CE672 Term Paper Report:

Supervised Classification Techniques for Remotely Sensed Data

Department of Civil Engineering

Indian Institute of Technology Kanpur

Aayush Sidana	(220023)
Amipriya Anand	(220122)
Hardik Gupta	(220419)
Kumar Aditya	(220592)

2024-25 IInd Sem

Contents

List of Figures

List of Tables

Abstract

This study presents a supervised classification approach applied to the Indian Pines hyperspectral dataset using four widely used classifiers: Gaussian Maximum Likelihood (implemented via both Naive Bayes and Quadratic Discriminant Analysis), Minimum Distance to Means, K-Nearest Neighbors, and Parallelepiped. The primary focus is on the accurate derivation of thematic land cover maps from hyperspectral data, which is notably challenging due to the dataset's high dimensionality and high spectral similarity among classes.

In addition to classification using the raw spectral bands, the study explores advanced feature extraction strategies to enhance performance. Dimensionality reduction via Principal Component Analysis (PCA) is employed to mitigate the curse of dimensionality, and a robust spatial feature extraction method based on Random Patches (RPNet) is also utilized. These additional methods are integrated with the original spectral data to form enriched feature spaces.

Each classifier is trained using available ground truth, and the performance is quantitatively evaluated using overall accuracy, average accuracy, and the Kappa coefficient. The results indicate that while the primary classification approaches yield robust thematic maps, the integration of PCA and RPNet further improves performance. The comprehensive comparison of classifier performance across different feature spaces underscores the significance of both optimal classifier selection and advanced feature extraction techniques in hyperspectral image analysis.

1 Introduction

Digital image processing plays a crucial role in extracting meaningful information from remotely sensed data. One of the primary applications is image classification, which aims to assign each pixel in an image to a predefined thematic class (e.g., water, forest, urban). Supervised classification algorithms learn the spectral characteristics of these classes from user-defined training samples and then use this knowledge to classify the entire image.

Supervised classification is a widely used approach in remote sensing, where the analyst selects representative training samples for each land cover class based on prior knowledge or field data. These training samples are used to generate statistical signatures, which form the basis for classifying all other pixels in the image. The accuracy of supervised classification depends on the quality and representativeness of the training data, as well as the distinctness of the spectral signatures among classes [1, 3, 4, 6]. In practice, a combination of spectral, spatial, and sometimes temporal features is used to improve classification performance, especially in complex landscapes or when classes have similar spectral responses.

This project focuses on implementing and comparing six established supervised classification techniques:

- Gaussian Maximum Likelihood (GML) classification (including both Naive Bayes and Quadratic Discriminant Analysis variants)
- Parallelepiped classification
- Minimum Distance to Means classification
- K-Nearest Neighbors (KNN) classification
- Support Vector Machine (SVM) classification (**Additional**)

The classifiers are applied to the Indian Pines hyperspectral remote sensing dataset, a widely used benchmark in the field. To address the high dimensionality and spatial complexity of hyperspectral data, we evaluate three different feature extraction strategies: using the raw spectral bands, principal component analysis (PCA) for dimensionality reduction, and a Random Patches Network (RPNet) approach for spatial-spectral feature extraction. The resulting classified land cover maps are assessed using standard accuracy metrics such as overall accuracy, average accuracy, and the Kappa coefficient.

This comprehensive comparison aligns with the objectives of CE672, covering fundamental pattern recognition techniques in remote sensing data analysis, namely, **feature extraction, classification**, and **evaluation** [1]. The study also highlights the importance of integrating spatial information and advanced feature extraction methods to improve classification accuracy in challenging remote sensing scenarios.

2 Brief Literature Survey

The classification of remotely sensed imagery is a cornerstone of Earth observation, with extensive documentation available in academic literature [1, 3, 4]. Foundational supervised techniques, including Minimum Distance to

Means, Parallelepiped, and Gaussian Maximum Likelihood (GML), are thoroughly explained in standard remote sensing textbooks such as Mather and Koch (2011) and Richards (2013), which detail their theoretical underpinnings and practical implementations [3, 4]. Jensen (2004) also provides valuable insights into these methods [6]. The K-Nearest Neighbors (KNN) algorithm, a non-parametric approach often covered in pattern recognition literature like Duda, Hart, and Stork (2001), has gained traction for remote sensing applications due to its flexibility [5]. Reviews like those by Verma et al. (2021) and Kumar et al. (2015) survey various classification methods, including those used in this project, highlighting their application domains and relative performance.

In recent years, the field has seen a rapid evolution with the integration of advanced machine learning and deep learning techniques. Support Vector Machines (SVMs), as surveyed by Mountrakis et al. (2011), have become a mainstay for hyperspectral image classification due to their ability to handle high-dimensional data and complex class boundaries [3, 4]. More recently, deep neural networks, including Convolutional Neural Networks (CNNs) and Deep Multilayer Neural Networks (DMNs), have demonstrated superior performance in extracting spatial-spectral features and improving classification accuracy [12, **?**]. The emergence of foundation models and self-supervised learning strategies, such as masked autoencoders and contrastive learning, is further pushing the boundaries of remote sensing image analysis, enabling robust feature extraction from large-scale, unlabeled datasets.

Hyperspectral datasets, such as the **Indian Pines** scene used in this study, present unique challenges due to their high dimensionality and spectral redundancy [11, 13]. This dataset, captured by the AVIRIS sensor, is a widely used benchmark for evaluating classification algorithms. Accurate performance evaluation necessitates robust assessment methods; the confusion matrix, overall accuracy, and the Kappa coefficient are standard metrics discussed by Congalton and Green (2009) [8] and are essential outputs in the course syllabus. Preprocessing steps, such as noise band removal, normalization, and dimensionality reduction (e.g., PCA), are critical for improving data quality and classifier performance [4].

While this project focuses on traditional supervised methods, the literature highlights a clear trend toward hybrid and spatial-spectral approaches. Techniques such as Random Patches Network (RPNet) leverage spatial context by extracting local features, which, when combined with spectral information, can significantly enhance classification results [14]. Object-based image analysis and advanced feature selection strategies are also gaining prominence, especially for high-resolution and complex scenes. The ongoing development of robust, scalable, and interpretable models remains a central research focus, with the goal of improving land cover mapping, environmental monitoring, and decision support in remote sensing applications.

3 Theory

Supervised classification is a pattern recognition approach involving two major stages: training, where spectral characteristics of known classes are learned from labeled data, and classification, where these characteristics are used to assign class labels to unknown pixels [2, 3, 4]. This process is fundamental in remote sensing for generating thematic maps from multispectral or hyperspectral imagery, with effectiveness depending on classifier selection, feature extraction, and training data quality.

3.1 Minimum Distance to Means

The Minimum Distance to Means classifier assigns a pixel $\mathbf{x}$ to the class whose mean vector $\boldsymbol{\mu}_i$ is closest, using Euclidean distance [2, 5]:

$$d(\mathbf{x}, \boldsymbol{\mu}_i) = \sqrt{(\mathbf{x} - \boldsymbol{\mu}_i)^\top (\mathbf{x} - \boldsymbol{\mu}_i)}$$

Decision rule:

$$\mathbf{x} \in \omega_k \quad \text{if} \quad k = \arg\min_i d(\mathbf{x}, \boldsymbol{\mu}_i)$$

While computationally efficient, it assumes spherical class distributions and ignores covariance structures.

3.2 Parallelepiped

This non-parametric method defines hyper-rectangular decision boundaries using per-band extrema [2, 6]:

$$\mathbf{x} \in \omega_i \quad \text{iff} \quad x_j \in [\min_{ij}, \max_{ij}],\ \forall j = 1, ..., d$$

Susceptible to unclassified regions and correlated features, it serves as baseline for simple datasets.

3.3 Gaussian Maximum Likelihood (GML)

The GML classifier models classes as multivariate Gaussian distributions, with two variants implemented:

Naive Bayes: Assumes feature independence ($\mathbf{\Sigma}_i$ diagonal):

$$g_i(\mathbf{x}) = -\frac{1}{2}\sum_{j=1}^{d}\left(\frac{(x_j-\mu_{ij})^2}{\sigma_{ij}^2} + \ln\sigma_{ij}^2\right) + \ln P(\omega_i)$$

Quadratic Discriminant Analysis (QDA): Full covariance estimation:

$$g_i(\mathbf{x}) = -\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_i)^\top\mathbf{\Sigma}_i^{-1}(\mathbf{x}-\boldsymbol{\mu}_i) - \frac{1}{2}\ln|\mathbf{\Sigma}_i| + \ln P(\omega_i)$$

Both require $n > d+1$ samples to avoid singular covariance matrices [3].

3.4 K-Nearest Neighbors (KNN)

A non-parametric approach leveraging local similarity:

$$\omega(\mathbf{x}) = (\{\omega(y_j) \mid y_j \in \mathcal{N}_K(\mathbf{x})\})$$

where $\mathcal{N}_K(\mathbf{x})$ denotes the K nearest neighbors. Performance depends on K selection and distance metric [5].

3.5 Support Vector Machines (SVM) (Additional)

SVM finds optimal separating hyperplanes in high-dimensional space using kernel tricks. The decision function for RBF kernel:

$$f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{N}\alpha_i y_i \exp(-\gamma\|\mathbf{x}_i-\mathbf{x}\|^2) + b\right)$$

where γ controls kernel width and α_i are learned weights. Effective for high-dimensional data [12].

3.6 Feature Extraction (Additional)

3.6.1 Principal Component Analysis (PCA):

Reduces dimensionality by projecting data onto orthogonal eigenvectors:

$$\mathbf{Y} = \mathbf{X}\mathbf{W}$$

where $\mathbf{W}$ contains eigenvectors of $\mathbf{X}^\top\mathbf{X}$.

