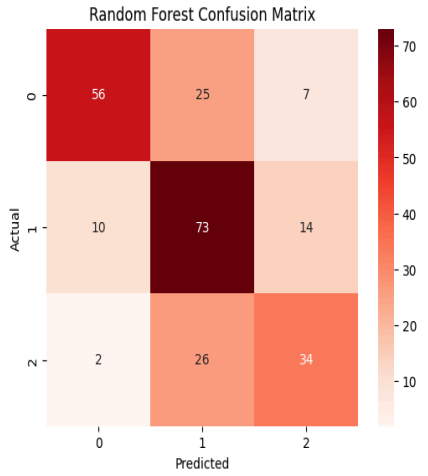


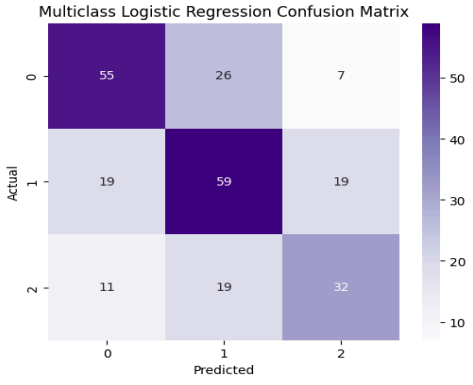


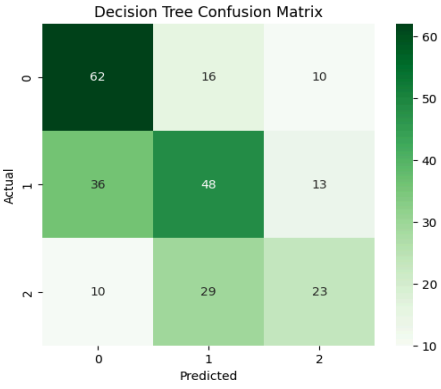


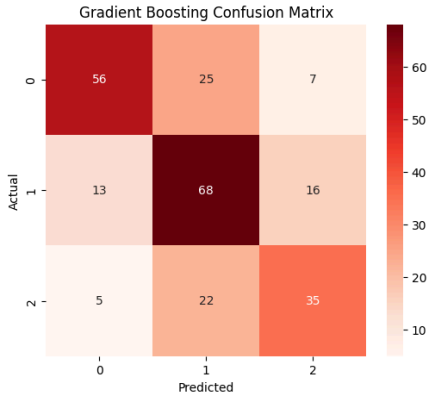


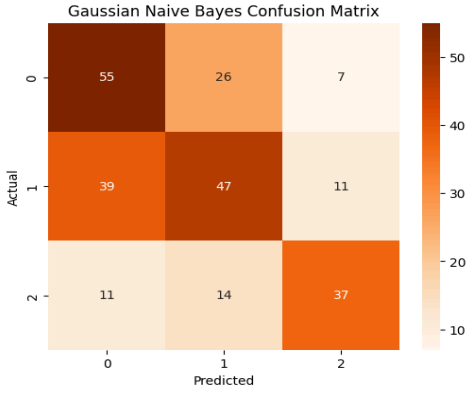




