Student ID: 11513161, 11583573, 11561015, 14141925, 14126610

Coursework Report

1. Unsupervised Clustering Method

1.1. Concept of model-based clustering

Model-based clustering is statistical method that assumes the data came from probabilistic model (usually a mixture of distribution). Unlike heuristic method like k-means, model-based clustering explicitly estimates the probability distributions that define the clusters.

1) Mixture model: The data is assumed to come from a mix of several probability distributions, often Gaussian mixtures.
2) Soft class assignment: Rather than being placed in a single cluster, a data point has a probability of belonging to each cluster.
3) Expectation-Maximization (EM) Algorithm: EM is commonly used to estimate the parameters of the mixture model by iteratively updating cluster members and model parameters.

1.2. Gaussian Mixture Model (GMM)

A general, k-component Gaussian mixture has the form:

$$f(x) = \sum_{j=1}^{k} \pi_j f\left(x \middle| \mu_j, \Sigma_j\right)$$

k is the number of clusters

π_j is the weight of the cluster

$f\left(x \middle| \mu_j, \Sigma_j\right)$ is a Gaussian distribution with mean=μ_j, covariance=Σ_j

1.3. Expectation-Maximization Algorithm

In this section we are going to show the EM algorithm step by step.

Given there are data points $X = \{x_1, x_2, \ldots\ldots, x_n\}$, assume a mixture of K Gaussians:

The steps are:

1) Initialize parameters
   - Choose K, number of clusters
   - Initialize π_j, μ_j, Σ_j randomly or use k-means.
2) E-Step (Expectation)

   Compute the cluster assignment probabilities z_{ij} (the probability of data i in cluster j) of each cluster for each data point using Bayes' rule:

$$z_{ij} = \frac{\pi_j f(x_i \mid \mu_j, \Sigma_j)}{\sum_{r=1}^{k} \pi_r f(x_i \mid \mu_r, \Sigma_r)}$$

3) M-Step (Maximization)

   Update the mixture parameters based on the computed matrix z.

   Update mixing coefficients:

$$\pi_j = \frac{1}{n}\sum_{i=1}^{n} z_{ij}$$

Update means:

$$\mu_j = \frac{\sum_{i=1}^{n} z_{ij} x_i}{\sum_{i=1}^{n} z_{ij}}$$

Update covariances:

$$\Sigma_j = \frac{\sum_{i=1}^{n} z_{ij} \left(x_i - \mu_j\right)^T \left(x_i - \mu_j\right)}{\sum_{i=1}^{n} z_{ij}}$$

This step is called the *maximization* because we update the parameter estimates to maximize the likelihood of the observed data, given the expected values computed in the E-Step.

4) Check Convergence

Repeat step 2 and 3 until $\boldsymbol{z_{ij}}$ stop changing significantly.

2. Supervised Clustering Method

2.1 Introduction to Supervised Classification

Supervised classification is a type of machine learning where a model is trained on labelled data to learn the relationship between input features and their corresponding class labels. Once trained, the model predicts the class of new, unseen data points.

Support vector machine (SVM) is an effective supervised classification method that maximizes the margins between different classes by finding an optimal separation hyperplane (Rogers & Girolami, 2016; Bishop, 2006).

2.2 The SVM Optimization Problem

2.2.1 Linearly Separable Case

When categories are fully distinguishable, SVM maximizes the margin between classes while ensuring all data points are correctly classified. The decision boundary is defined as:

$$f(x) = w^T x + b$$

where is the weight vector and is the bias term. The optimization problem to maximize the margin is formulated as:

$$\min_{w,b} \frac{1}{2} \|w\|^2$$

subject to:

$$y_i(w^T x_i + b) \geq 1, \quad \forall i$$

where y_i represents the class label of each data point x_i.

2.2.2 Non-Separable Case: Soft-Margin SVM

In real-world scenarios, data is often non-separable. SVM uses slack variables (ξi) to allow some misclassifications while preserving a large margin, leading to the following optimization problem:

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \xi_i$$

subject to:

$$y_i(w^T x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0, \quad \forall i$$

The parameter C controls the trade-off between maximizing the margin and minimizing misclassification errors (Week 4 Lecture Slides, DATA70132).

2.3 Handling Non-Linearly Separable Data

When a linear decision boundary is insufficient, SVM uses the kernel trick to map data into a higher-dimensional space for linear separation. Common kernel functions include:

- Linear Kernel: $K(x_i, x_j) = x_i^T x_j$
- Polynomial Kernel: $K(x_i, x_j) = (x_i^T x_j + c)^d$
- Radial Basis Function (RBF) Kernel: $K(x_i, x_j) = \exp\left(-\gamma|(|x_i - x_j|)|^2\right)$

The RBF kernel maps data into a Gaussian-defined space, improving classification in complex datasets (Week 4 Lecture Slides, DATA70132).

2.4 SVM Decision Function

Once trained, an SVM classifies new instances based on the following decision rule:

$$\hat{y} = sign(w^T x + b)$$

This ensures that each new data point is assigned to one of the predefined classes based on its position relative to the decision boundary.

2.5 Advantages of SVM

SVMs excel in high-dimensional spaces, ideal for text classification and bioinformatics. With proper regularization and kernel selection, they resist overfitting and ensure good generalization. Their use of convex optimization guarantees a stable, unique global solution.

3. Exploratory Data Analysis and Preprocessing

3.1. Univariate EDA and Preprocessing

3.1.1. Data Description

The dataset contains 310 rows and 6 features: Pelvic Tilt, Lumbar Lordosis Angle, Sacral Slope, Pelvic Radius, Grade of Spondylolisthesis, and Class (target variable), with no missing values.

3.1.2. Class Distribution

The majority class (AB) in the target variable, Class, comprises 67.74% of the dataset, whereas the minority class (NO) accounts for only 32.26%. This class imbalance should be considered during the model training and evaluation.

3.1.3. Log Transformation

Log transformation was applied to Grade of Spondylolisthesis to handle skewness in the data, this helps normalize the data and reduce the impact of extreme values.

3.1.4. Outlier Detection and Removal

These outliers in Figure 3-1 were identified and removed to ensure the data's robustness and improve model performance using IQR method. Although medical datasets are sensitive to outlier removal, orthopaedical research supports it for improved performance (Rezapour et al., 2024; Rouzrokh et al., 2024; Li et al., 2015).

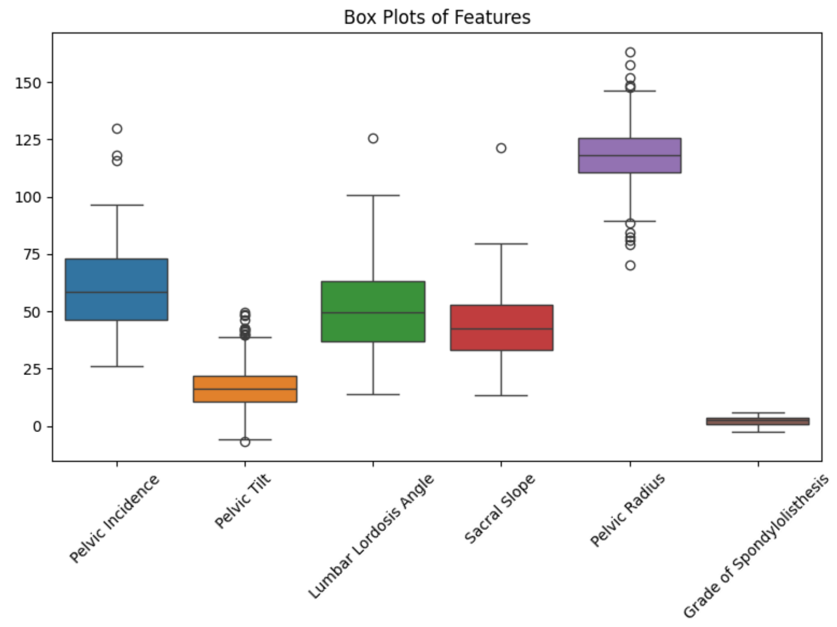

Figure 3-1: Boxplot of features

3.1.5. Standardization

Since the features have different scales, standardization is necessary to bring all features to a common scale.

3.1.6. Target feature encoding

Class is encoded into numerical values (AB = 1, NO = 0) for supervised learning.