3.6.2 Random Patches Network (RPNet)[14]:

Extracts spatial-spectral features through:

1. PCA whitening: $\mathbf{X}_{white} = \text{PCA}(\mathbf{X}, p)$
2. Random patch extraction: $\mathcal{P} = \{\mathbf{P}_k \sim \mathcal{U}(\mathbf{X}_{white})\}_{k=1}^{K}$
3. Convolution and ReLU: $\mathbf{O}_l = \max(0, \mathbf{O}_{l-1} * \mathcal{P}_l)$
4. Feature stacking: $\mathbf{F} = [\mathbf{X}\|\mathbf{O}_1\|\cdots\|\mathbf{O}_L]$

RPNet enhances discriminability through hierarchical feature learning [14].

3.7 Accuracy Assessment

Performance is quantified using:

- **Overall Accuracy (OA):**

$$OA = \frac{\sum_{i=1}^{c} M_{ii}}{N} \times 100\%$$

- **Kappa Coefficient (κ):**

$$\kappa = \frac{N\sum_{i=1}^{c} M_{ii} - \sum_{i=1}^{c}(M_{i+}M_{+i})}{N^2 - \sum_{i=1}^{c}(M_{i+}M_{+i})}$$

where M is the confusion matrix [8].

4 Data Used and Study Area

The Indian Pines hyperspectral dataset, acquired on June 12, 1992 by the Airborne Visible/Infrared Imaging Spectrometer (AVIRIS) sensor, serves as the primary dataset for this study [11]. Key characteristics include:

- **Sensor:** AVIRIS (Airborne Visible/Infrared Imaging Spectrometer)
- **Location:** Agricultural test site in Northwestern Indiana, USA (Lat: 40.46°N, Lon: 86.99°W)
- **Image Dimensions:** 145×145 pixels
- **Spectral Characteristics:**
  - Original 224 bands (0.4-2.5 μm)
  - 20m spatial resolution
  - 10nm spectral resolution
  - Corrected version removes 4 noisy bands (104-108, 150-163, 200+), resulting in 200 clean bands [2]
- **Ground Truth:** 16 agricultural classes including corn, soybeans, wheat, and forest, with approximately 10,000 labeled pixels [11]
- **Data Files:**
  - `Indian_pines_corrected.mat`: Calibrated hyperspectral cube ($145\times145\times200$)
  - `Indian_pines_gt.mat`: 2D ground truth labels (145×145)

This dataset presents unique challenges for classification due to:

- High dimensionality (200 spectral bands)
- Spectral similarity between crop types
- Presence of mixed pixels at field boundaries
- Limited training samples for some classes

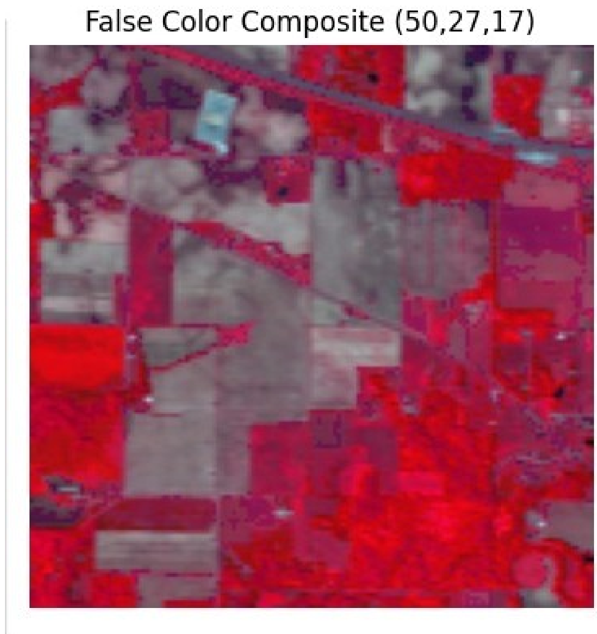

Figure 1: False Color Composite (FCC) of the dataset using bands (17,27,50). The near-infrared band (50) is displayed in red, red-edge band (27) in green, and visible band (17) in blue.

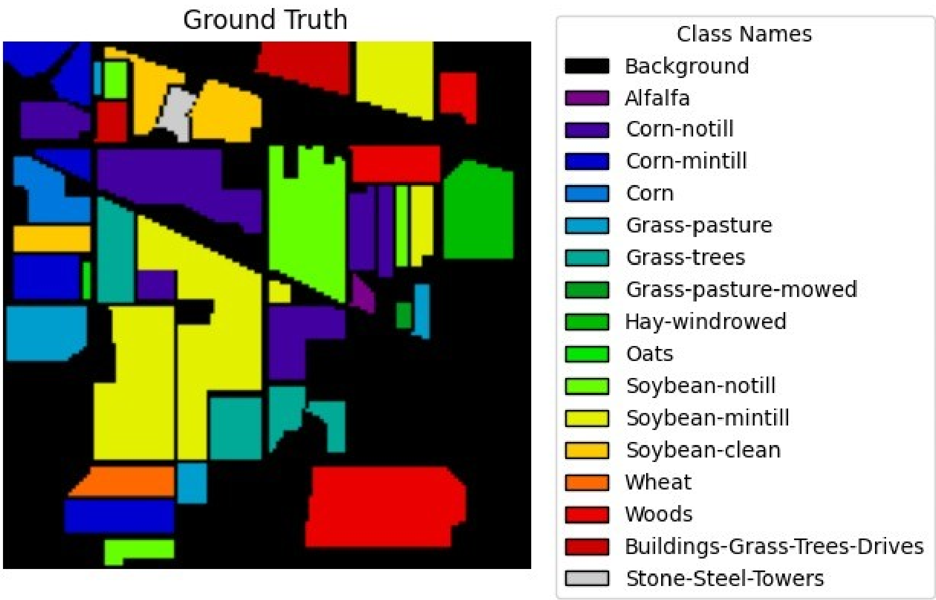

Figure 2: Ground truth map showing 16 agricultural classes. Class distribution follows the original labeling convention with class 0 representing unlabeled background pixels.

5 Methodology

This study implements and evaluates several supervised classification algorithms for hyperspectral imagery using a custom Graphical User Interface (GUI) built with Python's Tkinter library. The primary goal was to classify land cover types based on their spectral signatures, utilizing established techniques and exploring advanced feature extraction and classification methods for comparison.

5.1 Data Loading and Preparation

The workflow begins by loading two essential MATLAB (.mat) files provided by the user via the GUI:

- **Corrected Hyperspectral Image Cube:** A 3D NumPy array representing the hyperspectral data ($H \times W \times B$), where H and W are spatial dimensions and B is the number of spectral bands. The code assumes radiometrically corrected data.
- **Ground Truth Map:** A 2D NumPy array ($H \times W$) containing integer labels corresponding to known land cover classes for specific pixels.

The code automatically identifies the relevant data arrays within the loaded .mat files. An option is provided within the GUI to exclude background pixels (typically labeled as 0 in the ground truth) from the training process. A binary training mask is generated based on the ground truth map and this user selection.

5.2 Feature Extraction

To investigate the impact of feature representation on classification accuracy, three distinct feature extraction approaches were implemented, selectable via the GUI:

1. **Raw Spectral Features (None):** The baseline approach uses the original spectral vectors for each pixel directly as input features (dimensionality = B).
2. **Principal Component Analysis (PCA):** As an additional dimensionality reduction technique, PCA (`sklearn.decomposition.PCA`) was applied. The user specifies the desired number of principal components (p) via the GUI. PCA is fitted on the entire image dataset (reshaped to $N \times B$, where $N = H \times W$) to capture the main variance across all pixels. The data is then transformed into the lower-dimensional PCA feature space (dimensionality = p).

3. **Random Patches Network (RPNet):** Further exploring advanced unsupervised spatial-spectral feature extraction, RPNet was implemented based on the principles described in related literature and the provided proj3.py script. This method, requiring the `PyTorch` library, aims to learn spatial context through fixed convolutional filters derived from random patches of an initially PCA-reduced feature space. The steps include:

    - Initial dimensionality reduction and whitening of the input data using PCA (typically to 4 components, `apply_pca_and_whiten`).
    - Iterative application of RPNet layers (`run_rpnet_layers`, typically $L = 3$ layers):
        - Optional intermediate PCA and whitening between layers.
        - Extraction of k (e.g., 10) random patches (e.g., 20×20 spatial size) from the current feature map (`extract_random_patches_as_filters`).
        - Treating these patches as fixed filters in a 2D convolutional layer (`RPNetFixedLayer`, `torch.nn.functional.conv2d`) applied to the feature map.
    - Concatenation of the output feature maps from all L layers.
    - Combination (`combined_features`) of the derived RPNet spatial features with the original (scaled) spectral data, followed by a final scaling step, resulting in a combined spatial-spectral feature vector for each pixel.

5.3 Feature Scaling

Regardless of the chosen feature extraction method, robust feature scaling is crucial for many classifiers. A `sklearn.preprocessing.StandardScaler` is employed. **Critically**, the scaler is *fitted only* on the feature vectors corresponding to the training pixels (identified by the training mask). Subsequently, this fitted scaler is used to *transform both* the training feature set and the full image feature set (used for prediction), ensuring consistency and preventing data leakage from the prediction set into the scaling parameters.

5.4 Supervised Classification Models

A suite of standard supervised classifiers were implemented, along with an additional advanced model for comparison:

- **Minimum Distance to Mean:** Classifies pixels based on the minimum Euclidean distance to the mean feature vector of each class, computed from the scaled training data (`sklearn.neighbors.NearestCentroid`).
- **Parallelepiped:** A non-statistical classifier that defines hyper-rectangular decision boundaries based on the minimum and maximum feature values for each class observed in the scaled training data. A custom implementation checks if a test pixel's features fall within the bounds of any class.
- **Gaussian Maximum Likelihood (GML):** Implemented using two common probabilistic approaches assuming Gaussian class distributions:
    - *Naive Bayes variant (GML NB):* Assumes conditional independence between features (`sklearn.naive_bayes.GaussianNB`).
    - *Quadratic variant (GML QDA):* Models each class with a full quadratic decision boundary, estimating separate covariance matrices per class (`sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis`).
- **K-Nearest Neighbors (KNN):** A non-parametric instance-based learning algorithm (`sklearn.neighbors.KNeighborsClassifier`). The number of neighbors (k) is a user-defined parameter in the GUI (defaulting to 5).
- **Support Vector Machine (SVM):** As an additional, powerful classification technique, an SVM with a Radial Basis Function (RBF) kernel was included (`sklearn.svm.SVC`). Based on common practices and the reference script, fixed hyperparameters were used (`C=10`, `gamma='scale'`).

