Machine Learning Report

Automated Material Stream Identification (MSI) System

1. Data Preprocessing and Augmentation

First, corrupted images were identified and removed from the original dataset to prevent invalid or misleading data from influencing the learning process. The cleaned dataset was then split into training and testing subsets before any augmentation. This step was performed intentionally to avoid data leakage, ensuring that no augmented version of a test image appears in the training set, and vice versa.

Eighty percent of the non-corrupted images from each category were allocated to the training set, with the remaining twenty percent reserved for testing. An exception was made for the *trash* category, which contained the smallest number of images and exhibited high intra-class variability. To address this limitation, 71 images (approximately 67% of the available data) were used for training and augmentation, while 35 original images were preserved for testing.

To reduce dataset bias and improve class balance, a target of 500 images per category was established for the training set. Data augmentation was applied only to training images and was performed with replacement, allowing multiple augmented samples to be generated from the same original image when necessary. The number of augmented images required to reach the target threshold was computed individually for each category and divided equally across three augmentation phases.

Each augmentation phase was designed to introduce progressively less aggressive transformations. This strategy ensures that substantial variability is introduced early while preventing classes with fewer original samples from being dominated by unrealistic or overly distorted images. The overall goal of this approach is to simulate realistic variations encountered in real-world conditions, thereby improving the robustness and generalization capability of the model.

Additionally, each class was assigned a class-specific augmentation strength, which controls the magnitude of applied transformations. These strengths were determined based on factors such as the visual characteristics of each material category, its tolerance to distortion, and the number of original images available.

Phase 1: Geometric and Lighting Variations

The first augmentation phase focuses on introducing geometric diversity and illumination variability. Images in this phase undergo random rotations, horizontal flips, random resizing, and cropping, as well as brightness and contrast adjustments. The probabilities associated with these augmentations are intentionally high, ranging from **0.7 to 1.0**, to maximize visual diversity and reduce sensitivity to orientation and lighting changes.

Phase 2: Sensor Noise and Motion Effects

The second phase emphasizes robustness to sensor-related artifacts and motion-induced distortions. During this phase, images are subjected to controlled brightness adjustments, mild Gaussian noise, and slight motion blur to emulate real-world image capture conditions. The probabilities of these transformations are lower than those in the first phase, ranging from **0.4 to 0.7**, to avoid excessive distortion while still enhancing realism.

Phase 3: Mild Regularization

The final augmentation phase applies minimal transformations, consisting of random resized cropping and subtle adjustments to brightness and contrast. This phase serves as a regularization step, reinforcing learned patterns without introducing significant visual alterations.

2. Feature Extraction

For converting raw images into fixed-length numerical feature vectors, we used a Convolutional Neural Network (CNN), specifically ResNet50 from PyTorch.

Steps:

- Each image was resized to 224×224 pixels, converted to a tensor, and normalized using standard values.
- The processed image was passed through the model to obtain a 2048-dimensional feature vector.
- Features from all training and testing images were extracted and saved as NumPy arrays (features.npy and labels.npy).

Feature extraction methods like HOG, LBP, or color histograms are simple but often perform poorly on real-world images with varying lighting, angles, and backgrounds. In contrast, features from a pretrained CNN like ResNet50 are much more powerful because the network has already learned to recognize complex patterns (edges, textures, shapes, and objects) from millions of images.

3. Classifier Models

SVM Model

The Support Vector Machine (SVM) classifier was implemented using the `SVC` class from the **scikit-learn** library. SVM was selected due to its strong performance in high-dimensional feature spaces and its effectiveness in multi-class classification problems.

The model was trained using a **Radial Basis Function (RBF)** kernel to capture non-linear relationships in the data. The optimal hyperparameters were determined using the **GridSearchCV** method, which systematically evaluated multiple parameter combinations to identify the configuration that best fits the training data.