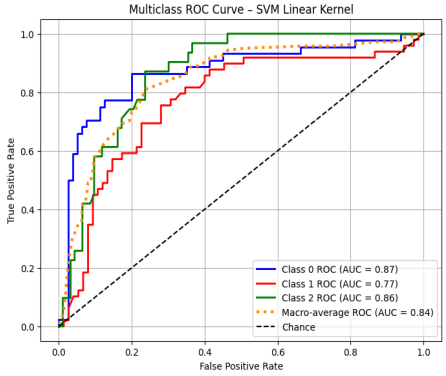


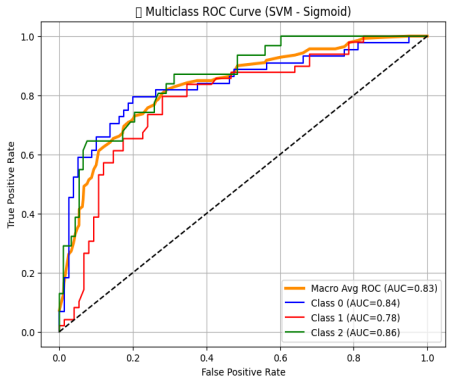


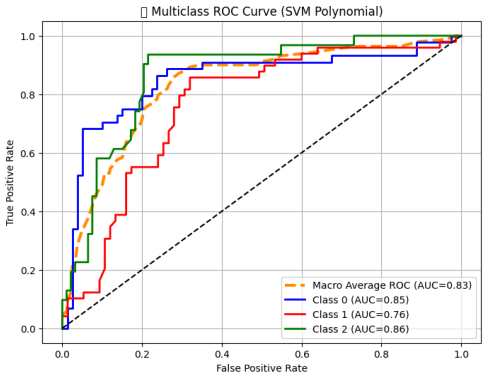


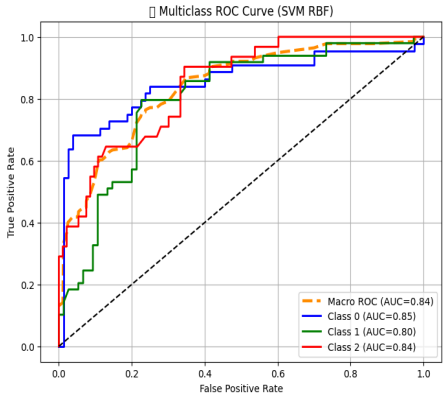


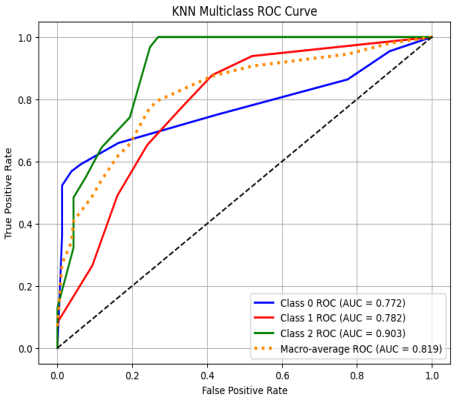
ROC Curve for classical ML Models (80 : 20)