All classifiers are trained using the scaled training feature vectors and their corresponding ground truth labels.

5.5 Prediction and Evaluation

Once trained, the selected classifier predicts labels for *all* pixels in the image using the full (scaled) feature dataset derived from the chosen feature extraction method. The resulting flat prediction vector is reshaped into a 2D classification map ($H \times W$).

Performance evaluation is conducted by comparing the predicted labels specifically for the pixels used during training against their known ground truth labels. The following metrics are calculated using `scikit-learn`:

- **Overall Accuracy (OA):** The percentage of correctly classified training pixels.
- **Kappa Coefficient (κ):** A statistical measure of inter-rater agreement, correcting for chance agreement.
- **Confusion Matrix:** A table summarizing the classification performance for each class. (Calculated internally but not directly displayed in the final plot title).

5.6 Visualization and Implementation

The final classification map is displayed using `matplotlib.pyplot`, color-coded according to the ground truth classes. The plot title includes the chosen classifier, the feature extraction method used, and the calculated OA and Kappa values. The GUI provides controls for loading data, selecting methods, setting parameters (PCA components, KNN's k), and initiating the classification and display process. The core implementation relies on `NumPy` for numerical operations, `SciPy` for .mat file loading, `Scikit-learn` for PCA, scaling, classifiers, and metrics, `Matplotlib` for plotting, and optionally `PyTorch` for the RPNet feature extraction.

6 Results and Discussions

This section details the performance analysis of various supervised classification algorithms applied to the Indian Pines hyperspectral dataset, as implemented and evaluated in the accompanying Python code. The core objective was to classify land cover types using established methods and explore the potential benefits of advanced feature extraction techniques and an additional classification model (SVM).

6.1 Experimental Setup

The analysis utilized the corrected Indian Pines dataset. The ground truth data was used to define training and testing sets, with 80% of the labeled pixels (excluding the background class) reserved for testing and 20% for training, ensuring stratification by class. Classification performance was primarily evaluated based on Overall Accuracy (OA), Average Accuracy (AA), and the Kappa Coefficient (κ), calculated on the test set.

The following supervised classifiers were evaluated:

- Gaussian Maximum Likelihood (GML - QDA variant)
- Parallelepiped
- Minimum Distance to Mean (Nearest Centroid)
- K-Nearest Neighbors (KNN, with $k = 5$ determined via preliminary analysis)
- Support Vector Machine (SVM) with RBF kernel (C=10, gamma='scale')

These classifiers were applied to features derived using three different approaches:

1. **Raw Features (None):** Using the original spectral bands directly.
2. **PCA Features:** Using the first 4 Principal Components derived from the raw data. The choice of 4 components was informed by preliminary analysis balancing variance explained and classification performance (as shown in the notebook).
3. **RPNet Features:** Using combined spatial-spectral features generated by the Random Patches Network (RPNet) method (k=10 filters, patch size=20, L=3 layers), concatenated with the original spectral data, and subsequently scaled.

Standard scaling (`StandardScaler`) was applied to all feature sets, fitting on the training data and transforming both training and test sets.

6.2 Performance with Raw Spectral Features

Initial classification using the raw spectral features provided a baseline performance measure. The results showed considerable variability:

- The SVM classifier achieved the highest accuracy among all methods on raw data, with an Overall Accuracy (OA) of approximately 83.50% and a Kappa coefficient of 81.05%.
- KNN (k=5) also performed relatively well, achieving around 70.46% OA.
- Other traditional methods demonstrated significantly lower performance: Minimum Distance (approx. 42.17% OA), GML (approx. 39.40% OA), and Parallelepiped showing the poorest results (approx. 16.77% OA).

These baseline results, particularly the lower accuracies for GML, Parallelepiped, and Minimum Distance, suggested that relying solely on spectral information might be insufficient for accurate classification of this complex scene, motivating the exploration of feature extraction techniques.

6.3 Impact of PCA Feature Extraction

Applying PCA to reduce the dimensionality to 4 components yielded mixed results:

- GML (QDA) performance saw a substantial increase, reaching approximately 60.21% OA, likely due to the reduced dimensionality mitigating issues related to the curse of dimensionality or covariance matrix estimation.
- However, the performance of the previously best-performing classifiers, SVM and KNN, decreased when using only 4 PCA components (OA dropped to approx. 68.80% and 68.72%, respectively).
- Minimum Distance performance also slightly decreased (approx. 39.91% OA).
- Parallelepiped performance degraded significantly (approx. 3.80% OA), suggesting its sensitivity to the feature space transformation.

Overall, while PCA (n=4) benefited the GML classifier, it did not provide a universal improvement and negatively impacted the top-performing models from the baseline.

6.4 Impact of RPNet Spatial-Spectral Feature Extraction

The RPNet method, which combines learned spatial features with the original spectral information, demonstrated a profound positive impact on classification accuracy across the board:

- SVM achieved the highest accuracy observed in the study, reaching approximately 97.39% OA and a Kappa of 97.03%. This represents a significant improvement over both raw and PCA features.
- KNN (k=5) performance also saw a major boost, reaching approx. 91.05% OA, making it the second-best performing classifier with RPNet features.
- Notably, even classifiers that performed poorly initially showed considerable gains. Parallelepiped accuracy jumped to approx. 52.05% OA, and Minimum Distance improved to approx. 57.60% OA.
- GML (QDA) performance with RPNet features (approx. 40.87% OA) was comparable to its performance on raw data but lower than its performance with PCA features.

The consistent and significant improvements, especially for SVM and KNN, strongly highlight the value of incorporating spatial context, as captured by RPNet, for classifying the Indian Pines dataset.

6.5 Comparative Analysis and Conclusion

The comparative results, clearly visualized in the bar graph generated by the code (comparing OA across classifiers for Raw, PCA, and RPNet features) [1][2], lead to the following conclusions:

1. Feature extraction significantly influences classifier performance for the Indian Pines dataset. Relying solely on raw spectral data limits the accuracy achievable by several standard classifiers.

2. Simple dimensionality reduction via PCA (with n=4 components) provided mixed results, benefiting GML but hindering SVM and KNN in this specific configuration.

3. The spatial-spectral features generated by RPNet offered the most substantial and consistent improvements, dramatically boosting the performance of nearly all classifiers, especially SVM and KNN.

4. Among the classifiers tested, SVM consistently performed well, achieving the highest accuracy when combined with both raw features and, particularly, RPNet features (>97% OA). KNN also proved effective, especially with RPNet features (>91% OA).

In summary, the analysis indicates that while standard classifiers like GML, Minimum Distance, and Parallelepiped struggle with the high dimensionality and spectral similarity in the Indian Pines dataset when using raw data, their performance can be influenced by feature extraction. The addition of SVM provided a strong baseline. However, the most effective strategy identified was the combination of advanced spatial-spectral feature extraction using RPNet with a powerful classifier like SVM or KNN, demonstrating the critical role of spatial context in achieving high-accuracy hyperspectral image classification.