3.2. Multivariate EDA

3.2.1. PCA

Figure 3-2 shows the data projected into a lower-dimensional space using PCA.

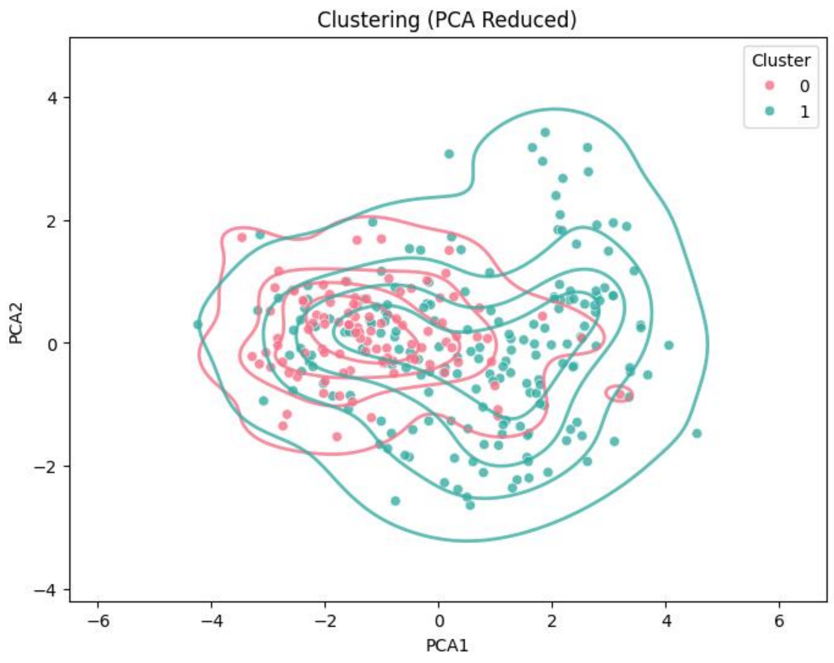

Figure 3-2: PCA Clustering

3.2.2. Correlation Matrix

Figure 3-3 reveals correlations between features:

- Pelvic Incidence is highly correlated with Lumbar Lordosis Angle (0.79) and Sacral Slope (0.85).
- Class has weak correlations with most features, indicating that the relationship between features and the target is non-linear.

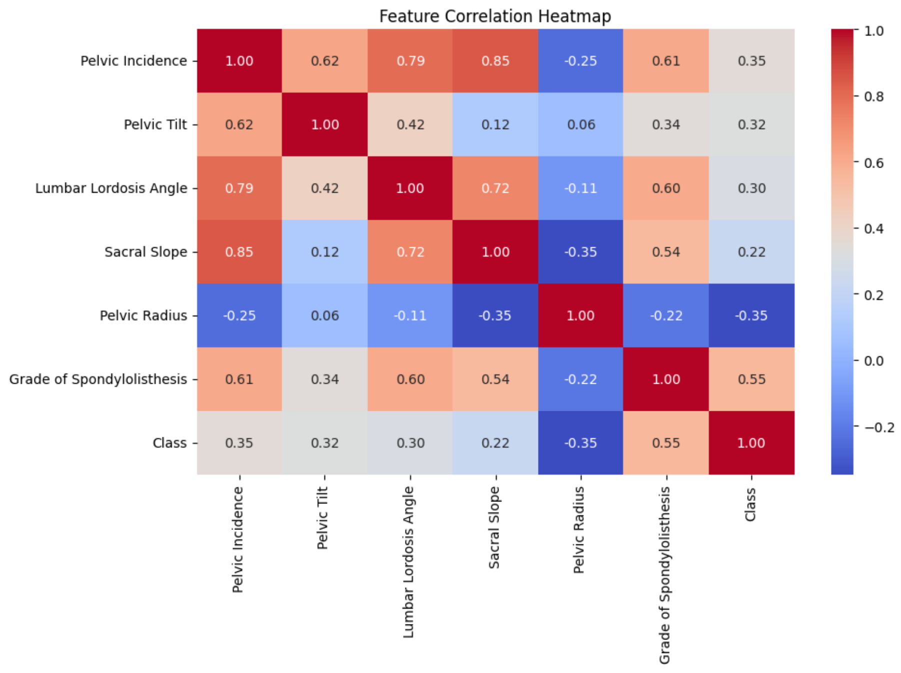

Figure 3-3: Correlation Heatmap

3.2.3. Pairplot

Figure 3-4 visualizes pairwise relationships between features, colored by the target variable. Pelvic Incidence show a strong positive correlation with Lumbar Lordosis Angle and Sacral Slope.

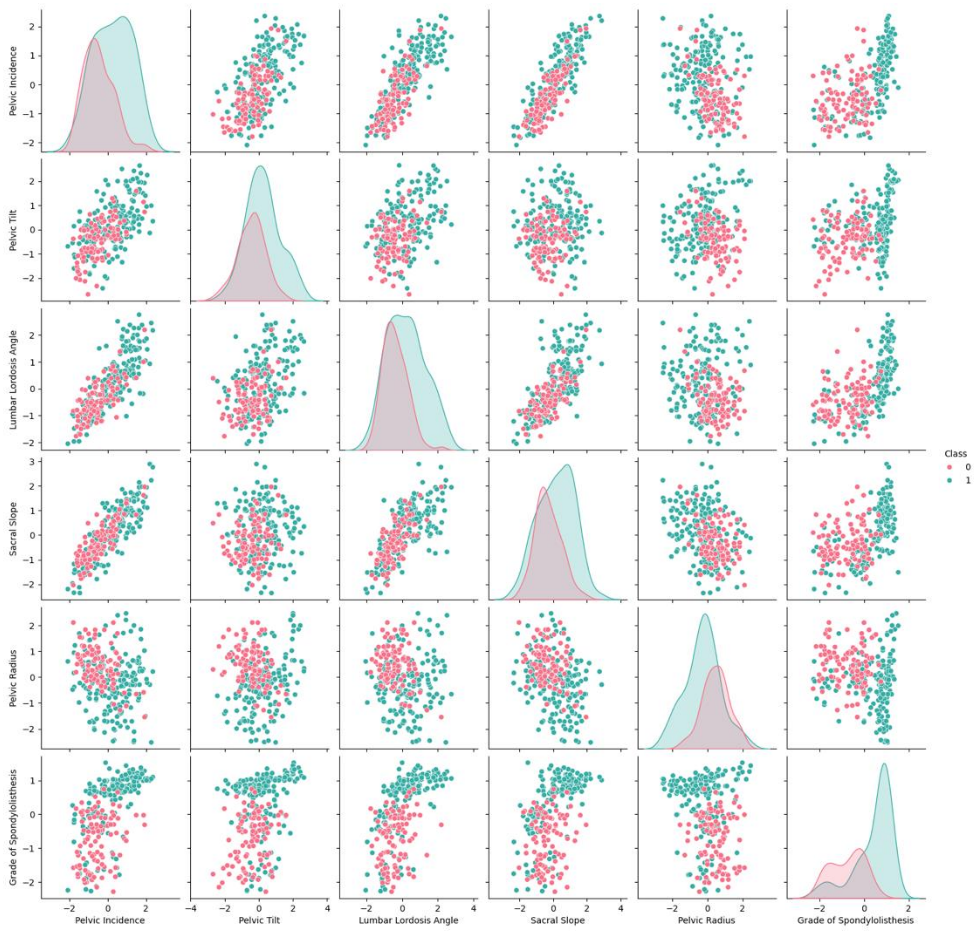

Figure 3-4: Pairplot

3.3. Preprocessing for Modeling

3.3.1. Train-Test Split

The dataset is split into training and testing sets (80-20 split) to evaluate model performance. Stratified sampling is used to maintain the class distribution in both sets.

4. Result of Analysis

Recall is used in this analysis instead of the accuracy score, due to the imbalanced class in the target feature. Moreover, recall is an appropriate metric when the priority is to identify as many positive cases as possible. In this case, it helps minimizes the risk of missing abnormal cases (false negatives), which is more serious than incorrectly identifying a normal case as abnormal (false positive).

The weighted F1-score, on the other hand, is prioritized as the primary evaluation metric to ensure the balance assessment of both majority (Abnormal) and minority (Normal) class predictions, and it has been widely used for classification problems (Sokolova et al., 2006; Wiwie et al., 2015). This paper utilizes both the recall and F1-score, to ensure that it does not disregard abnormal detection failure that is a risk in medical circumstances, also evaluates the model in a balanced way taking both recall and precision into account.