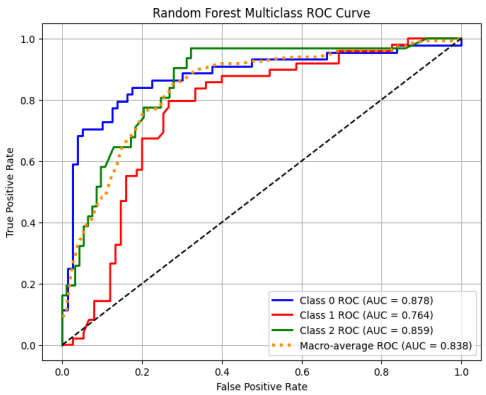


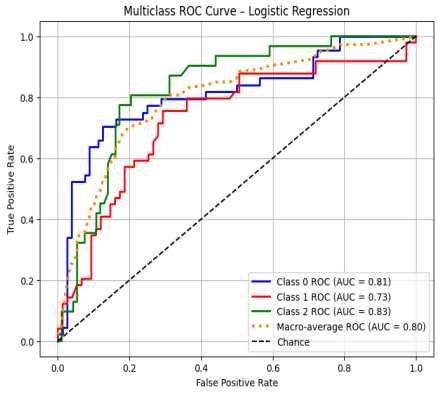


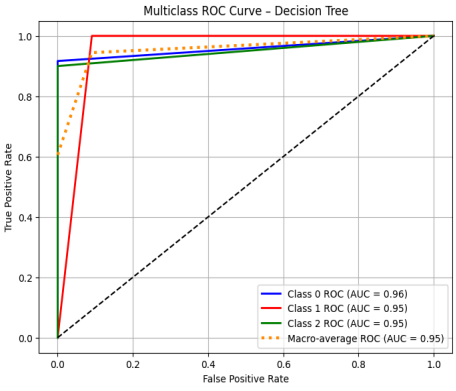


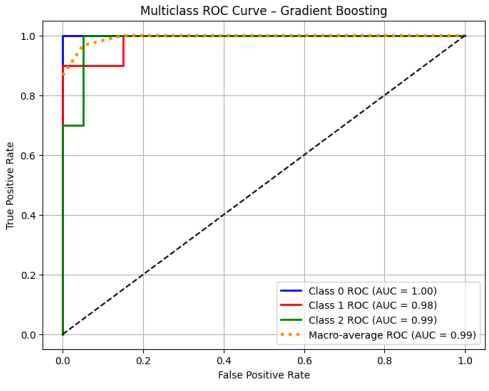


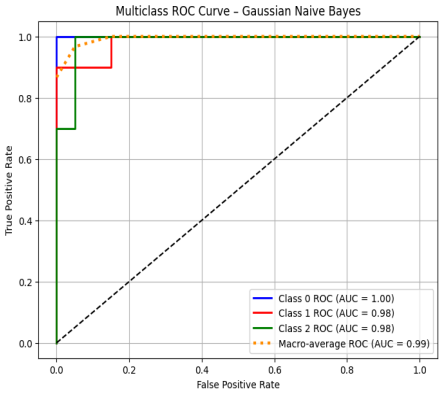




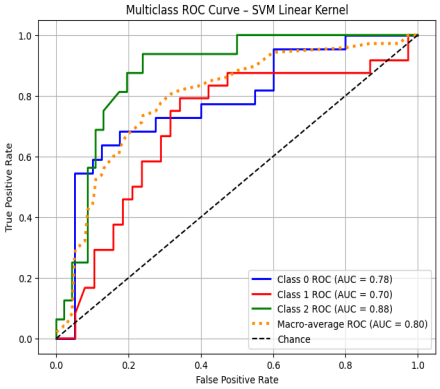


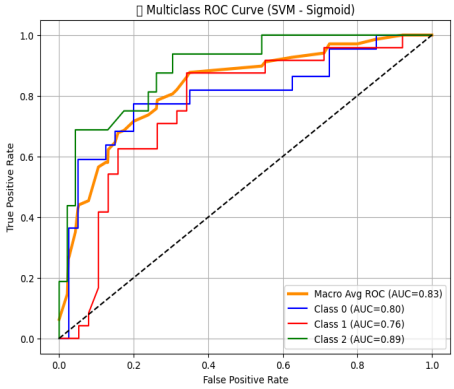


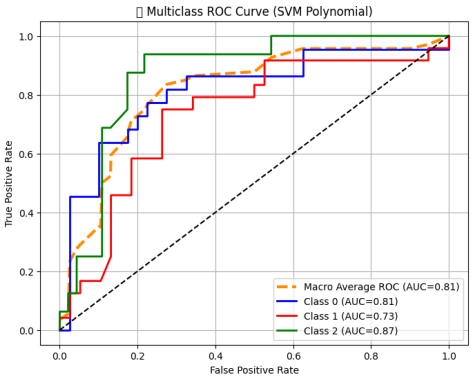


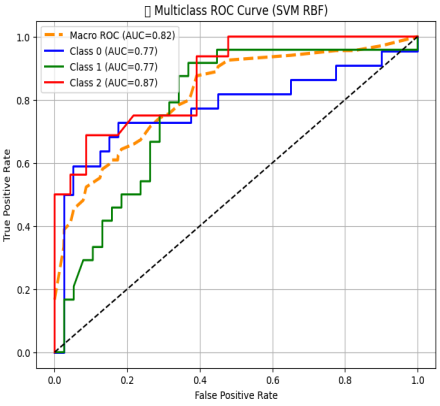


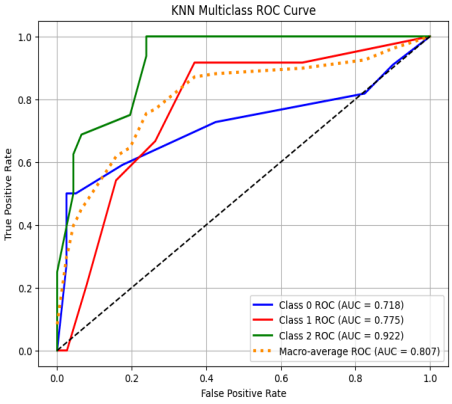
ROC Curve for classical ML Models (90 : 10)