6.6 Visual Results

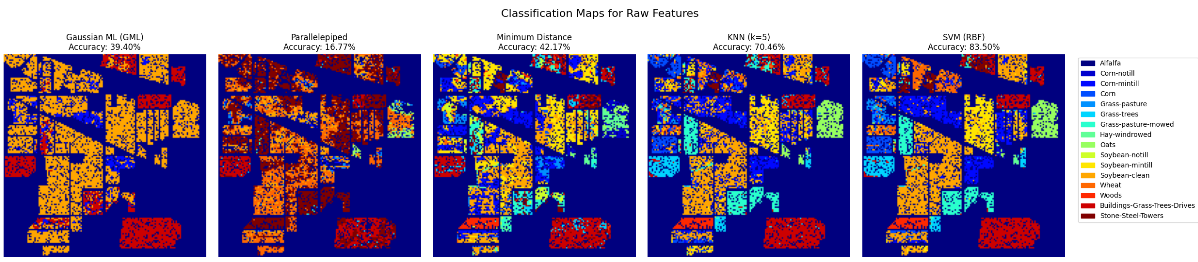

Figure 3: Classification using Raw Image

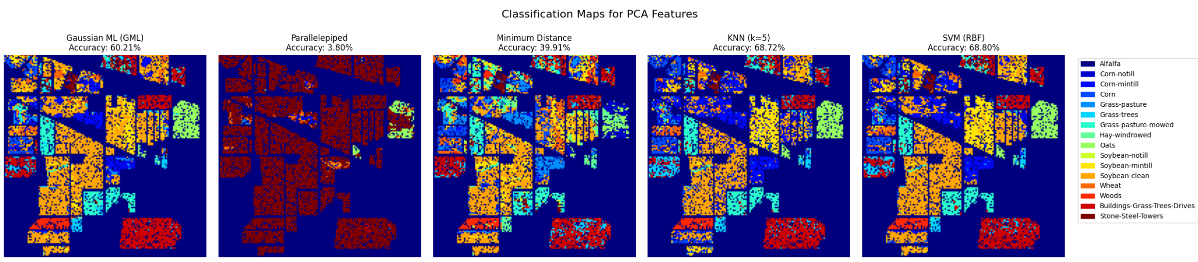

Figure 4: Classification using Principal Components

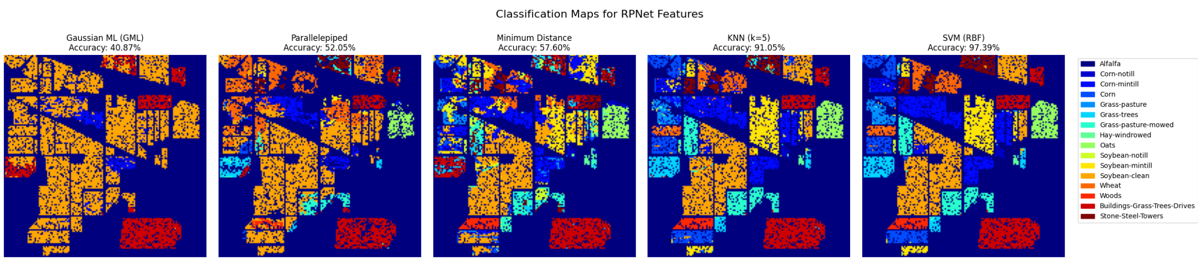

Figure 5: Classification using RPNet+Spectral Features

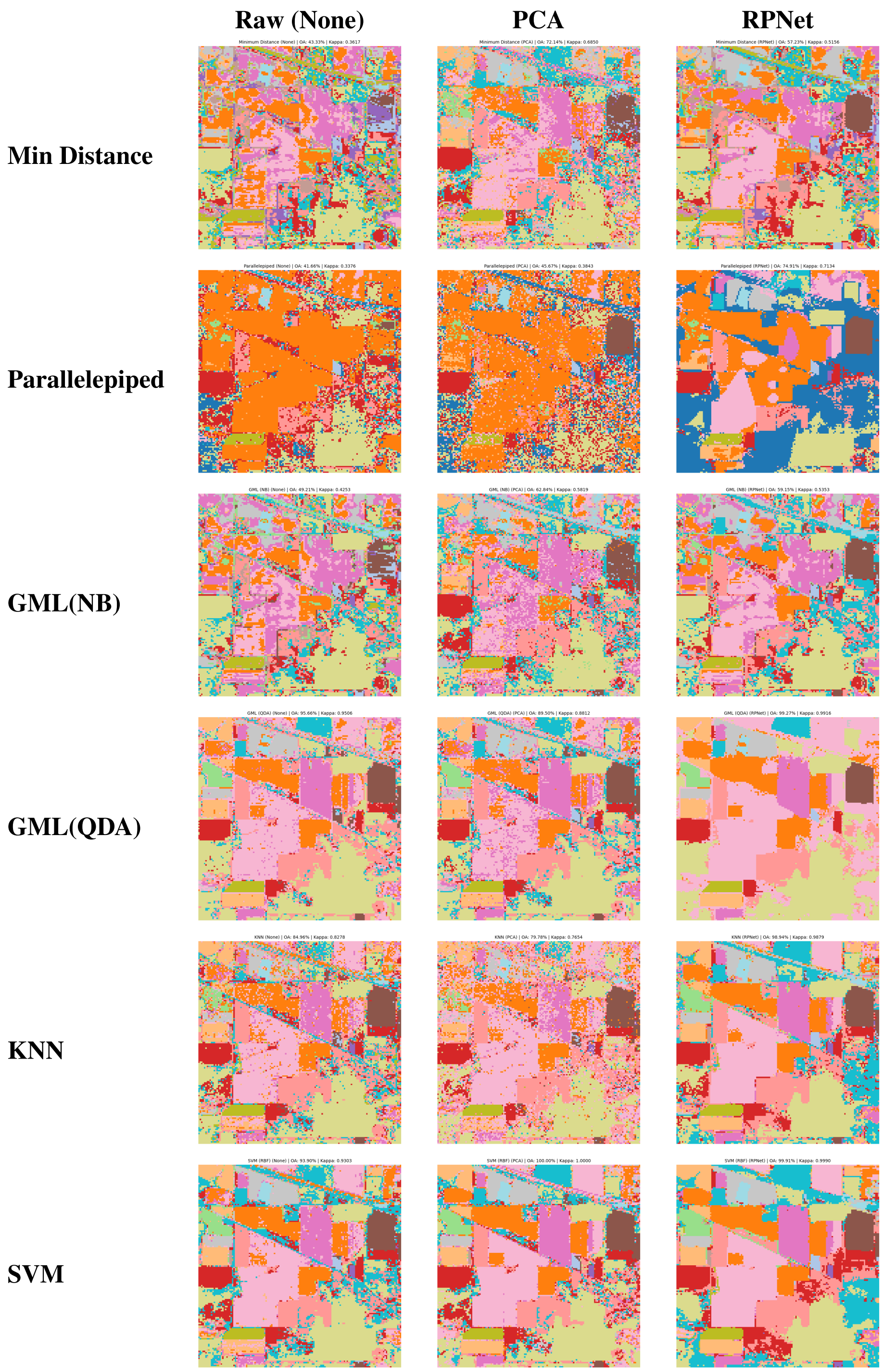

Figure 6: Collage of classification results from the Final Code (GUI): Each row corresponds to a classifier (top to bottom: Min Distance, Parallelepiped, GML(NB), GML(QDA), KNN, SVM), and each column to a feature set (None, PCA, RPNet). Each image is a square output map for the respective combination.

6.7 Accuracy Analysis

- It can be seen from the graph attached below that using **RPNet+Spectral** featurs give us the highest accuracy in nearly all cases.
- The only exception is in case of **Gaussian Maximum Likelihood** classifier where the highest accuracy is achieved for using **principal components** as the feature.
- It can be also seen that the **Parallelepiped** classifier is the lease accurate.

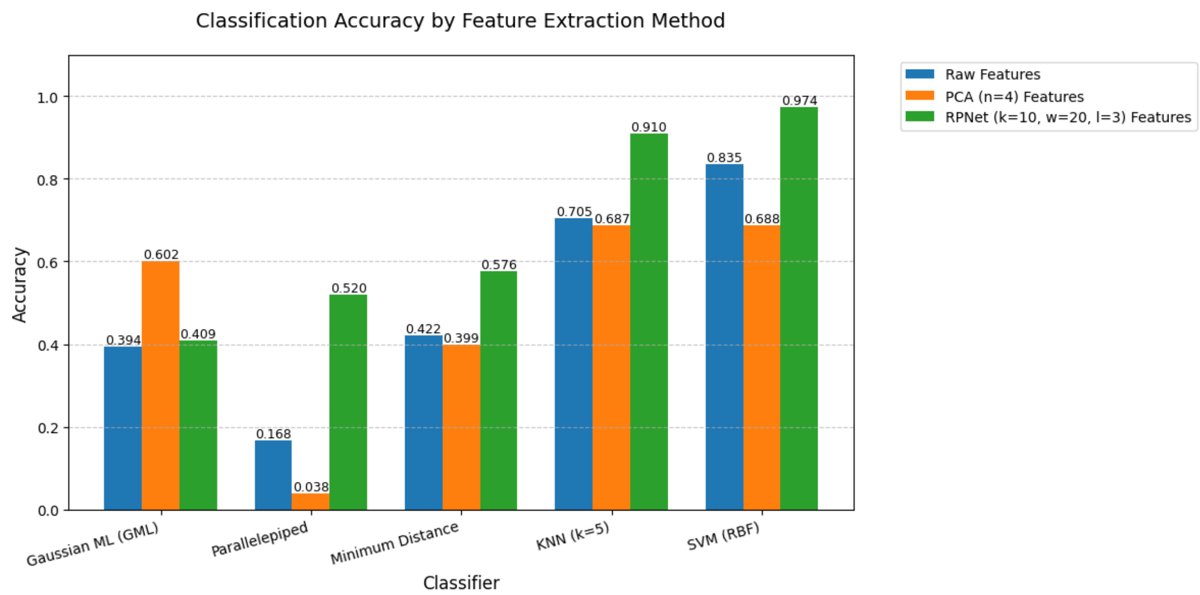

Figure 7: Overall Accuracy for Different Combinations

Table 1: Classification Performance Comparison (Accuracy %)
OA : Overall Accuracy, AA : Average Accuracy, κ : Kappa Coefficient

Metric	Raw - GML	Raw - Parallelepiped	Raw - Min Dist	Raw - KNN	Raw - SVM	PCA - GML	PCA - Parallelepiped	PCA - Min Dist	PCA - KNN	PCA - SVM	RPNet - GML	RPNet - Parallelepiped	RPNet - Min Dist	RPNet - KNN	RPNet - SVM
Class 1	0.00	0.00	67.57	5.41	62.16	37.84	0.00	62.16	2.70	0.00	0.00	2.70	83.78	89.19	94.59
Class 2	25.46	5.42	43.74	64.04	62.65	29.57	3.24	6.74	62.47	47.51	33.51	33.95	54.42	89.08	97.03
Class 3	0.00	0.30	15.96	54.22	61.05	30.72	0.00	20.93	50.60	41.72	0.00	6.33	26.81	45.60	45.88
Class 4	0.00	11.58	41.05	18.42	85.49	23.16	4.74	45.26	23.16	30.00	0.00	5.26	53.16	70.53	94.21
Class 5	0.00	2.07	9.33	84.20	98.12	38.34	0.00	9.33	66.84	55.44	0.00	50.26	15.03	91.71	94.82
Class 6	0.00	1.88	42.98	96.75	86.36	87.33	0.00	69.86	94.35	94.86	99.80	35.45	59.08	98.07	99.32
Class 7	0.00	0.00	100.00	18.18	97.91	50.00	0.00	95.45	50.00	72.73	0.00	0.00	100.00	95.45	95.45
Class 8	28.80	28.80	26.44	98.43	31.25	97.12	44.76	38.22	99.74	99.74	0.00	60.73	86.13	100.00	100.00
Class 9	0.00	0.00	93.75	18.75	76.35	0.00	0.00	68.75	0.00	0.00	0.00	0.00	100.00	37.50	100.00
Class 10	0.00	0.00	47.69	67.48	87.83	38.17	0.00	20.18	71.98	67.61	0.00	11.44	45.76	91.65	97.69
Class 11	98.27	18.53	46.18	74.75	74.53	81.06	0.10	61.25	72.45	85.90	99.80	86.20	67.72	92.67	96.87
Class 12	0.00	31.16	1.68	30.74	100.00	25.47	1.89	3.58	32.84	29.68	0.00	69.26	37.68	75.37	95.37
Class 13	0.00	0.00	96.95	100.00	97.53	96.34	0.00	93.90	98.78	99.39	0.00	45.73	100.00	100.00	100.00
Class 14	99.80	44.76	73.52	93.38	59.22	95.95	1.88	68.08	90.42	92.39	99.60	78.66	79.94	97.92	98.81
Class 15	0.00	47.90	22.98	21.36	90.54	29.77	0.00	12.62	19.09	25.89	0.00	64.08	38.51	80.56	94.91
Class 16	0.00	63.51	86.49	83.78	83.50	89.19	87.84	90.54	86.49	91.89	0.00	18.92	86.49	91.05	97.69
OA	39.40	16.77	42.17	70.46	78.20	60.21	3.80	39.91	68.72	68.80	40.87	52.05	57.60	86.51	97.39
AA	13.97	16.00	51.02	58.12	81.05	53.13	9.03	47.93	57.62	58.42	14.56	35.56	64.66	89.17	97.11
κ	23.66	10.92	34.85	66.02	81.05	53.50	2.60	32.74	64.12	63.73	24.65	44.49	51.91	89.79	97.03

6.8 Parameter Analysis

We also analyzed how the accuracy depends on:

- No. of Principal Components

  We fixed the classifier as **Support Vector Machine**, since we got the maximum accuracy in case of using PCs for SVM only, and increased the number of principal components from 1 to 10 and plotted the corresponding graph.