4.1. Unsupervised Clustering Results

Two clustering methods, Kernel K-means Clustering and Model-Based Clustering (Gaussian Mixture Model; GMM), are applied in this analysis with results summarized in Table 4-1. Kernel K-means slightly improves when eliminating "Pelvic Incidence" (F1-score: 0.716 → 0.741, Recall(weighted): 0.711 → 0.735), which exhibits high covariances (Figure 3-4). The Gram Matrix in Kernel K-means becomes unstable with high multicollinearity, leading to near-linear dependence and numerical instability in its inversion (Straub, 2018) Despite this, Kernel K-means performs stably overall.

GMM exhibits greater stability across all metrics (F1 Score: 0.748, Recall (Weighted): 0.742, Recall (Class1): 0.681), indicating robustness against multicollinearity. Silhouette Score (0.319) solely informs the degree of cluster overlap, and its score is deemed acceptable for the research objective. Figure 4-1 illustrates forming two clusters with some overlap in the principal component space. Given GMM's advantage, subsequent discussion focuses on its EM Algorithm implementation. Using the GaussianMixture module from scikit-learn, the algorithm initializes components randomly, computes probabilities, and iteratively maximizes likelihood (Scikit-learn Developers, 2024).

To ensure convergence to a local optimum, grid search is conducted, and Table 4-2 presents the best parameters achieving the highest F1-score. Three optimizations significantly enhance model performance: (1) The tied covariance matrix reduces free parameters, preventing overfitting in small sample sizes (Bouveyron & Brunet-Saumard, 2012). (2) K-means initialization mitigates local optima risks by structuring the initial clustering process (Celeux & Govaert, 1992). (3) The regularization term (*reg_covar*) stabilizes numerical computation in high-dimensional data. These optimizations fine-tune the model, ensuring both convergence and high performance for this dataset.

Table 4-1: Performance scores of the unsupervised learning methods.

No	Method	Silhouette Score	F1 Score (Weighted Average)	Recall (Weighted)	Recall (Class1)
1	Kernel K-Means	0.342	0.716	0.711	0.600
2	Kernel K-Means (w/o Pelvic Incidence)	0.318	0.741	0.735	0.637
3	**GMM (EM Algorithm)**	**0.319**	**0.748**	**0.742**	**0.681**
4	GMM (EM Algorithm) (w/o Pelvic Incidence)	0.311	0.748	0.742	0.681

Table 4-2: The optimal parameters of GaussianMixture methods.

No	Parameter	Optimal Value
1	n_components	2
2	covariance_type	“tied"
3	max_iter	500
4	warm_start	TRUE
5	tol	0.0001
6	init_params	“kmeans"
7	reg_covar	0.001
8	n_init	10

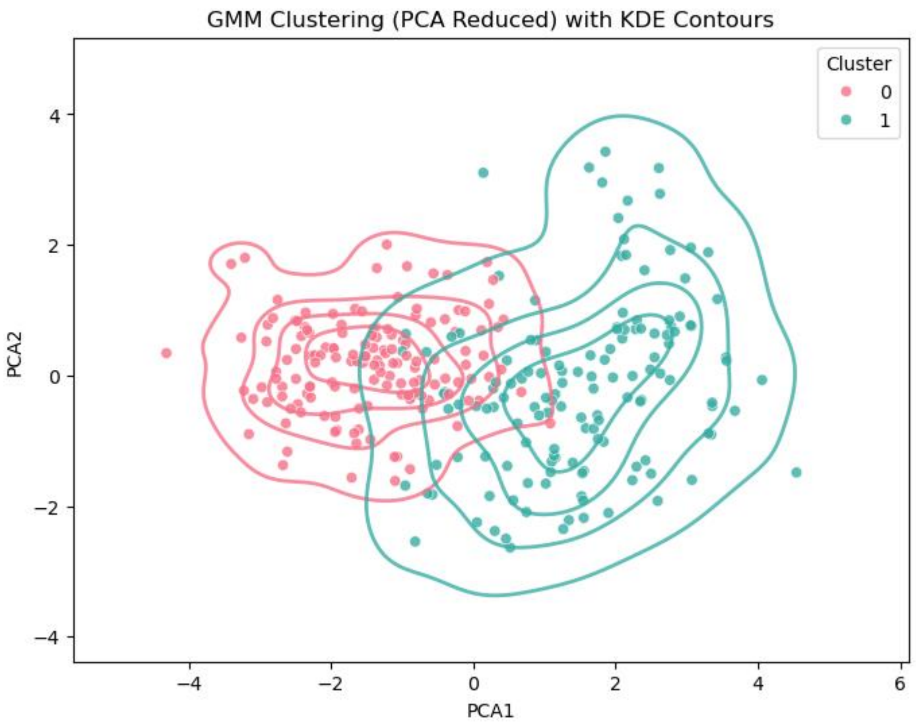

Figure 4-1: The PCA of the GMM model's classification result.

4.2. Supervised Clustering Results

Two types of clustering methods, namely k-Nearest Neighbours (kNN) and Support Vector Machines (SVM), are applied in this analysis and the results are shown in **Table 4-3**. kNN was optimized using Grid Search to identify the best number of neighbors with 5-fold cross-validation, while SVM's base model was evaluated using 5-fold cross-validation, selecting the models with the highest recall.

SVM was selected due to its strong generalization ability, particularly in handling complex decision boundaries through the Radial Basis Function (RBF) kernel (Cortes & Vapnik, 1995). The best SVM model achieved a cross-validation train recall of 0.8064, which means it effectively captures patterns in the training set. This recall score is comparable to its test recall (weighted average) of 0.870, indicating its robustness and strong generalization to unseen data.

For comparison, kNN was trained using 11 neighbors, as it gave the best performance during hyperparameter tuning. While kNN exhibited a higher recall on the training set (0.8281) compared to SVM, this may suggest a tendency to overfit. Unlike SVM, kNN relies on distance metrics making it highly sensitive to small changes in the dataset (Halder et al, 2024) and less suitable for categorical or binary data.

SVM remains a better model because it is more efficient in identifying abnormal case, achieving the test recall of 0.9189 for Class 1 (Abnormal), compared to 0.8649 for kNN, and outperformed in F-1 score, achieving 0.7993 compared to 0.7867. This indicates that SVM maintains a better balance between recall and precision, reducing both false positives and false negatives.

Additionally, based on these findings and analysis, SVM is a preferred model. To further analyze the classification results, PCA was implemented to visualize the well-classified result from the SVM model, as shown in the Figure 4-2.

Table 4-3: Performance scores of the supervised learning methods.

No	**Method**	**Training set (Cross-Validation)**	**Test set**			
		Recall (Average)	**Recall**		**F1-score Weighted Average**	**Confusion Matrix**
			Class 1 (AB)	**Weighted Average**		
1	**SVM**	**0.8064**	**0.9189**	**0.8070**	**0.7993**	**[[12 8] [3 34]]**
2	kNN	0.8281	0.8649	0.7895	0.7867	[[13 7] [5 32]]