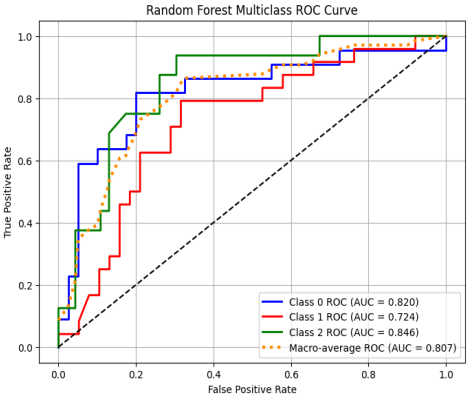


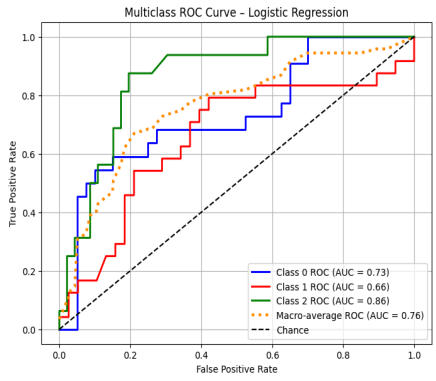


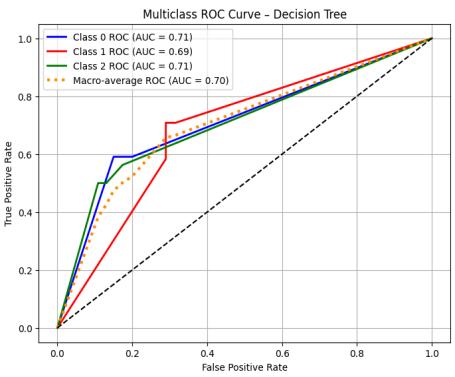


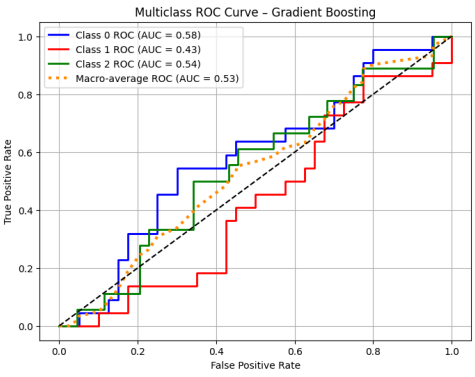


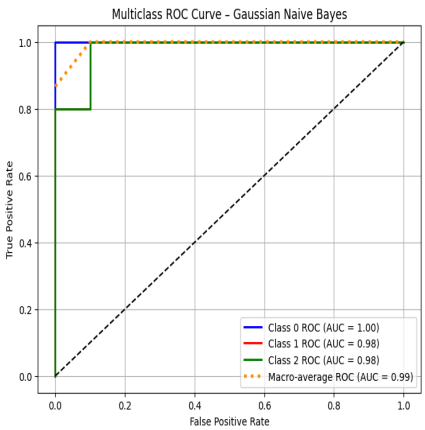