- It can be observed that the accuracy increase with the increaae in number of principal components but it tends to stabilise after some time.

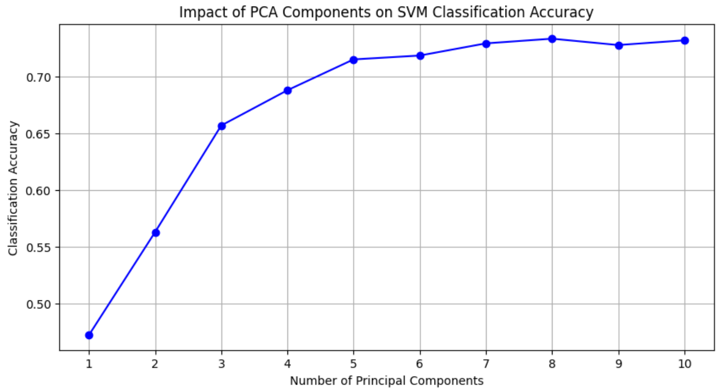

Figure 8: Variation of Accuracy with Number of Principal Components

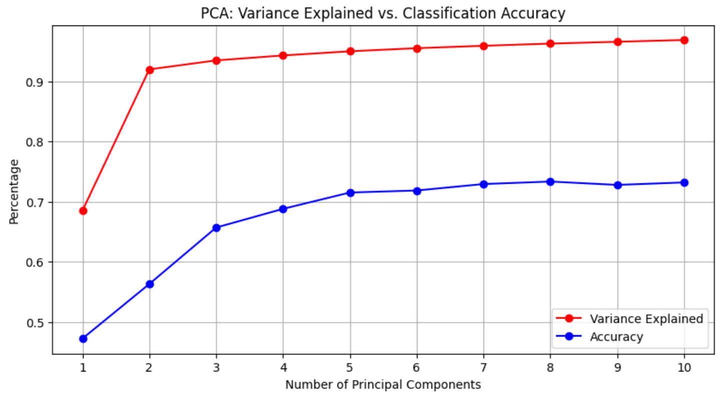

Figure 9: Variation of Accuracy with Number of Principal Components

6.9 Discussion

7 Scope of Future Work

Following the implementation and evaluation of the four core supervised classifiers, several avenues exist for extending this work:

- **Feature Selection:** Explore methods beyond general dimensionality reduction to specifically select the most informative spectral bands for discriminating between the land cover classes. Techniques based on statistical separability measures (e.g., Jeffries-Matusita distance, mentioned in the syllabus) could be implemented and tested.
- **Advanced Classification Algorithms:** Implement and compare the performance of the baseline classifiers against more advanced supervised learning algorithms. Support Vector Machines (SVM) and Artificial Neural Networks (ANN), both mentioned in the course syllabus, are widely used for hyperspectral classification due to their ability to handle complex, high-dimensional data.
- **Parameter Optimization:** Conduct systematic tuning of classifier parameters. For instance, employ cross-validation techniques to determine the optimal value of K for the K-Nearest Neighbors classifier or to optimize parameters for SVM (like kernel type, C, gamma) if implemented.
- **Incorporation of Spatial Information:** Enhance the classification by integrating spatial context. This could involve using texture features derived from the imagery, employing contextual classifiers that consider neighboring pixel labels, or exploring object-based image analysis (OBIA) approaches as suggested in the course content. Such methods often improve map smoothness and classification accuracy.

- **Refinement of Training Data:** Analyze the impact of the training data selection process. Investigate strategies for refining training samples or exploring techniques less sensitive to limited or potentially noisy training data.
- **Comparative Dataset Analysis:** Apply the implemented classification framework to other publicly available remote sensing datasets, such as the Salinas or Pavia University hyperspectral scenes, to evaluate the generalizability and robustness of the classifiers across different environments and sensor characteristics.

8 Conclusions

This project successfully implemented four supervised classification algorithms – Minimum Distance, Parallelepiped, Gaussian Maximum Likelihood, and K-Nearest Neighbors – for hyperspectral image classification using the Indian Pines dataset.

References

[1] CE672 Course Handout (2025). *Machine Processing of Remotely Sensed Data*. Department of Civil Engineering, IIT Kanpur.

[2] CE672 Lecture Notes (2025). *Digital Image Processing and Pattern Recognition*. Department of Civil Engineering, IIT Kanpur. Instructor: Onkar Dikshit.

[3] Mather P.M. and Koch M. (2011). *Computer Processing of Remotely-Sensed Images: An Introduction*, 4th ed., Wiley-Blackwell.

[4] Richards J.A. (2013). *Remote Sensing Digital Image Analysis*, 5th ed., Springer.

[5] Duda R.O., Hart P.E. and Stork D.G. (2001). *Pattern Classification*, 2nd ed., John Wiley & Sons.

[6] Jensen J.R. (2004). *Introductory Digital Image Processing: A Remote Sensing Perspective*, 3rd ed., Prentice-Hall.

[7] Gonzalez R.C. and Woods R.E. (2008). *Digital Image Processing*, 3rd ed., Pearson Prentice-Hall.

[8] Congalton R.G. and Green K. (2009). *Assessing the Accuracy of Remotely Sensed Data: Principles and Practices*, 2nd ed., CRC Press.

[9] Pratt W.K. (2001). *Digital Image Processing: PIKS Inside*, 3rd ed., John Wiley & Sons.

[10] Schowengerdt R.A. (2006). *Remote Sensing: Models and Methods for Image Processing*, 3rd ed., Academic Press.

[11] Baumgardner, M.F., Biehl, L.L., & Landgrebe, D.A. (2015). 220 Band AVIRIS Hyperspectral Image Data Set: June 12, 1992 Indian Pine Test Site 3. Purdue University Research Repository.

[12] Zhao, C., Wan, X., Zhang, Y., & Li, J. (2018). *Classification of hyperspectral imagery using a new fully convolutional neural network*. Journal of Applied Geophysics, 159, 158-167. doi:`10.1016/j.jag.2018.08.014`. Available: `https://www.sciencedirect.com/science/article/pii/S0924271618301473`

[13] University of the Basque Country. (2025). *Hyperspectral Remote Sensing Scenes: Indian Pines Dataset*. Retrieved April 14, 2025, from `https://www.ehu.eus/ccwintco/index.php/Hyperspectral_Remote_Sensing_Scenes#Indian_Pines`

[14] Y. Xu, Q. Du, and L. Zhang, *Hyperspectral image classification via a random patches network*, ISPRS Journal of Photogrammetry and Remote Sensing, vol. 142, pp. 344–357, 2018.

[15] Colwell, R.N. (Ed.). (1983). *Manual of Remote Sensing*, 2nd ed., American Society of Photogrammetry.

[16] Sabins, F.F. (2007). *Remote Sensing: Principles and Interpretation*, 3rd ed., Waveland Press.

[17] Pellemans, A.H.J.M. et al. (1993). "Meris Potential for Land Applications". *International Journal of Remote Sensing*, 14(9), 1797-1812.

[18] Theodoridis, S. & Koutroumbas, K. (2008). *Pattern Recognition*, 4th ed., Academic Press.

[19] Tou, J.T. & Gonzalez, R.C. (1974). *Pattern Recognition Principles*, Addison-Wesley.

A Python Code Implementation

```
##### *Start by importing all neccessary Libraries*
# Importing all necessary Libraries
import tkinter as tk
from tkinter import filedialog, messagebox
from tkinter import ttk
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
from collections import Counter
import contextlib # Added for dummy torch context manager

# --- Scikit-learn Imports ---
from sklearn.metrics import confusion_matrix, accuracy_score, cohen_kappa_score
from sklearn.neighbors import KNeighborsClassifier, NearestCentroid
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.decomposition import PCA
# importing RPNet Specific Imports with exceptions handling
# --- RPNet Specific Imports (Optional Dependency) ---
try:
    import torch
    import torch.nn as nn
    import torch.nn.functional as F
    TORCH_AVAILABLE = True
except ImportError:
    TORCH_AVAILABLE = False
    # Define dummy classes/functions if torch not found
    class nn: Module = object; Parameter = object
    class F:
        @staticmethod
        def conv2d(*args, **kwargs): raise ImportError("PyTorch not found. RPNet
            requires PyTorch installation.")
    class torch:
        float32 = None
        no_grad = contextlib.contextmanager(lambda: (yield))
        @staticmethod
        def tensor(*args, **kwargs): raise ImportError("PyTorch not found. RPNet
            requires PyTorch installation.")
        @staticmethod
        def cat(*args, **kwargs): raise ImportError("PyTorch not found. RPNet
            requires PyTorch installation.")
        @staticmethod
        def from_numpy(*args, **kwargs): raise ImportError("PyTorch not found.
            RPNet requires PyTorch installation.")
        @staticmethod
        def unsqueeze(*args, **kwargs): raise ImportError("PyTorch not found. RPNet
             requires PyTorch installation.")
        @staticmethod
        def squeeze(*args, **kwargs): raise ImportError("PyTorch not found. RPNet
            requires PyTorch installation.")

##### *Main*
# creating Global Variables for GUI
# --- Global Variables ---
corrected_img = None
ground_truth = None