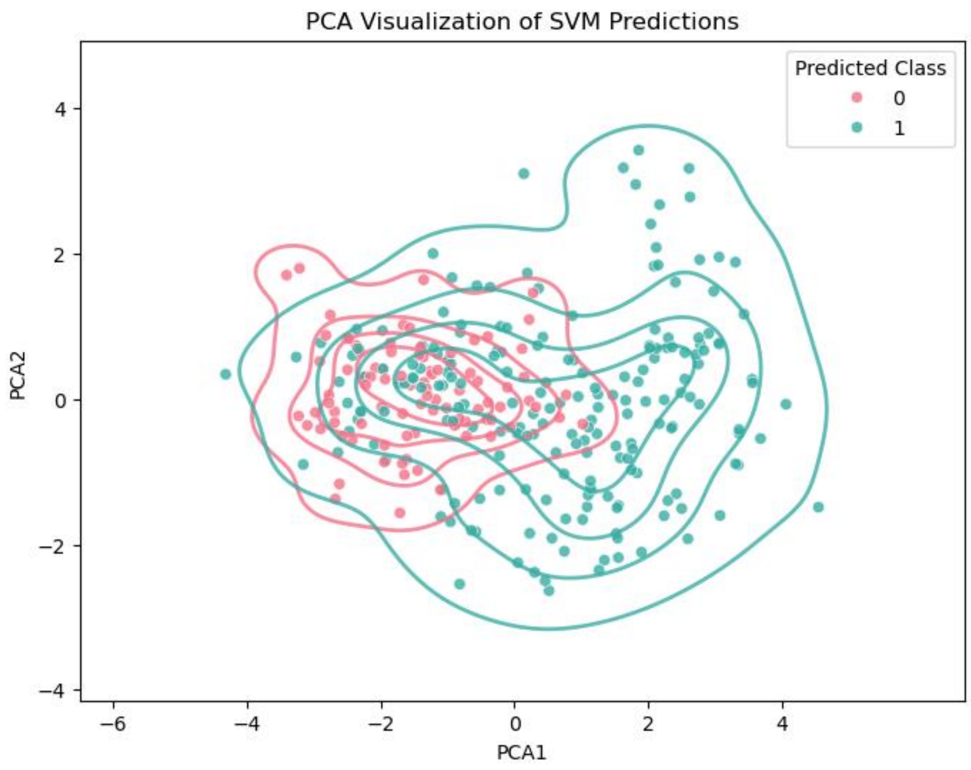

Figure 4-2: The PCA of the SVM model's classification result (0=NO, 1=AB).

4.3. Discussion

When using GMM, data was grouped into clusters based on patterns without prior knowledge of their actual classes. However, a key limitation of unsupervised learning is that while it effectively separates data into clusters, illustrated in Figure 4-1, it does not inherently define their meaning. In other words, we can identify patterns, but we do not know which cluster corresponds to a "normal" or "abnormal" class in the training process. To address this, we incorporated supervised learning using SVM classification, which utilizes labeled data to provide meaningful class predictions. By overlaying the supervised classification results onto the unsupervised clustering visualization, we assigned the discovered clusters with actual class labels, as shown in Figure 4-2. This allowed for better interpretation of the clusters, ensuring that the identified patterns were meaningful in the context of classification. Additionally, visualization helped assess whether the clusters captured the true class distribution and whether refinements such as feature selection or hyperparameter tuning were necessary.

This interplay between supervised and unsupervised learning is particularly valuable in this dataset due to the limited labeled samples. Additionally, unsupervised clustering can reveal hidden subgroups, such as patient clusters with shared clinical characteristics, while mismatches indicate subgroups requiring further investigation.

5. Codes and Explanation

5.1. Gaussian Mixture Model (GMM)

The function run_em_gmm was defined to train Expectation-Maximization (EM) algorithm using a Gussian Mixture Model (GMM), which is a GussianMixture class from sklearn.mixure. The function takes feature matrix X and other optional hyperparameters, fits the GMM, predicts cluster labels, and computes the silhouette score to evaluate clustering quality. It returns the cluster labels, the GMM model, and the silhouette score.

```
def run_em_gmm(X, n_components=2, covariance_type="tied", random_state=42):

    gmm = GaussianMixture(n_components=n_components, covariance_type=covariance_type,
            random_state=random_state)
    cluster_labels = gmm.fit_predict(X)

    silhouette = silhouette_score(X, cluster_labels)

    return cluster_labels, gmm, silhouette
```

The grid_search_gmm function performs a grid search for the best hyperparameters of a Gaussian Mixture Model (GMM) using the F1-score. It explores different parameter combinations for GMM fitting, evaluates clustering performance by F1-score, and returns the optimal hyperparameters with the highest F1-score. The parameters include the number of clusters, covariance type, maximum iterations, initialization method, and more, ensuring a completely search for the best hyperparameters configuration.

```
def grid_search_gmm(X, true_labels):
    """
    Performs a comprehensive grid search for the best hyperparameters in
            GaussianMixture using F1-score.

    Parameters:
    - X: NumPy array, the feature matrix.
    - true_labels: Actual ground truth labels.

    Returns:
    - best_params: Dictionary containing the best hyperparameters and F1-
score.
    """

    # Define the range of hyperparameters to test
    param_grid = {
        "n_components": [2],  # Number of clusters
        "covariance_type": ["full", "tied", "diag", "spherical"],  #
Covariance         matrix type
        "max_iter": [500, 1000],  # Maximum number of EM iterations
        "warm_start": [True, False],  # Whether to continue training from
            previous fit
        "tol": [1e-4, 1e-3],  # Convergence tolerance
        "init_params": ["kmeans", "random"],  # Initialization method
        "reg_covar": [1e-6, 1e-4, 1e-3],  # Regularization for covariance
matrix
        "n_init": [10, 20]  # Number of initializations
    }

    # Generate all possible combinations of hyperparameters
    param_combinations = list(product(*param_grid.values()))

    best_score = -1
    best_params = {}

    # Perform grid search
    for params in param_combinations:
        # Unpack the parameter combination
        n_components, cov_type, max_iter, warm_start, tol, init_params,
            reg_covar, n_init = params

        # Initialize and fit GMM model
        gmm = GaussianMixture(
            n_components=n_components,
            covariance_type=cov_type,
            random_state=42,
            max_iter=max_iter,
            warm_start=warm_start,
            tol=tol,
            init_params=init_params,
            reg_covar=reg_covar,
            n_init=n_init
        )

        gmm.fit(X)
        cluster_labels = gmm.predict(X)

        # Compute the F1-score
        class_report, f1_weighted_avg, recall_class0, recall_class1,
                 recall_weighted_avg = compute_f1_recall(true_labels,
                        cluster_labels)

        # Update the best parameters if the current F1-score is better
        if f1_weighted_avg > best_score:
            best_score = f1_weighted_avg
            best_params = {
                "n_components": n_components,
                "covariance_type": cov_type,
                "max_iter": max_iter,
                "warm_start": warm_start,
                "tol": tol,
                "init_params": init_params,
                "reg_covar": reg_covar,
                "n_init": n_init,
                "f1_score": f1_weighted_avg
            }

    return best_params
```

5.2. Support Vector Machines (SVM)

The data set was split into a training set(80%) and a test set(20%) using stratified sampling to account for class imbalance.

```
# Split data
X = df.drop(columns=["Class"])  # Features
y = df["Class"]  # Target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)
```

The function run_svm was defined to train SVM algorithm, using a SVC class from sklearn.svm and its hyperparameters. Stratified K-Fold cross-validation was implemented to ensure balanced class distributions across folds, addressing SVM's sensitivity to imbalanced datasets. The model was trained using weighted recall as the evaluation metric to prioritize performance across all classes.

```
def run_svm(kernel='rbf', C=0.1, gamma='scale', random_state=42):
    svm_model = SVC(kernel=kernel, C=C, gamma=gamma,
random_state=random_state)

    # Perform 5-fold cross-validation
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
    cv_recall = cross_val_score(svm_model, X_train, y_train, cv=cv,
                             scoring='recall_weighted')

    svm_model.fit(X_train, y_train)

    return cv_recall, svm_model
```

This code visualizes the classification results of the SVM model using PCA. The function visualize_pca takes the PCA-transformed data (X_pca) and predicted class labels (y_pred), which are obtained by using the trained SVM to predict the entire dataset. It then creates a scatter plot, where different colors represent different predicted classes. KDE contours are overlais to show the density distribution of each class.

```
def visualize_pca(X_pca, y_pred):
    df_plot = pd.DataFrame(X_pca, columns=['PCA1', 'PCA2'])
    df_plot['Predicted Class'] = y_pred

    class_colors = sns.color_palette("husl", len(np.unique(y_pred)))

    plt.figure(figsize=(8, 6))
    ax = sns.scatterplot(x='PCA1', y='PCA2', hue='Predicted Class',
data=df_plot,    palette=class_colors, alpha=0.8)

    for i, cls in enumerate(np.unique(y_pred)):
        class_data = df_plot[df_plot['Predicted Class'] == cls][['PCA1',
           'PCA2']]
        if len(class_data) > 1:
            sns.kdeplot(x=class_data['PCA1'], y=class_data['PCA2'],
                        levels=5, color=class_colors[i], linewidths=2,
alpha=0.8,                   ax=ax)
    plt.xticks(np.arange(int(df_plot['PCA1'].min()) - 2,
      int(df_plot['PCA1'].max()) + 2, 2))
    plt.yticks(np.arange(int(df_plot['PCA2'].min()) - 2,
      int(df_plot['PCA2'].max()) + 2, 2))

    plt.title("PCA Visualization of SVM Predictions", fontsize=14,
      fontweight='bold')
    plt.xlabel("Principal Component 1", fontsize=12)
    plt.ylabel("Principal Component 2", fontsize=12)
    plt.legend(title="Predicted Class", fontsize=10, loc="upper right",
       frameon=True)
    plt.show()

# Predictions on the whole dataset (not just a test set)
y_pred = svm_model.predict(X)

# Apply PCA to the whole dataset
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

visualize_pca(X_pca, y_pred)
```