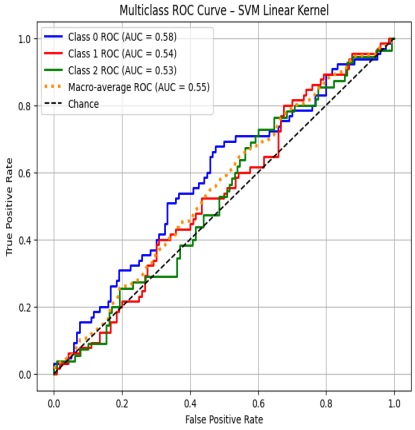


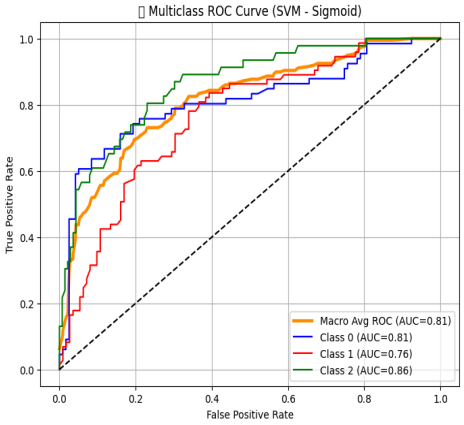


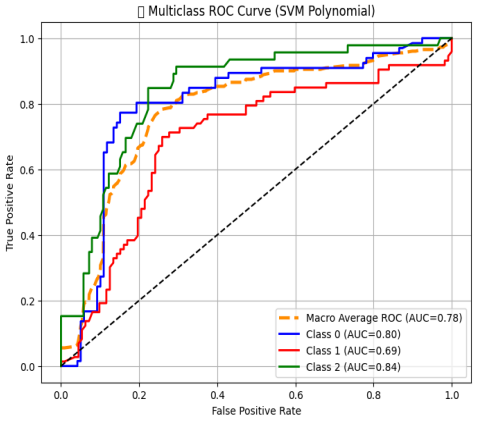


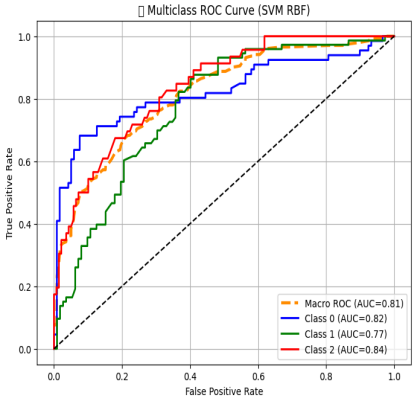


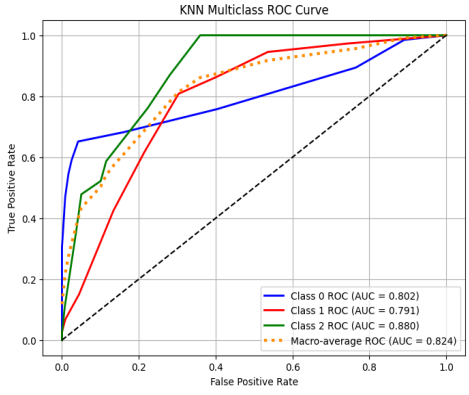
ROC Curve for classical ML Models (70 : 30)