```

```python
# Creating RPNet Functions for Feature Extraction
# --- RPNet Functions ---

# Creating apply_pca_and_whiten for PCA, takes
def apply_pca_and_whiten(X, p):
    '''
    Input Parameters:
        X: feature map (R, C, N)
        p: number of PCs to keep
    Output Parameters:
        X_white: whitened feature map (R, C, p) or (R, C, 0)
        pca: PCA object fitted on the data or None
    '''
    if X.ndim != 3:
        raise ValueError(f"Input X must be 3D (R, C, N), but got shape {X.shape}")
    R, C, N = X.shape

    if p <= 0: # Handle case where 0 components are requested explicitly or
       implicitly
        print(f"Warning: PCA requested with p={p}. Returning zero features.")
        return np.zeros((R, C, 0)), None # Return empty features and None for pca
            object

    if N < p:
        print(f"Warning: Requested {p} PCA components, but input only has {N}
            features. Using n_components={N}.")
        p = N # Adjust p to the maximum possible

    if N == 0: # Handle case where input has no features
        print(f"Warning: Input to PCA has 0 features. Returning zero features.")
        return np.zeros((R, C, 0)), None

    reshaped = X.reshape(-1, N)

    try:
        pca = PCA(n_components=p)
        X_pca = pca.fit_transform(reshaped)

        # Check if PCA actually produced components (it might not if variance is
            zero)
        if X_pca.shape[1] == 0:
             print(f"Warning: PCA resulted in 0 components for p={p} (input shape {
                 N}). Returning zero features.")
             return np.zeros((R, C, 0)), pca # Return empty features

        # Whiten (Standardize PCA components)
        scaler = StandardScaler()
        X_white_flat = scaler.fit_transform(X_pca)
        # Ensure output shape matches requested 'p' (or adjusted 'p')
        return X_white_flat.reshape(R, C, X_pca.shape[1]), pca
    except ValueError as e:
         # Catch errors during fit (e.g., all zero variance input)
         print(f"Error during PCA/Whitening (p={p}, N={N}): {e}. Returning zero
             features.")
         return np.zeros((R, C, 0)), None # Return empty on error


# extract_random_patches_as_filters (minor robustness)
def extract_random_patches_as_filters(X_white, patch_size, k):
    ''' Extracts k random patches from X_white to be used as Conv filters. '''
    if not TORCH_AVAILABLE:
         raise ImportError("PyTorch not found. RPNet requires PyTorch installation.
             ")
```

```
    if X_white.ndim != 3:
        raise ValueError(f"Input X_white must be 3D (R, C, P), but got shape {X_white.shape}")

    R, C, P = X_white.shape
    if P == 0:
        raise ValueError("Input to extract_random_patches_as_filters has 0 features (P=0). Cannot extract filters.")

    pad = patch_size // 2
    # Use 'reflect' padding
    padded = np.pad(X_white, ((pad, pad), (pad, pad), (0, 0)), mode='reflect')
    patches = []
    attempts = 0
    max_attempts = k * 5 # Try a bit harder to find patches

    while len(patches) < k and attempts < max_attempts:
        attempts += 1
        # Ensure indices are within valid range for the *original* dimensions R, C
        # We sample center points from the original image grid
        i_center = np.random.randint(0, R)
        j_center = np.random.randint(0, C)
        # Calculate slice indices in the *padded* array
        i_start, i_end = i_center, i_center + patch_size
        j_start, j_end = j_center, j_center + patch_size
        patch = padded[i_start:i_end, j_start:j_end, :]

        if patch.shape == (patch_size, patch_size, P):
            # Transpose to (P, H, W) - Channels first for PyTorch Conv2d filters
            patches.append(np.transpose(patch, (2, 0, 1)))
        else:
             # This case should be rare with correct padding and indexing
             print(f"Warning: Extracted patch shape mismatch. Expected {(patch_size, patch_size, P)}, got {patch.shape}. Indices i_center={i_center}, j_center={j_center}")

    if len(patches) < k:
         print(f"Warning: Could only extract {len(patches)} out of {k} desired patches.")
    if not patches:
        raise ValueError(f"Could not extract any valid patches after {max_attempts} attempts. Check patch_size ({patch_size}), input dimensions ({R},{C},{P}), and padding.")

    # Stack patches to form the filter bank: (num_extracted_patches, P, H, W)
    filter_bank = np.stack(patches)
    return torch.tensor(filter_bank, dtype=torch.float32)


class RPNetFixedLayer(nn.Module):
    def __init__(self, filters):
        super(RPNetFixedLayer, self).__init__()
        if not TORCH_AVAILABLE:
             raise ImportError("PyTorch not found. RPNet requires PyTorch installation.")
        self.filters = nn.Parameter(filters, requires_grad=False)

    def forward(self, x):
        if not isinstance(x, torch.Tensor):
            x = torch.tensor(x, dtype=torch.float32)
        elif x.dtype != torch.float32:
            x = x.float()
        # filters shape: (C_out=k, C_in=P, kH, kW)
```

```python
        return F.conv2d(x, self.filters, padding='same')

# run_rpnet_layers
def run_rpnet_layers(X_input, patch_size, k, L, initial_pca_components=4):
    """ Runs RPNet. Returns (R, C, Features) or (R, C, 0) if fails. """
    if not TORCH_AVAILABLE:
        raise ImportError("PyTorch not found. RPNet requires PyTorch installation.")

    print(f"Running RPNet: patch_size={patch_size}, k={k}, L={L}, initial_pca={initial_pca_components}")
    R_orig, C_orig, N_orig = X_input.shape

    # follow these steps to for running the RPNet:
    # 1. Initial PCA and Whitening
    print("  - Applying initial PCA...")
    current_feature_map, _ = apply_pca_and_whiten(X_input, initial_pca_components)
    if current_feature_map.shape[-1] == 0:
         print("  - Error: Initial PCA resulted in 0 features. Aborting RPNet.")
         return np.zeros((R_orig, C_orig, 0))                        # Return empty features
    print(f"  - Initial PCA output shape: {current_feature_map.shape}")

    feature_stack = []

    for layer_idx in range(L):
        print(f"  - Processing RPNet Layer {layer_idx + 1}/{L}...")
        current_R, current_C, current_P = current_feature_map.shape

        # 2. Intermediate PCA (if not first layer)
        if layer_idx > 0:
             print(f"    - Applying intermediate PCA (target 4 components)...")
             intermediate_pca_target = 4
             current_feature_map, _ = apply_pca_and_whiten(current_feature_map, intermediate_pca_target)
             # Check if intermediate PCA failed
             if current_feature_map.shape[-1] == 0:
                 print(f"    - Error: Intermediate PCA resulted in 0 features at layer {layer_idx + 1}. Stopping RPNet processing.")
                 break # Stop adding layers if features vanish
             print(f"    - Intermediate PCA output shape: {current_feature_map.shape}")
             # Update P for filter extraction
             current_P = current_feature_map.shape[-1]

        # 3. Extract Filters
        print(f"    - Extracting {k} random patches (size {patch_size}x{patch_size})...")
        try:
            # Check if there are enough features to extract patches from
            if current_P == 0:
                print("    - Error: Cannot extract patches with 0 input features. Stopping layer processing.")
                break
            filters = extract_random_patches_as_filters(current_feature_map, patch_size, k)
            # Update k if fewer filters were extracted
            actual_k = filters.shape[0]
            print(f"    - Actual filters extracted: {actual_k}. Shape: {filters.shape}")
        except ValueError as e:
            print(f"    - Error extracting patches: {e}. Stopping RPNet processing.")
```

```
            break # Stop if filters cannot be created

        # 4. Prepare Input Tensor for Convolution
        #    Input needs shape (1, P, R, C)
        inp_tensor = torch.tensor(current_feature_map.transpose(2, 0, 1)).unsqueeze(0)
        print(f"    - Input tensor shape for Conv2D: {inp_tensor.shape}")

        # 5. Define and Apply Fixed Convolution Layer
        model = RPNetFixedLayer(filters)
        with torch.no_grad():
            output_tensor = model(inp_tensor)                # Output shape (1, actual_k, R, C)

        # 6. Process Output
        output_map = output_tensor.squeeze(0).numpy().transpose(1, 2, 0) # Shape (R, C, actual_k)
        print(f"    - Output map shape for layer {layer_idx + 1}: {output_map.shape}")

        feature_stack.append(output_map)
        current_feature_map = output_map                    # Output of this layer is input for next

    # 7. Concatenate features
    if not feature_stack:
         print("Warning: RPNet generated no feature maps in any layer.")
         return np.zeros((R_orig, C_orig, 0)) # Return shape (R, C, 0)

    try:
        final_features = np.concatenate(feature_stack, axis=-1)
        print(f"  - Final RPNet features concatenated. Shape: {final_features.shape}")
        # Ensure final shape matches original spatial dimensions
        if final_features.shape[:2] != (R_orig, C_orig):
             print(f"Warning: Final RPNet feature spatial dimensions {final_features.shape[:2]} don't match original {R_orig, C_orig}. This shouldn't happen with 'same' padding.")
             # Attempt to resize? Or return empty? For now, return empty.
             return np.zeros((R_orig, C_orig, 0))
        return final_features
    except ValueError as e:
         print(f"Error concatenating RPNet features: {e}. Feature stack shapes: {[f.shape for f in feature_stack]}")
         return np.zeros((R_orig, C_orig, 0))

# compile all the featurrses together in  using combined_features function
def combined_features(spatial_feat, spectral_data):
    """ Combines RPNet spatial features and original spectral data. Handles empty spatial_feat. """
    print("Combining features...")
    R, C, N = spectral_data.shape

    # --- Handle Spectral Data ---
    spectral_flat = spectral_data.reshape(-1, N)
    scaler_spectral = StandardScaler()
    spectral_scaled_flat = scaler_spectral.fit_transform(spectral_flat)
    print(f"  - Scaled spectral features shape: {spectral_scaled_flat.shape}")

    # --- Handle Spatial Data ---
    if spatial_feat is None or spatial_feat.size == 0 or spatial_feat.shape[-1] == 0:
```

```
        print("  - No valid spatial features provided. Using only spectral features.")
        # No spatial features, return only scaled spectral (already flat)
        # We still need to scale them together IF spatial existed, but here only spectral exists.
        # So, scale spectral alone.
        scaler_combined = StandardScaler()
        combined_scaled_flat = scaler_combined.fit_transform(spectral_scaled_flat)
        print(f"  - Final combined scaled flat shape (spectral only): {combined_scaled_flat.shape}")
        return combined_scaled_flat
    else:
        # Spatial features exist
        S = spatial_feat.shape[-1]                                  # Number of spatial features
        print(f"  - Processing {S} spatial features.")
        spatial_flat = spatial_feat.reshape(-1, S)
        scaler_spatial = StandardScaler()
        spatial_scaled_flat = scaler_spatial.fit_transform(spatial_flat)
        print(f"  - Scaled spatial features shape: {spatial_scaled_flat.shape}")

        # --- Combine Scaled Features ---
        combined_flat = np.concatenate([spectral_scaled_flat, spatial_scaled_flat], axis=-1)
        print(f"  - Combined flat shape before final scaling: {combined_flat.shape}")

        # Scale the combined features together
        scaler_combined = StandardScaler()
        combined_scaled_flat = scaler_combined.fit_transform(combined_flat)
        print(f"  - Final combined scaled flat shape: {combined_scaled_flat.shape}")
        return combined_scaled_flat
#### *Create the Final GUI functions and GUI*

# --- GUI Functions ---
# function to laod the corrected or Raw file
def load_corrected_file():
    global corrected_img
    path = filedialog.askopenfilename(title="Select Corrected Image (.mat)", filetypes=[("MAT files", "*.mat")])
    if path:
        try:
            data = scipy.io.loadmat(path)
            potential_keys = [k for k, v in data.items() if isinstance(v, np.ndarray) and v.ndim == 3]
            if not potential_keys:
                raise ValueError("No 3D numpy array found in corrected MAT file.")
            img_key = max(potential_keys, key=lambda k: data[k].size)
            corrected_img = data[img_key].astype(np.float32)            # Ensure float32
            status_label.config(text=f"    Corrected image loaded (key: '{img_key}')", fg='green')
            status_label2.config(text=f"Shape: {corrected_img.shape}, Bands: {corrected_img.shape[2]}, Type: {corrected_img.dtype}")
            print(f"Loaded corrected image: shape={corrected_img.shape}, dtype={corrected_img.dtype}")
        except Exception as e:
            corrected_img = None
            status_label.config(text=f"    Error loading corrected image: {e}", fg='red')
            status_label2.config(text="")
```

```
            messagebox.showerror("Load Error", f"Failed to load corrected image:\n{e}")

# function to load the Ground Truth File
def load_gt_file():
    global ground_truth
    path = filedialog.askopenfilename(title="Select Ground Truth (.mat)", filetypes=[("MAT files", "*.mat")])
    if path:
        try:
            data = scipy.io.loadmat(path)
            potential_keys = [k for k, v in data.items() if isinstance(v, np.ndarray) and v.ndim == 2]
            if not potential_keys:
                raise ValueError("No 2D numpy array found in ground truth MAT file.")
            gt_key = max(potential_keys, key=lambda k: data[k].size)
            ground_truth = data[gt_key]
            ground_truth = ground_truth.astype(int)                     # Ensure GT is integer type
            status_label.config(text=f"    Ground truth loaded (key: '{gt_key}')", fg='green')
            status_label2.config(text=f"Shape: {ground_truth.shape}, Classes: {len(np.unique(ground_truth))}")
        except Exception as e:
            ground_truth = None
            status_label.config(text=f"    Error loading ground truth: {e}", fg='red')
            status_label2.config(text="")
            messagebox.showerror("Load Error", f"Failed to load ground truth:\n{e}")

# Function for enabling PCA
def toggle_pca_options(*args):
    """Enable/disable PCA components entry based on feature extraction choice."""
    if feature_extraction_method.get() == "PCA":
        pca_label.config(state=tk.NORMAL)
        pca_entry.config(state=tk.NORMAL)
    else:
        pca_label.config(state=tk.DISABLED)
        pca_entry.config(state=tk.DISABLED)

# --- classify_and_display Function ---
def classify_and_display():
    # (Initial checks for loaded data and matching dimensions are unchanged) ---> they will be highlighted in the status bar of our GUI
    if corrected_img is None or ground_truth is None:
        messagebox.showerror("Error", "Please load both corrected image and ground truth files.")
        status_label.config(text="    Load both corrected and ground truth files first!", fg='red')
        return
    if corrected_img.shape[:2] != ground_truth.shape:
         messagebox.showerror("Error", f"Image dimensions {corrected_img.shape[:2]} do not match Ground Truth dimensions {ground_truth.shape}.")
         status_label.config(text="    Image and Ground Truth dimensions do not match!", fg='red')
         status_label2.config(text=f"Image: {corrected_img.shape[:2]}, GT: {ground_truth.shape}")
         return

    feature_method = feature_extraction_method.get()
    clf_name = classifier_type.get()
```

```
    status_label.config(text=f"    Preparing data (Feature Method: {feature_method})...", fg='blue')
    status_label2.config(text="")
    gui_frame.update_idletasks()

    try:
        h, w, b = corrected_img.shape                # height, width , bands
        gt_flat = ground_truth.ravel()               # sinle band image

        # --- Feature Extraction ---
        X_extracted_flat = None                      # Will hold the features for all pixels (h*w, num_features)
        num_features = 0

        # Apply method for feature extraction
        # None ---> takes raw image and classifies
        if feature_method == "None":
            status_label2.config(text="Using raw spectral features.")
            gui_frame.update_idletasks()
            X_extracted_flat = corrected_img.reshape(-1, b)

        # PCA -----> for dimessionality reduction (faster)
        elif feature_method == "PCA":
            n_comp = pca_components.get()
            if n_comp <= 0 or n_comp > b:
                messagebox.showerror("Error", f"Number of PCA components must be between 1 and {b}.")
                status_label.config(text=f"    Invalid PCA components (1-{b})", fg='red')
                return
            status_label.config(text=f"    Applying PCA ({n_comp} components)...", fg='blue')
            gui_frame.update_idletasks()
            # Use apply_pca_and_whiten to get scaled PCA features directly
            # We need the flat version, so reshape input and apply
            pca_features_scaled_flat, pca_obj = apply_pca_and_whiten(corrected_img, n_comp)
            # Reshape the output of apply_pca_and_whiten (which is R,C,P) back to flat
            if pca_features_scaled_flat.shape[-1] == 0:
                messagebox.showerror("Error", f"PCA resulted in 0 components.")
                status_label.config(text=f"    PCA failed to produce components.", fg='red')
                return
            X_extracted_flat = pca_features_scaled_flat.reshape(-1, pca_features_scaled_flat.shape[-1])
            explained_variance = np.sum(pca_obj.explained_variance_ratio_) * 100 if pca_obj else 0
            status_label2.config(text=f"PCA Applied ({X_extracted_flat.shape[1]} comps). Var: {explained_variance:.2f}%")
            print(f"PCA completed. Output shape: {X_extracted_flat.shape}")

        # RPNet ------> More Robust for Feature extraction in classification problems (does that by using patch selection)
        elif feature_method == "RPNet":
            if not TORCH_AVAILABLE:
                messagebox.showerror("Dependency Error", "RPNet requires PyTorch.\nPlease install it (e.g., `pip install torch`).")
                status_label.config(text="    RPNet requires PyTorch. Please install it.", fg='red')
                return