Appendix

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.svm import SVC
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split, cross_val_score,
StratifiedKFold, GridSearchCV
from sklearn.metrics import recall_score, confusion_matrix, f1_score
```

```
# import data into a pandas dataframe
df = pd.read_csv("vertebral_column_data.csv", encoding="utf-8", sep=',')
df
```

```
# Check for missing values
print("Missing Values:\n", df.isnull().sum())

# # Encode categorical labels
# df["Class"] = LabelEncoder().fit_transform(df["Class"])  # Converts 'AB'
to 0, 'Normal' to 1, etc.
df["Class"] = df["Class"].map({'NO': 0, 'AB': 1})
```

```
# Create box plots for each feature
plt.figure(figsize=(8, 6))
sns.boxplot(data=df.drop(columns=["Class"]))
plt.xticks(rotation=45)
plt.title("Box Plots of Features")
plt.tight_layout()
plt.show()
```

```
# Create plot to visualize the transformed feature
plt.figure(figsize=(8, 6))
sns.histplot(data=df, x="Grade of Spondylolisthesis", kde=True)
plt.title("Distribution of Log-Transformed Grade of Spondylolisthesis")
plt.xlabel("Log(Grade of Spondylolisthesis + 1)")
plt.ylabel("Count")
plt.show()
```

```
Q1 = df.drop(columns=["Class"]).quantile(0.25)
Q3 = df.drop(columns=["Class"]).quantile(0.75)
IQR = Q3 - Q1

# Define valid range for non-outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Create a mask to filter out outliers
outlier_mask = ~((df.drop(columns=["Class"]) < lower_bound) |
(df.drop(columns=["Class"]) > upper_bound)).any(axis=1)

# Apply mask to remove outliers from both features and target
df = df[outlier_mask].reset_index(drop=True)
df
```

```
df
```

```
skewness = df["Grade of Spondylolisthesis"].skew()
print(f"Skewness of Grade of Spondylolisthesis: {skewness:.3f}")
```

```
# Normalize numerical features (optional, but helps for some models)
scaler = StandardScaler()
features = df.columns[:-1]  # All except the Class column
df[features] = scaler.fit_transform(df[features])
```

```
# Create box plots for each feature
plt.figure(figsize=(8, 6))
sns.boxplot(data=df.drop(columns=["Class"]))
plt.xticks(rotation=45)
plt.title("Box Plots of Features")
plt.tight_layout()
plt.show()
```

```
df.describe()
```

```
# Split data
X = df.drop(columns=["Class"])  # Features
y = df["Class"]  # Target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)
```

```
def run_svm(kernel='rbf', C=0.1, gamma='scale', random_state=42):
    svm_model = SVC(kernel=kernel, C=C, gamma=gamma,
random_state=random_state)

    # Perform 5-fold cross-validation
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
    cv_recall = cross_val_score(svm_model, X_train, y_train, cv=cv,
scoring='recall_weighted')

    svm_model.fit(X_train, y_train)

    return cv_recall, svm_model

# Run SVM model and get cross-validation results
cv_recall, svm_model = run_svm()

# Print cross-validation performance
print("\n5-Fold Cross-Validation Results:")
print(f"Mean Recall Score: {cv_recall.mean():.4f}")

# Make predictions on the test set
y_pred = svm_model.predict(X_test)

# Print test set performance
print("\nPerformance on Test Set:")
print(f"Recall (Class 1): {recall_score(y_test, y_pred,
average='binary'):.4f}")
print(f"Recall (Weighted-Averaged): {recall_score(y_test, y_pred,
average='weighted'):.4f}")
print(f"F1-Score (Weighted-Averaged): {f1_score(y_test, y_pred,
average='weighted'):.4f}")
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

```
def visualize_pca(X_pca, y_pred):
    df_plot = pd.DataFrame(X_pca, columns=['PCA1', 'PCA2'])
    df_plot['Predicted Class'] = y_pred

    class_colors = sns.color_palette("husl", len(np.unique(y_pred)))

    plt.figure(figsize=(8, 6))
    ax = sns.scatterplot(x='PCA1', y='PCA2', hue='Predicted Class',
data=df_plot, palette=class_colors, alpha=0.8)

    for i, cls in enumerate(np.unique(y_pred)):
        class_data = df_plot[df_plot['Predicted Class'] == cls][['PCA1',
'PCA2']]
        if len(class_data) > 1:
            sns.kdeplot(x=class_data['PCA1'], y=class_data['PCA2'],
                                levels=5, color=class_colors[i], linewidths=2,
alpha=0.8, ax=ax)

     plt.xticks(np.arange(int(df_plot['PCA1'].min()) - 2,
int(df_plot['PCA1'].max()) + 2, 2))
     plt.yticks(np.arange(int(df_plot['PCA2'].min()) - 2,
int(df_plot['PCA2'].max()) + 2, 2))


     plt.title("PCA Visualization of SVM Predictions", fontsize=14,
fontweight='bold')
     plt.xlabel("Principal Component 1", fontsize=12)
     plt.ylabel("Principal Component 2", fontsize=12)
     plt.legend(title="Predicted Class", fontsize=10, loc="upper right",
frameon=True)
     plt.show()

# Predictions on the whole dataset (not just a test set)
y_pred = svm_model.predict(X)

# Apply PCA to the whole dataset
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

visualize_pca(X_pca, y_pred)
```

```
def run_knn(n_neighbors=None):
    if n_neighbors is None:

        n_neighbors = [1, 3, 5, 7, 9, 11]

    param_grid = {'n_neighbors': n_neighbors}

    knn = KNeighborsClassifier()

    # Perform 5-fold cross-validation
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
    grid_search = GridSearchCV(knn, param_grid, cv=cv,
scoring='recall_weighted', n_jobs=-1, verbose=1)

    grid_search.fit(X_train, y_train)

    return grid_search

grid_search = run_knn()

# Display the best parameters and best CV score
print("\nBest Hyperparameters for KNN:", grid_search.best_params_)
print(f"Best Cross-Validation Recall Score:
{grid_search.best_score_:.4f}")

# Get the best model from Grid Search
best_knn = grid_search.best_estimator_

# Make predictions on the test set
y_pred = best_knn.predict(X_test)

# Print test set performance
print("\nPerformance on Test Set:")
print(f"Recall (Weighted-Average): {recall_score(y_test, y_pred,
average='weighted'):.4f}")
print(f"Recall (Binary Class 1): {recall_score(y_test, y_pred,
average='binary'):.4f}")
print(f"F1-Score (Weighted-Average): {f1_score(y_test, y_pred,
average='weighted'):.4f}")
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

```
y_pred = best_knn.predict(X)

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

visualize_pca(X_pca, y_pred)
```

```
# Exract true lables (AB → 1, NO → 0) and assign it to a new variable
(true_labels_numeric)
true_labels_numeric = df["Class"].values  # AB → 1, NO → 0
```

```
# Import SpectralClustering to perform kernel k-means
from sklearn.cluster import SpectralClustering
# Import silhouette_score to generate silhouette scores to evaluate the
performance
from sklearn.metrics import silhouette_score

# Define a function to run the kernel k-means method
def run_kernel_kmeans_alt(X, n_clusters=2, gamma=0.1, random_state=42):
    """
    Runs Kernel K-Means clustering using Spectral Clustering.