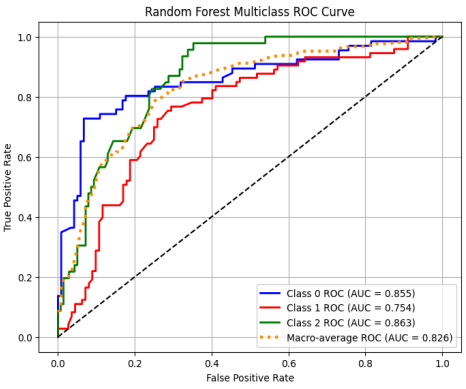


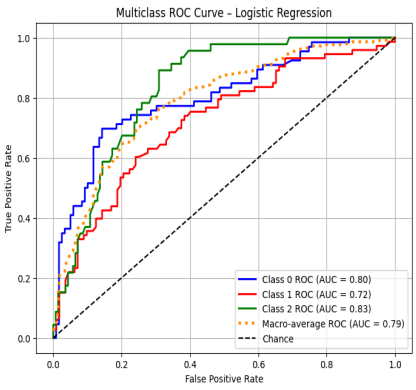


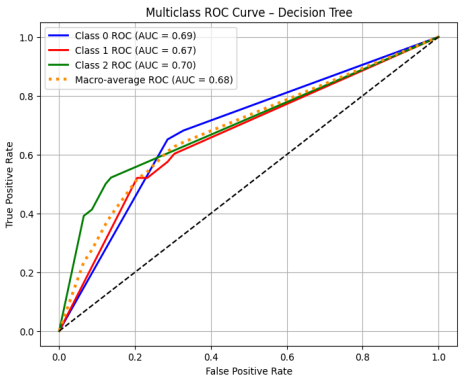


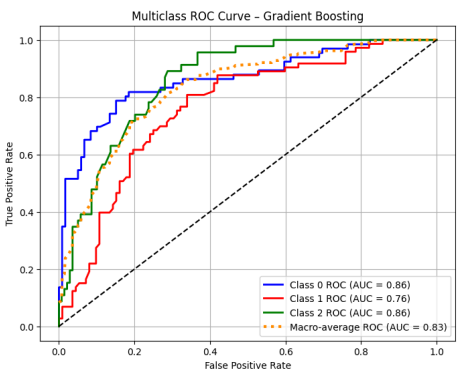


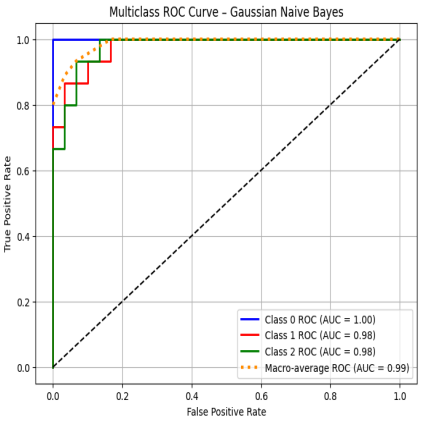




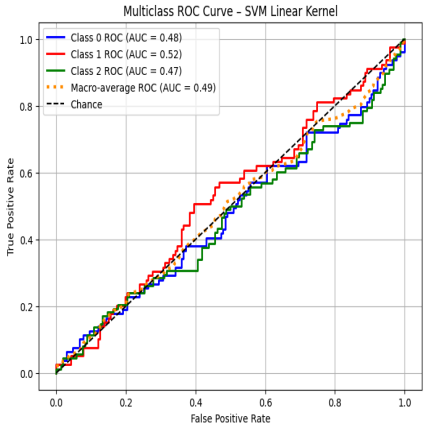


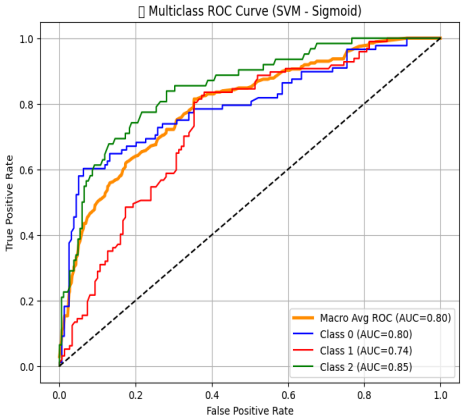


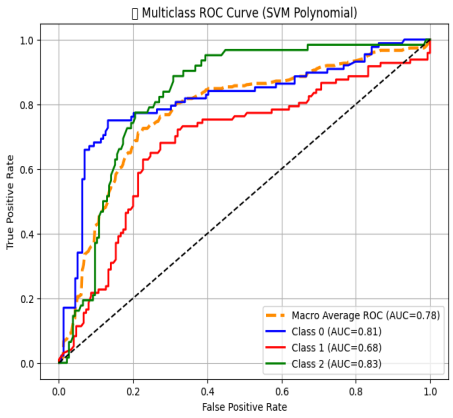


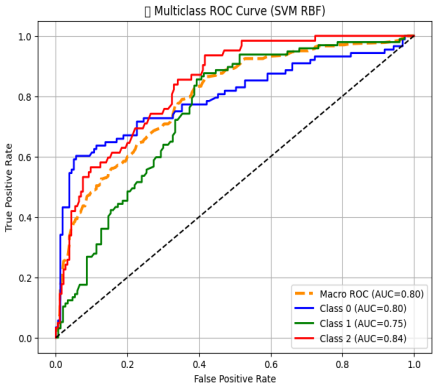


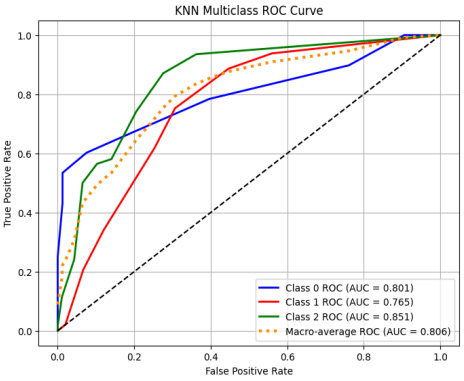
ROC Curve for classical ML Models (60 : 40)