```

```
            status_label.config(text="    Running RPNet Feature Extraction (can take time)...", fg='blue')
            gui_frame.update_idletasks()

            rp_patch_size = 20; rp_k = 10; rp_L = 3; rp_initial_pca = 4

            spatial_features_map = run_rpnet_layers(corrected_img, rp_patch_size, rp_k, rp_L, rp_initial_pca) # Shape (R, C, k*L or 0)

            # Check if RPNet produced any features before combining
            if spatial_features_map.shape[-1] == 0:
                status_label2.config(text="RPNet failed to generate spatial features. Using spectral only.")
                print("RPNet returned 0 features. Proceeding with spectral only for combination.")
                # combined_features will handle the zero-feature case now
            else:
                status_label2.config(text="RPNet spatial features generated. Combining with spectral...")

            gui_frame.update_idletasks()
            X_extracted_flat = combined_features(spatial_features_map, corrected_img)          # Shape (R*C, CombinedFeatures)

            status_label2.config(text=f"Feature combination complete. Total features: {X_extracted_flat.shape[1]}")
            print(f"RPNet+Combined completed. Output shape: {X_extracted_flat.shape}")


        else:
            messagebox.showerror("Error", "Unknown feature extraction method selected.")
            status_label.config(text="    Unknown feature extraction method selected.", fg='red')
            return

        # --- Check if features were actually extracted ---
        if X_extracted_flat is None or X_extracted_flat.size == 0:
            messagebox.showerror("Error", f"Feature extraction ({feature_method}) failed to produce any features.")
            status_label.config(text=f"    Feature extraction ({feature_method}) produced no data.", fg='red')
            return
        num_features = X_extracted_flat.shape[1]
        print(f"Features extracted successfully: shape={X_extracted_flat.shape}")


        # --- Prepare Training/Prediction Data ---
        status_label.config(text="    Preparing training data...", fg='blue')
        gui_frame.update_idletasks()

        if remove_bg.get(): train_mask = gt_flat > 0
        else: train_mask = gt_flat >= 0

        X_train_raw = X_extracted_flat[train_mask]
        y_train = gt_flat[train_mask]

        if X_train_raw.shape[0] == 0:
            messagebox.showerror("Error", "No training samples found. Check ground truth or 'Exclude background' option.")
            status_label.config(text="    No training samples found with the current background setting.", fg='red')
```

```
             return

        status_label2.config(text=f"Training samples: {X_train_raw.shape[0]}, Features: {num_features}")
        gui_frame.update_idletasks()

        # --- Scaling ---
        # NOTE: The extracted features from PCA/RPNet might already be scaled.
        # Re-scaling here ensures consistency, fitting on train and applying to all.
        status_label.config(text="    Scaling features (fit on train)...", fg='blue')
        gui_frame.update_idletasks()
        scaler = StandardScaler()
        X_train_scaled = scaler.fit_transform(X_train_raw)
        X_predict_scaled = scaler.transform(X_extracted_flat) # Use same scaler for prediction data

        # now apply for classification
        # --- Classification ---
        # (Classification logic remains the same, using X_train_scaled and X_predict_scaled)
        status_label.config(text=f"    Training {clf_name}...", fg='blue')
        gui_frame.update_idletasks()

        clf = None
        preds_flat = None                                      # Predictions for the entire image (flat)

        unique_classes_train = np.unique(y_train)

        # Classifier fitting and prediction logic
        # for Minimum distance to means Classifier -----> Distance metric is Euclidean Distance
        if clf_name == 'Minimum Distance':
            clf = NearestCentroid(metric='euclidean')
            clf.fit(X_train_scaled, y_train)
            preds_flat = clf.predict(X_predict_scaled)

        # For Parallelepiped classifier (created mmanually)
        elif clf_name == 'Parallelepiped':
             class_bounds = {}
             preds_flat = np.zeros(X_predict_scaled.shape[0], dtype=int)
             for c in unique_classes_train:
                 if c == 0 and remove_bg.get(): continue
                 X_c = X_train_scaled[y_train == c]
                 if X_c.shape[0] > 0:
                     min_vals = np.min(X_c, axis=0); max_vals = np.max(X_c, axis=0)
                     class_bounds[c] = {'min': min_vals, 'max': max_vals}
             for c in unique_classes_train:
                  if c in class_bounds:
                     is_within = np.all((X_predict_scaled >= class_bounds[c]['min']) & (X_predict_scaled <= class_bounds[c]['max']), axis=1)
                     preds_flat[is_within & (preds_flat == 0)] = c

        # other classifiers created directly form skleaarn
        # GML(NB) ---> Gaussina Maximum Likelihood Classifier using Naive Bayes (Assumes all classes to follow Gaussian Distribution)
        elif clf_name == 'GML (NB)':
            clf = GaussianNB()
            clf.fit(X_train_scaled, y_train)
            preds_flat = clf.predict(X_predict_scaled)