    Parameters:
    - X: NumPy array or Pandas DataFrame (preprocessed feature matrix)
    - n_clusters: Number of clusters (default: 2)
    - gamma: Kernel coefficient for RBF (default: 0.1)
    - random_state: Random seed for reproducibility (default: 42)

    Returns:
    - cluster_labels: Array of cluster assignments for each sample
    - silhouette: Silhouette score indicating clustering quality
    """
    # Approximate Kernel K-Means using Spectral Clustering with
nearest_neighbors affinity
    model = SpectralClustering(n_clusters=n_clusters, # no. of clusters (2
in this case)
                               affinity="nearest_neighbors", # Type of
affinity (e.g., "rbf" or "nearest_neighbors")
                               n_neighbors=10, # Number of neighbors (used
only if affinity is "nearest_neighbors")
                               assign_labels="discretize", # Method for
assigning labels after clustering (e.g., "kmeans" or "discretize")
                               gamma=gamma, # Kernel coefficient
                               random_state=random_state, # Random seed
for reproducibility
                               n_init=5 # Number of initializations to run
before selecting the best clustering result
                              )

    # Fit and predict clusters and assign it to cluster_labels
    cluster_labels = model.fit_predict(X)

    # Compute silhouette score for evaluation and assign it to silhouette
    silhouette = silhouette_score(X, cluster_labels)

    return cluster_labels, silhouette
```

```
# Extract numerics only in the dataset and assign it to an array X
X = df.drop(columns=["Class"]).select_dtypes(include=['number']).values

# Run kernel k-means method using the function above with X
cluster_labels, silhouette = run_kernel_kmeans_alt(X, n_clusters=2,
gamma=0.5)

# Get the result of the clustering and Silhouette
print("Predicted Cluster Labels:", cluster_labels)
print(f"Silhouette Score: {silhouette:.4f}")
```

```
## Run the Kernel K-means without "Pelvic Incidence"
# Extract numerics only in the dataset and assign it to an array X2
```

```
X2 = df.drop(columns=["Pelvic Incidence",
"Class"]).select_dtypes(include=['number']).values

# Run kernel k-means method using the function above with X2
cluster_labels2, silhouette2 = run_kernel_kmeans_alt(X2, n_clusters=2,
gamma=0.5)

# Get the result of the clustering and Silhouette
print("Predicted Cluster Labels:", cluster_labels2)
print(f"Silhouette Score: {silhouette2:.4f}")
```

```
# Import linear_sum_assignment and accuracy_score to compute the accuracy
of the clustering
from scipy.optimize import linear_sum_assignment
from sklearn.metrics import accuracy_score

# define a function to compute accuracy against the original cluster
labeling
def compute_clustering_accuracy(true_labels, predicted_labels):
    """
    Computes accuracy between true labels and predicted cluster
assignments.

    Parameters:
    - true_labels: Actual class labels
    - predicted_labels: Cluster assignments from Kernel K-Means

    Returns:
    - accuracy: Adjusted clustering accuracy
    """
    # Get unique label mappings of the original true clustering lables
    true_classes = np.unique(true_labels)
    # Get unique label mappings of the model-predicted clustering lables
    predicted_clusters = np.unique(predicted_labels)

    # Create confusion matrix with true lables and predicted lables
    contingency_matrix = np.zeros((len(true_classes),
len(predicted_clusters)), dtype=int)
    for i, true_label in enumerate(true_classes):
        for j, cluster_label in enumerate(predicted_clusters):
            contingency_matrix[i, j] = np.sum((true_labels == true_label)
& (predicted_labels == cluster_label))

    # Solve the assignment problem using Hungarian Algorithm
    row_ind, col_ind = linear_sum_assignment(-contingency_matrix)  #
Maximization problem
```

```
    # Compute accuracy based on best mapping obtained from the contingency
matrix
    best_mapping = {predicted_clusters[j]: true_classes[i] for i, j in
zip(row_ind, col_ind)}
    mapped_preds = np.array([best_mapping[label] for label in
predicted_labels])

    return accuracy_score(true_labels, mapped_preds)
# Import classification_report and confusion_matrix to compute F1 and
recall scores
from sklearn.metrics import classification_report, confusion_matrix

# Define a function to compute F1 and recall scores
def compute_f1_recall(true_labels, predicted_labels):

    # Compute Confusion Matrix and Classification Report with the true
labels and predicted labels
    conf_matrix = confusion_matrix(true_labels_numeric, predicted_labels)
    class_report = classification_report(true_labels_numeric,
predicted_labels, output_dict=True)

    # Extract and print F1 and Recall scores for each class
    f1_weighted_avg = class_report["weighted avg"]["f1-score"] # Weighted
F1
    recall_class0 = class_report["0"]["recall"]  # Recall for class 0
    recall_class1 = class_report["1"]["recall"]  # Recall for class 1
    recall_weighted_avg = class_report["weighted avg"]["recall"]  #
Weighted Recall

    return class_report, f1_weighted_avg, recall_class0, recall_class1,
recall_weighted_avg
```

```
# Compute accuracy using the above function with true and predicted labels
accuracy = compute_clustering_accuracy(true_labels_numeric,
cluster_labels)
print(f"Clustering Accuracy: {accuracy:.4f}")

# Compute F1 and Recall scores using the above function with true and
predicted labels
class_report, f1_weighted_avg, recall_class0, recall_class1,
recall_weighted_avg = compute_f1_recall(true_labels_numeric,
cluster_labels)

# Get the obtained scores
print(f"Weighted F1-score: {f1_weighted_avg:.4f}")  # Weighted F1-score
print(f"Weighted Recall: {recall_weighted_avg:.4f}")  # Weighted Recall
print(f"Class1 Recall: {recall_class1:.4f}")  # Weighted Recall
```

```
# Compute accuracy using the above function with true and predicted labels
(dataset w/o Pelvic Incidence)
accuracy2 = compute_clustering_accuracy(true_labels_numeric,
cluster_labels2)
print(f"Clustering Accuracy: {accuracy2:.4f}")

# Compute F1 and Recall scores using the above function with true and
predicted labels (dataset w/o Pelvic Incidence)
class_report2, f1_weighted_avg2, recall_class02, recall_class12,
recall_weighted_avg2 = compute_f1_recall(true_labels_numeric,
cluster_labels2)

# Get the obtained scores
print(f"Weighted F1-score: {f1_weighted_avg2:.4f}")  # Weighted F1-score
print(f"Weighted Recall: {recall_weighted_avg2:2.4f}")  # Weighted Recall
print(f"Class1 Recall: {recall_class12:.4f}")  # Weighted Recall
```

```
# Import product
from itertools import product

# Define a function to grid search the optimal hyperparameters of the
model
def tune_spectral_clustering(X, true_labels):
    """
    Performs a comprehensive grid search for the best hyperparameters in
Spectral Clustering using F1-score.

    Parameters:
    - X: NumPy array, the feature matrix.
    - true_labels: Actual ground truth labels.

    Returns:
    - best_params: Dictionary containing the best hyperparameters and F1-
score.
    """

    # Define the range of hyperparameters to test
    param_grid = {
        "n_clusters": [2],  # Number of clusters
        "affinity": ["nearest_neighbors", "rbf"],  # Similarity
computation method
```

```
        "n_neighbors": [10, 30, 50],  # Number of neighbors (for
nearest_neighbors affinity)
        "assign_labels": ["kmeans", "discretize"],  # Label assignment
method
        "gamma": [0.01, 0.1, 0.5, 1.0, 2.0],  # RBF Kernel Gamma values
        "random_state": [42],  # Set random state for reproducibility
        "n_init": [5, 10]  # Number of initializations
    }

    # Generate all possible combinations of hyperparameters
    param_combinations = list(product(*param_grid.values()))

    best_score = -1
    best_params = {}

    # Perform grid search
    for params in param_combinations:
        # Unpack the parameter combination
        n_clusters, affinity, n_neighbors, assign_labels, gamma,
random_state, n_init = params

        # Skip invalid combinations (n_neighbors is only applicable for
"nearest_neighbors" affinity)
        if affinity != "nearest_neighbors" and n_neighbors != 10:
            continue

        # Initialize and fit Spectral Clustering model
        model = SpectralClustering(
            n_clusters=n_clusters,
            affinity=affinity,
            n_neighbors=n_neighbors if affinity == "nearest_neighbors"
else 50,
            assign_labels=assign_labels,
            gamma=gamma if affinity == "rbf" else 0.1,
            random_state=random_state,
            n_init=n_init
        )