The final hyperparameter settings used for training were as follows:

- **C = 10**
- **gamma = 'scale'**
- **kernel = 'rbf'**
- **class_weight = 'balanced'**
- **decision_function_shape = 'ovr'**
- **probability = True**

Class weighting was set to be balanced to address class imbalance by adjusting the importance of each class inversely proportional to its frequency in the dataset. Since the problem involves multiple classes, the **one-vs-rest (OvR)** decision function strategy was adopted, where a separate classifier is trained for each class against all others.

Enabling probability estimation allowed the model to output class probability scores rather than only hard class predictions. This capability was essential for handling unknown class detection

Unknown Class Detection

To enable unknown class classification, a probability-based thresholding approach was employed. During inference, the model computes the probability distribution across all known classes. If the highest predicted class probability is below a predefined threshold, the sample is classified as unknown.

In this project, the probability threshold was set to 0.6. Any input sample whose maximum class probability was less than 0.6 was labeled as unknown. This approach ensures that only predictions with sufficient confidence are assigned to one of the six known classes, while low-confidence predictions are safely rejected as unknown.

Accuracy Report

```
Accuracy: 0.9164

Classification Report:
              precision    recall  f1-score   support

           0       0.89      0.95      0.92        77
           1       0.94      0.91      0.93        89
           2       0.90      0.92      0.91        49
           3       0.94      0.90      0.92        72
           4       0.91      0.94      0.92        63
           5       0.89      0.81      0.85        21

    accuracy                           0.92       371
   macro avg       0.91      0.90      0.91       371
weighted avg       0.92      0.92      0.92       371
```

KNN Model

K-Nearest Neighbors (KNN) was implemented as a baseline classification model to evaluate the quality of the extracted features. After scaling the features and experimenting with several dimensionality reduction techniques—PCA, ICA, NMF, and LDA—PCA was selected as it provided the highest classification accuracy. The KNN model was trained with the following hyperparameters:

- **Distance metric:** Cosine, to better handle high-dimensional feature similarity
- **Weights:** Distance-based, so that closer neighbors have a stronger influence on predictions
- **Number of neighbors (K):** Optimized by testing multiple odd values and selecting the one with the highest validation accuracy
- **Algorithm:** Auto, allowing efficient neighbor search depending on the data
- **Parallelization:** Enabled via `n_jobs=-1` to use all CPU cores for faster training

These settings ensured that KNN could robustly classify samples while taking into account both distance and local neighbor distribution.

Although KNN provided reasonable performance, its accuracy and sensitivity to overlapping classes were lower compared to SVM. Therefore, KNN was mainly used for comparison, and Support Vector Machines (SVM) were selected as the final classification model due to their superior accuracy and generalization.

Accuracy Report

```
Accuracy of KNN: 0.8949, with K = 1

Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.90      0.91        77
           1       0.93      0.85      0.89        89
           2       0.90      0.96      0.93        49
           3       0.84      0.90      0.87        72
           4       0.89      0.92      0.91        63
           5       0.85      0.81      0.83        21

    accuracy                           0.89       371
   macro avg       0.89      0.89      0.89       371
weighted avg       0.90      0.89      0.89       371
```

4. Live System Deployment (realtime_app.py):

This script implements a live system interface that integrates our best-performing feature extractor using CNN (from feature_extraction2.py) together with our best-performing model (the trained SVM classifier) to allow us to classify materials into categories in real-time using nothing but a computer's webcam.

Key Features:

- Captures live video stream (@ 640×480) and performs inference once every second to achieve good performance.
- Provides an on screen feedback with a clear, centered prediction label and confidence percentage.
- Implements graceful error handling when opening the webcam, loading the model, and failures in the pipeline.

Pipeline:

- Frame capturing every second (using OpenCV).
- Loads and applies feature extraction on the image and receives the feature vector (needs PyTorch).

- Loads and applies feature scaling using our prefitted svm_scaler.pkl.
- Visualizes the predicted label returned from the model in a clear, bottom-centered box with the label & its confidence %.

To use: “python realtime_app.py”, and press ‘q’ to close the webcam and exit the app.

Note:- Make sure the prefitted models are present in “../classifiers”, if not then run ”pca.py”, “train_knn.py”, and “train_svm.py” once each before using the app.