```

```python
        # GML(QDA) ---> Gaussina Maximum Likelihood Classifier using Quadratic Discrimiantory Analysis (DOES NOT Assumes all classes to follow Gaussian Distribution) [Robust]
        elif clf_name == 'GML (QDA)':
            try:
                clf = QuadraticDiscriminantAnalysis()
                clf.fit(X_train_scaled, y_train)
                preds_flat = clf.predict(X_predict_scaled)
            except Exception as qda_error:
                messagebox.showerror("QDA Error", f"QDA failed. This can happen if a class has too few samples for the number of features.\nError: {qda_error}")
                status_label.config(text=f"    QDA Error: {qda_error}", fg='red')
                return

        # K Nearest Neighbour Classifier (direct from Sklearn)
        elif clf_name == 'KNN':
            k = k_value.get()
            if k <= 0:
                messagebox.showerror("Error", "K value for KNN must be positive.")
                status_label.config(text="    K value must be positive.", fg='red')
                return
            clf = KNeighborsClassifier(n_neighbors=k)
            clf.fit(X_train_scaled, y_train)
            preds_flat = clf.predict(X_predict_scaled)

        # Suppoort Vector Machines Classifier (direct from Sklearn)
        elif clf_name == 'SVM (RBF)':
            clf = SVC(kernel='rbf', C=10, gamma='scale', probability=False)
            clf.fit(X_train_scaled, y_train)
            preds_flat = clf.predict(X_predict_scaled)

        # check for predictions
        if preds_flat is None:
            messagebox.showerror("Error", "Classification step failed to produce predictions.")
            status_label.config(text="    Classification failed.", fg='red')
            return

        classified_map = preds_flat.reshape(h, w)

        # --- Evaluation (on training pixels) ---
        status_label.config(text="    Evaluating...", fg='blue')
        gui_frame.update_idletasks()
        y_pred_train = preds_flat[train_mask]
        labels_present = np.unique(np.concatenate((y_train, y_pred_train)))
        cm = confusion_matrix(y_train, y_pred_train, labels=labels_present)
        oa = accuracy_score(y_train, y_pred_train) * 100
        kappa = cohen_kappa_score(y_train, y_pred_train)
        status_label.config(text="    Classification Complete!", fg='green')
        status_label2.config(text=f"OA: {oa:.2f}%, Kappa: {kappa:.4f} (evaluated on training pixels)")


        # --- Plotting --- (plot the result with accuracy and type of classifier)
        fig, ax = plt.subplots(figsize=(7, 7))
        max_class_val = np.max(ground_truth) if ground_truth is not None else 1
        cmap = plt.cm.get_cmap('tab20', max_class_val + 1)
        im = ax.imshow(classified_map, cmap=cmap, vmin=0, vmax=max_class_val)
        ax.set_title(f"{clf_name} ({feature_method}) | OA: {oa:.2f}% | Kappa: {kappa:.4f}", fontsize=10)
        ax.axis('off')
        plt.tight_layout()
```

```python
        plt.show()

    # --- Exception Handling  ---
    except ImportError as imp_err:
         messagebox.showerror("Import Error", f"{imp_err}\nMake sure required libraries (like PyTorch for RPNet) are installed.")
         status_label.config(text=f"    Import Error: {imp_err}", fg='red')
    except Exception as e:
        messagebox.showerror("Processing Error", f"An error occurred during {status_label.cget('text')}:\n{e}")
        status_label.config(text=f"    Error during processing: {e}", fg='red')
        status_label2.config(text="Check console for detailed traceback.")
        import traceback
        print("\n--- Error Traceback ---")
        traceback.print_exc()
        print("---------------------\n")


# --- GUI Setup  ---
# Main (root) GUI
gui_frame = tk.Tk()
gui_frame.geometry("500x650")
gui_frame.resizable(False, False)
gui_frame.title("Hyperspectral Classification GUI V2.1 (RPNet Fix)")

# --- Variables ---
feature_extraction_method = tk.StringVar(value="None")
classifier_type = tk.StringVar(value="Minimum Distance")
k_value = tk.IntVar(value=5)
pca_components = tk.IntVar(value=4)
remove_bg = tk.BooleanVar(value=True)

# --- Frames ---
load_frame = ttk.LabelFrame(gui_frame, text="1. Load Data")
load_frame.pack(pady=10, padx=10, fill='x')
feature_frame = ttk.LabelFrame(gui_frame, text="2. Feature Extraction Method")
feature_frame.pack(pady=5, padx=10, fill='x')
classify_frame = ttk.LabelFrame(gui_frame, text="3. Classification Method")
classify_frame.pack(pady=5, padx=10, fill='x')
options_frame = ttk.LabelFrame(gui_frame, text="4. Additional Options")
options_frame.pack(pady=5, padx=10, fill='x')
run_frame = tk.Frame(gui_frame)
run_frame.pack(pady=10, padx=10, fill='x')
status_frame = ttk.LabelFrame(gui_frame, text="Status")
status_frame.pack(pady=5, padx=10, fill='x', expand=True)

# --- Widgets ---
# Load Frame
tk.Button(load_frame, text="Load Corrected Image (.mat)", command=load_corrected_file).pack(pady=5, padx=10, fill='x')
tk.Button(load_frame, text="Load Ground Truth (.mat)", command=load_gt_file).pack(pady=5, padx=10, fill='x')

# Feature Frame
tk.Label(feature_frame, text="Select Method:").grid(row=0, column=0, padx=5, pady=5, sticky='w')
feature_dropdown = ttk.Combobox(feature_frame, textvariable=feature_extraction_method, values=["None", "PCA", "RPNet"], state="readonly", width=15)
feature_dropdown.grid(row=0, column=1, padx=5, pady=5, sticky='w')
feature_dropdown.bind("<<ComboboxSelected>>", toggle_pca_options)
pca_label = tk.Label(feature_frame, text="PCA Components:")
pca_label.grid(row=1, column=0, padx=5, pady=2, sticky='e')
```

```
pca_entry = tk.Entry(feature_frame, textvariable=pca_components, width=5)
pca_entry.grid(row=1, column=1, padx=5, pady=2, sticky='w')
toggle_pca_options() # Initial state

# Classification Frame
tk.Label(classify_frame, text="Select Classifier:").grid(row=0, column=0, padx=5,
    pady=5, sticky='w')
classifier_dropdown = ttk.Combobox(classify_frame, textvariable=classifier_type,
    values=['Minimum Distance', 'Parallelepiped', 'GML (NB)', 'GML (QDA)', 'KNN', '
    SVM (RBF)'], state="readonly", width=25)
classifier_dropdown.grid(row=0, column=1, columnspan=2, padx=5, pady=5, sticky='w')
knn_label = tk.Label(classify_frame, text="K value (for KNN only):")
knn_label.grid(row=1, column=0, padx=5, pady=5, sticky='e')
knn_entry = tk.Entry(classify_frame, textvariable=k_value, width=5)
knn_entry.grid(row=1, column=1, padx=5, pady=5, sticky='w')

# Options Frame
bg_check = tk.Checkbutton(options_frame, text="Exclude background (GT=0) for
    Training", variable=remove_bg)
bg_check.pack(padx=5, pady=5, anchor='w')

# Run Frame
tk.Button(run_frame, text="Run Classification and Display", command=
    classify_and_display, font=("Arial", 10, "bold")).pack(pady=10)

# Status Frame
status_label = tk.Label(status_frame, text="Load files to begin.", fg='blue',
    wraplength=450, justify="left", anchor='nw')
status_label.pack(pady=2, padx=5, fill='x')
status_label2 = tk.Label(status_frame, text="", fg='darkblue', wraplength=450,
    justify="left", anchor='nw')
status_label2.pack(pady=2, padx=5, fill='x')

# --- Main Loop ---
gui_frame.mainloop()
```

Listing 1: Complete Python Code