        # fit the model with X and assign it to cluster_labels
        cluster_labels = model.fit_predict(X)

        # Compute the F1 and recall scores
        class_report, f1_weighted_avg, recall_class0, recall_class1,
recall_weighted_avg = compute_f1_recall(true_labels, cluster_labels)

        # Update the best parameters if the current F1-score is better
        if f1_weighted_avg > best_score:
            best_score = f1_weighted_avg
            best_params = {
                "n_clusters": n_clusters,
                "affinity": affinity,
                "n_neighbors": n_neighbors if affinity ==
"nearest_neighbors" else 50,
                "assign_labels": assign_labels,
                "gamma": gamma if affinity == "rbf" else 0.1,
                "random_state": random_state,
                "n_init": n_init,
                "f1_score": f1_weighted_avg
            }

    return best_params
```

```
# Run hyperparameter tuning using the above function with X against true
labels
best_params = tune_spectral_clustering(X, true_labels_numeric)

# Print best parameters
print("\nBest Hyperparameters:")
print(best_params)
# Run hyperparameter tuning using the above function with X2 against true
labels (dataset w/o Pelvic Incidence)
best_params2 = tune_spectral_clustering(X2, true_labels_numeric)

# Print best parameters (dataset w/o Pelvic Incidence)
print("\nBest Hyperparameters:")
print(best_params2)
```

```
# import the package to run EM algorithm
from sklearn.mixture import GaussianMixture

# define a function to run EM algorithm
def run_em_gmm(X, n_components=2, covariance_type="tied",
random_state=42):
    """
    Runs Expectation-Maximization (EM) algorithm using Gaussian Mixture
Model (GMM).

    Parameters:
    - X: NumPy array or Pandas DataFrame (preprocessed feature matrix)
    - n_components: Number of clusters (default: 2)
    - covariance_type: Type of covariance matrix ('full', 'tied', 'diag',
'spherical')
    - random_state: Random seed for reproducibility

    Returns:
    - cluster_labels: Cluster assignments for each sample
    - gmm: Fitted GMM model
    - silhouette: Silhouette score for cluster evaluation
    """
    # Initialize and fit Gaussian Mixture Model (GMM)
    gmm = GaussianMixture(n_components=n_components, # Number of clusters
(Gaussian components)
                          covariance_type=covariance_type, # Type of
covariance matrix ("full", "tied", "diag", or "spherical")
                          random_state=random_state, # Random seed for
reproducibility
                          max_iter=500, # Maximum number of EM iterations
                          warm_start=True, # If True, reuses previous
parameters for initialization in subsequent fits
                          tol=0.0001, # Convergence threshold; stops EM
when log-likelihood improvement is below this value
                          init_params="kmeans", # Initialization method;
"kmeans" uses K-Means clustering for initial component means
                          reg_covar=0.001, # Small regularization term
added to covariance to prevent numerical instability
                          n_init=10 # Number of times to run GMM with
different initializations and choose the best
                          )

    # fit the model with X and assign the prediction result in
cluster_labels
    cluster_labels = gmm.fit_predict(X)

    # Compute silhouette score to evaluate clustering quality
    silhouette = silhouette_score(X, cluster_labels)

    return cluster_labels, gmm, silhouette

# import the package to run EM algorithm
from matplotlib.patches import Ellipse

# define a function to visualize the clusters with Gaussian ellipses
def visualize_gmm_clusters_with_kde(X, cluster_labels):
    """
    Visualizes GMM clustering results using PCA for dimensionality
reduction,
    with KDE contours overlaid.

    Parameters:
    - X: NumPy array or Pandas DataFrame (preprocessed feature matrix)
    - cluster_labels: Cluster assignments from GMM

    Returns:
    - None (Displays a scatter plot with KDE contours)
    """
    # Reduce dimensions using PCA
    pca = PCA(n_components=2)
    X_pca = pca.fit_transform(X)

    # Create DataFrame for visualization
    df_plot = pd.DataFrame(X_pca, columns=['PCA1', 'PCA2'])
    df_plot['Cluster'] = cluster_labels

    # Define color palette
    cluster_colors = sns.color_palette("husl",
len(np.unique(cluster_labels)))

    # Create scatter plot with the data with PCA axises
    plt.figure(figsize=(8, 6))
    ax = sns.scatterplot(x='PCA1', y='PCA2', hue='Cluster', data=df_plot,
palette=cluster_colors, alpha=0.8)

    # Overlay KDE contours for each cluster over PCA space
    for i, cluster in enumerate(np.unique(cluster_labels)):
        cluster_data = df_plot[df_plot['Cluster'] == cluster][['PCA1',
'PCA2']]
        if len(cluster_data) > 1:  # Ensure there's enough data
            sns.kdeplot(x=cluster_data['PCA1'], y=cluster_data['PCA2'],
                        levels=5, color=cluster_colors[i], linewidths=2,
alpha=0.8, ax=ax)

    plt.title("GMM Clustering (PCA Reduced) with KDE Contours")
    plt.legend(title="Cluster")
    plt.show()
```

```
# Run EM algorithm using the function above with X
gmm_cluster_labels, gmm_model, gmm_silhouette1 = run_em_gmm(X,
n_components=2)

# Get the predicted cluster lables and Silhouette score
print("Cluster Labels:", gmm_cluster_labels)
print(f"Silhouette Score: {gmm_silhouette1:.4f}")
```

```
# Visualize the clusters
visualize_gmm_clusters_with_kde(X, gmm_cluster_labels)
```

```
# Run EM algorithm using the function above with X2 without "Pelvic
Incidence"
gmm_cluster_labels2, gmm_model2, gmm_silhouette2 = run_em_gmm(X2,
n_components=2)

# Get the predicted cluster lables and Silhouette score
print("Cluster Labels:", gmm_cluster_labels2)
print(f"Silhouette Score: {gmm_silhouette2:.4f}")

# Visualize the clusters
visualize_gmm_clusters_with_kde(X2, gmm_cluster_labels2)
```

```
# Compute accuracy using the above defined function for EM algorithm
gmm_accuracy = compute_clustering_accuracy(true_labels_numeric,
gmm_cluster_labels)
print(f"Clustering Accuracy: {gmm_accuracy:.4f}")

# Compute F1 and Recall scores using the above defined function for EM
algorithm
gmm_class_report, gmm_f1_weighted_avg, gmm_recall_class0,
gmm_recall_class1, gmm_recall_weighted_avg =
compute_f1_recall(true_labels_numeric, gmm_cluster_labels)
print(f"Weighted F1-score: {gmm_f1_weighted_avg:.4f}")  # Weighted F1-
score
print(f"Weighted Recall: {gmm_recall_weighted_avg:.4f}")  # Weighted
Recall
print(f"Class1 Recall: {gmm_recall_class1:.4f}")  # Weighted Recall
```

```
# Compute accuracy using the above defined function for EM algorithm
without "Pelvic Incidence"
gmm_accuracy2 = compute_clustering_accuracy(true_labels_numeric,
gmm_cluster_labels2)
print(f"Clustering Accuracy: {gmm_accuracy2:.4f}")

# Compute F1 and Recall scores using the above defined function for EM
algorithm without "Pelvic Incidence"
gmm_class_report2, gmm_f1_weighted_avg2, gmm_recall_class02,
gmm_recall_class12, gmm_recall_weighted_avg2 =
compute_f1_recall(true_labels_numeric, gmm_cluster_labels2)
print(f"Weighted F1-score: {gmm_f1_weighted_avg2:.4f}")  # Weighted F1-
score
```

```
print(f"Weighted Recall: {gmm_recall_weighted_avg2:.4f}")  # Weighted
Recall
print(f"Class1 Recall: {gmm_recall_class12:.4f}")  # Weighted Recall
```

```
# import random
import random

# define a function to grid search optimal parameters for GaussianMixture
def grid_search_gmm(X, true_labels):
    """
    Performs a comprehensive grid search for the best hyperparameters in
GaussianMixture using F1-score.

    Parameters:
    - X: NumPy array, the feature matrix.
    - true_labels: Actual ground truth labels.