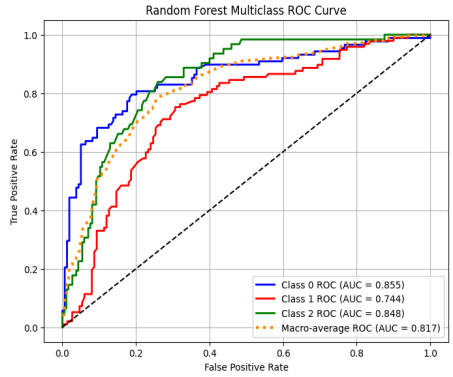


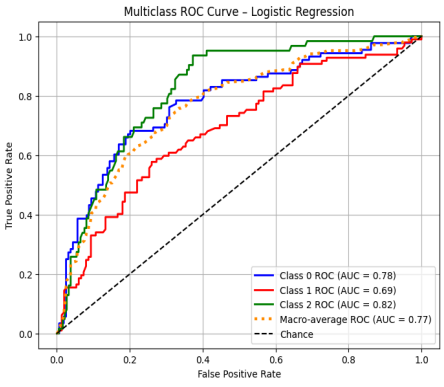


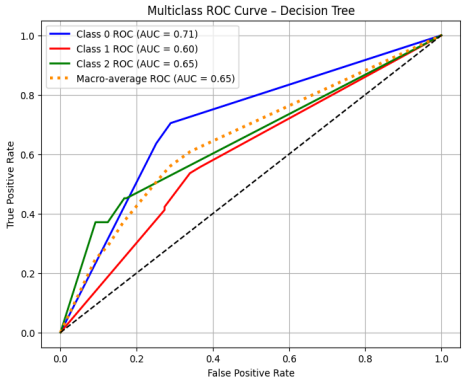


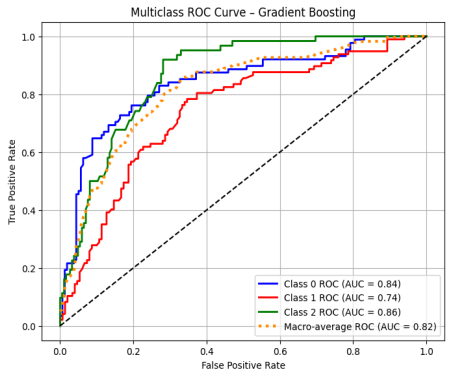


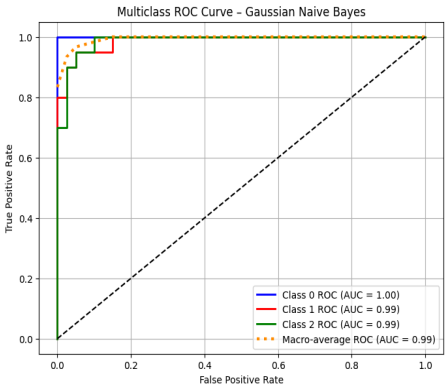












Conclusion

This work demonstrates that fractal geometry, particularly Sierpiński-triangle–based masking, can effectively capture the subtle texture variations present in trabecular bone on standard X-ray images. By combining careful preprocessing, geometric decomposition, and statistical feature extraction, the system successfully transforms raw images into a rich numerical representation of bone quality. The performance of multiple machine-learning models shows that these fractal-derived features carry strong discriminatory information for distinguishing Normal, Osteopenia, and Osteoporosis cases.

Across different train–test splits, models such as SVM, Random Forest, and Gradient Boosting consistently achieved competitive accuracy, confirming the reliability of the feature-engineering approach. Importantly, this method works entirely on routine X-ray scans, which makes it a cost-effective and accessible alternative to advanced tools like DEXA.

Overall, the study establishes a complete and lightweight pipeline capable of supporting automated bone-health assessment. With further refinement—such as deeper fractal levels, more diverse datasets, or integration with deep learning—the system has the potential to evolve into a practical clinical decision-support tool.