    Returns:
    - best_params: Dictionary containing the best hyperparameters and F1-
score.
    """

    # Define the range of hyperparameters to test
    param_grid = {
        "n_components": [2],  # Number of clusters
        "covariance_type": ["full", "tied", "diag", "spherical"],  #
Covariance matrix type
        "max_iter": [500, 1000],  # Maximum number of EM iterations
        "warm_start": [True, False],  # Whether to continue training from
previous fit
        "tol": [1e-4, 1e-3],  # Convergence tolerance
        "init_params": ["kmeans", "random"],  # Initialization method
        "reg_covar": [1e-6, 1e-4, 1e-3],  # Regularization for covariance
matrix
        "n_init": [10, 20]  # Number of initializations
    }

    # Generate all possible combinations of hyperparameters
    param_combinations = list(product(*param_grid.values()))

    best_score = -1
    best_params = {}

    # Perform grid search
    for params in param_combinations:
        # Unpack the parameter combination
        n_components, cov_type, max_iter, warm_start, tol, init_params,
reg_covar, n_init = params

        # Initialize and fit GMM model
        gmm = GaussianMixture(
            n_components=n_components,
            covariance_type=cov_type,
            random_state=42,
            max_iter=max_iter,
            warm_start=warm_start,
            tol=tol,
            init_params=init_params,
            reg_covar=reg_covar,
            n_init=n_init
        )

        gmm.fit(X)
        cluster_labels = gmm.predict(X)

        # Compute the F1-score
        class_report, f1_weighted_avg, recall_class0, recall_class1,
recall_weighted_avg = compute_f1_recall(true_labels, cluster_labels)

        # Update the best parameters if the current F1-score is better
        if f1_weighted_avg > best_score:
            best_score = f1_weighted_avg
            best_params = {
                "n_components": n_components,
                "covariance_type": cov_type,
                "max_iter": max_iter,
                "warm_start": warm_start,
                "tol": tol,
                "init_params": init_params,
                "reg_covar": reg_covar,
                "n_init": n_init,
                "f1_score": f1_weighted_avg
            }

    return best_params
```

```
# Run hyperparameter tuning of gmm using the above function with X against
true labels
best_gmm_params = grid_search_gmm(X, true_labels_numeric)

# Print best parameters
print("\nBest Hyperparameters for GMM:")
```

```
best_gmm_params
```

```
# Run hyperparameter tuning of gmm using the above function with X2
against true labels (w/o "Pelvic Incidence")
best_gmm_params2 = grid_search_gmm(X2, true_labels_numeric)

# Print best parameters
print("\nBest Hyperparameters for GMM:")
best_gmm_params2
```

```
# define a function to generate a comparison matirx among the method
performance
def compare_clustering_results():
    """
    Compares Kernel K-Means and GMM (EM Algorithm) in terms of F1 and
Recall scores.

    Parameters:
    - None

    Returns:
    - results_df: Pandas DataFrame containing comparison results
    """
    # Evaluate Kernel K-Means
    kmeans_acc = accuracy
    kmeans_silhouette = silhouette
    f1 = f1_weighted_avg
    recall1 = recall_weighted_avg
    recall_class_1 = recall_class1

    # Evaluate Kernel K-Means (without Pelvic Incidence)
    kmeans_acc2 = accuracy2
    kmeans_silhouette2 = silhouette2
    f12 = f1_weighted_avg2
    recall2 = recall_weighted_avg2
    recall_class_12 =recall_class12

    # Evaluate GMM (EM Algorithm)
    gmm_acc =  gmm_accuracy
    gmm_silhouette = gmm_silhouette1
    gmm_f1 = gmm_f1_weighted_avg
    gmm_recall = gmm_recall_weighted_avg
    gmm_recall_class_1 = gmm_recall_class1

    # Evaluate GMM (EM Algorithm) (without Pelvic Incidence)
    gmm_acc2 =  gmm_accuracy2
    gmm_silhouette_2 = gmm_silhouette2
    gmm_f12 = gmm_f1_weighted_avg2
    gmm_recall2 = gmm_recall_weighted_avg2
    gmm_recall_class_12 = gmm_recall_class12

    # Create summary table
    results = {
        "Method": ["Kernel K-Means",
                   "Kernel K-Means (w/o Pelvic Incidence)",
                   "GMM (EM Algorithm)",
                   "GMM (EM Algorithm) (w/o Pelvic Incidence)"],
        "Accuracy": [kmeans_acc, kmeans_acc2, gmm_acc, gmm_acc2],
        "Silhouette Score": [kmeans_silhouette,  kmeans_silhouette2,
gmm_silhouette, gmm_silhouette_2],
        "F1 Score": [f1,  f12, gmm_f1, gmm_f12],
        "Recall (Weighted)": [recall1,  recall2, gmm_recall, gmm_recall2],
        "Recall Class1": [recall_class_1,  recall_class_12,
gmm_recall_class_1, gmm_recall_class_12]

    }

    results_df = pd.DataFrame(results)

    return results_df
```

```
# print the result table
compare_clustering_results()
```

Reference

Bishop, C. M. (2006). *Pattern recognition and machine learning,* Springer-Verlag. Available at: https://www.microsoft.com/en-us/research/wp-content/uploads/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf.

Bouveyron, C. and Brunet-Saumard, C. (2014). 'Model-based clustering of high-dimensional data: A review', *Computational Statistics & Data Analysis*, 71, pp.52-78. Available at: https://doi.org/10.1016/j.csda.2012.12.008.

Celeux, G. and Govaert, G. (1992). 'A classification EM algorithm for clustering and two stochastic versions', *Computational Statistics & Data Analysis*, 14(3), pp.315-332. Available at: https://doi.org/10.1016/0167-9473(92)90042-E.

Cortes, C. and Vapnik, V. (1995). 'Support-vector networks', *Machine Learning*, 20, pp.273-297. Available at: https://doi.org/10.1023/A:1022627411411.

Halder, R. H., et al., (2024). ‘Enhancing K-nearest neighbor algorithm: a comprehensive review and performance analysis of modifications’, *Journal of Big Data*, 11(113). Available at: https://doi.org/10.1186/s40537-024-00973-y.

James, G. et al. (2013). ‘An introduction to statistical learning’, *Springer,*

Li, W. et al. (2015). ‘Outlier detection and removal improves accuracy of machine learning approach to multispectral burn diagnostic imaging’. *Journal of Biomedical Optics*, 20(12), p.121305. Available at: https://doi.org/10.1117/1.JBO.20.12.121305.

Rezapour, M. et al. (2024). ‘Employing machine learning to enhance fracture recovery insights through gait analysis’. *Journal of Orthopaedic Research*, 42(8), pp.1748-1761. Available at: https://doi.org/10.1002/jor.25837.

Rouzrokh, P. et al. (2024). ‘THA-AID: Deep learning tool for total hip arthroplasty automatic implant detection with uncertainty and outlier quantification’. *The Journal of Arthroplasty*, 39(4), pp.966-973.e17. Available at: https://doi.org/10.1016/j.arth.2023.09.025.

Scikit-learn Developers. (2024). ‘Expectation-Maximization for Gaussian Mixture Models’, *Scikit-learn documentation*. Available at: https://scikit-learn.org/stable/modules/mixture.html#expectation-maximization [Accessed 13 Mar. 2025].

Sokolova, M., Japkowicz, N. and Szpakowicz, S. (2006). ‘Beyond accuracy, F-score and ROC: A family of discriminant measures for performance evaluation’. In: Sattar, A. and Kang, B., eds. *AI 2006: Advances in Artificial Intelligence*. Lecture Notes in Computer Science, vol. 4304. Berlin, Heidelberg: Springer. Available at: https://doi.org/10.1007/11941439_114.

Straub, B.M. (2018). ‘A study of approximations and data reduction techniques for kernel regularized least squares’. Master's thesis, Pennsylvania State University. Available at: https://etda.libraries.psu.edu/catalog/15177bbs5179 [Accessed 14 Mar. 2025].

University of Manchester. (2025). ‘Week 4 Lecture Slides: Supervised Classification’.

Wiwie, C., Baumbach, J. and Röttger, R. (2015). ‘Comparing the performance of biomedical clustering methods’. *Nature Methods*, 12(11), p.1033+. Available at: http://dx.doi.org.manchester.idm.oclc.org/10.1038/NMETH.3583.

Rogers, S. and Girolami, M. (2016). ‘A first course in machine learning’. 2nd edn. Chapman & Hall/CRC.

Code

The full implementation is available at:
[GitHub Repository]
https://github.com/kokosan123/stats2_coursework/blob/main/final_code.ipynb