EE651 : Computer Vision

Texture Based Attacks on Intrinsic Signature Based Printer Identification

COURSE-BASED PROJECT

Presented By:

Khushi Dutta (12240820)

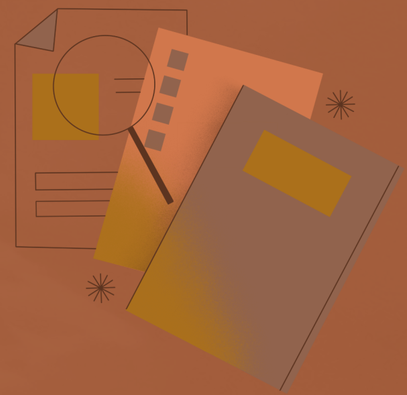
Ragini Vinay Mehta (12241420)

Submitted To:

Dr. Nitin Khanna

Associate Professor

Dept.of Electrical Engineering



Workflow

GENERATING PRINTER DATASET

GET ‘E’ & FEATURE EXTRACTION

- GLCM
- DFT
- VARIANCE & ENTROPY

SVM TRAINING

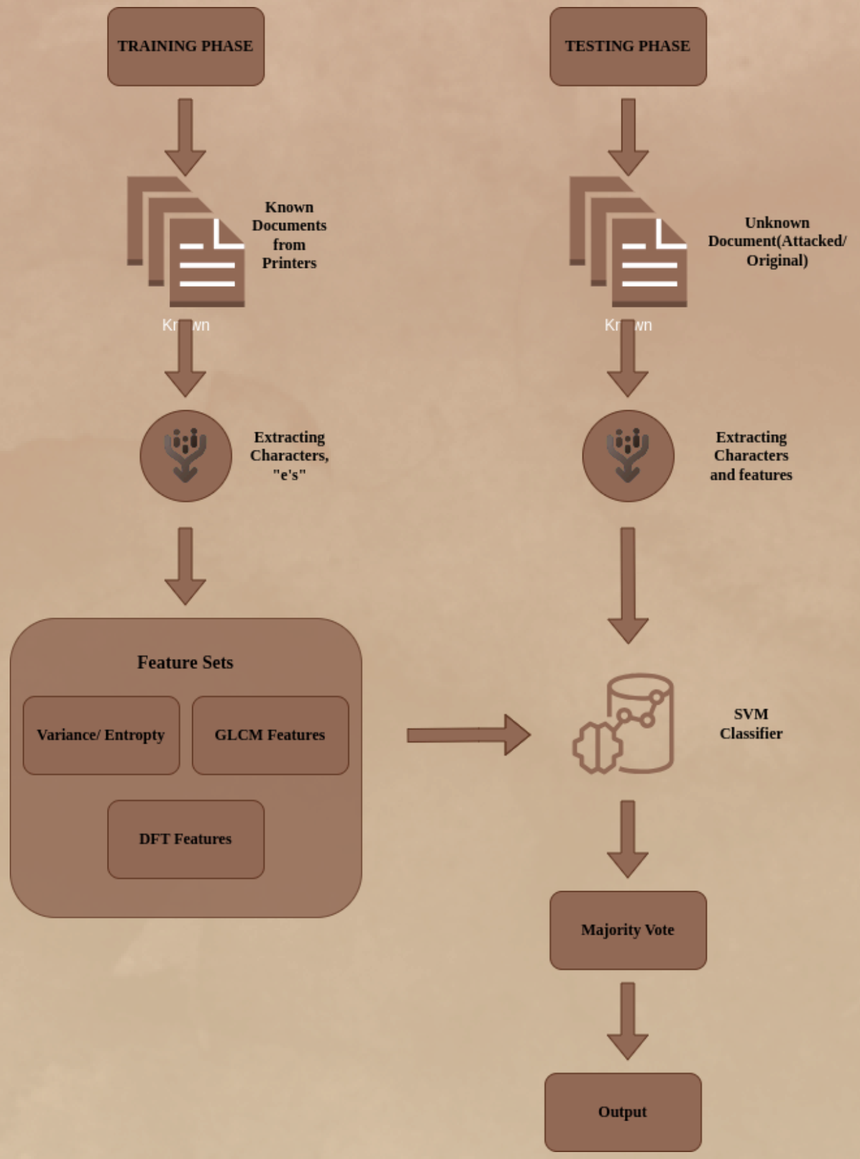
LDA VISUALIZATION IN RAW DATA

ATTACKS & TESTING DATASET

SVM TESTING

LDA PLOT ANALYSIS

Methodology



Generating Printer Dataset

- As the paper focuses on the intrinsic signatures left by printers during the printing process, a printer dataset has been generated to study and analyze such artifacts .
- The banding artifact was simulated using a sinusoidal function applied vertically across the image (perpendicular to the process direction), defined as:

$$I'(x, y) = I(x, y) + A \cdot \sin\left(2\pi f \cdot \frac{y}{H}\right)$$

- A: Amplitude controlling banding Strength
- f: Frequency of band repetition
- y: Row index
- H: Total number of image rows

```python
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

def apply_banding(img, f=10, A=0.01):

# See this is used for simulating printer banding artifact which is applied evenly across the i
    rows, cols = img.shape
    y = np.arange(rows).reshape(-1, 1)
    banding = A * np.sin(2 * np.pi * f * y / rows)

    banded_img = img + banding
    banded_img = np.clip(banded_img, 0, 1)
    banded_img = (banded_img * 255).astype(np.uint8)  # Convertto 8-bit image

    return banded_img

def save_image(array, name):
    img = Image.fromarray(np.clip(array, 0, 255).astype(np.uint8))
    img.save(name + ".jpg")

image_path = r"/home/khushi/Desktop/CV_Printer_Identification/Testing_images/TESTING_CV.jpg"
img = Image.open(image_path).convert('L')
img = np.array(img, dtype=np.float32)
img = img / 255.0

banded_img = apply_banding(img, f=10, A=0.01)
banded_img2 = apply_banding(img, f= 5, A = 0.1)
banded_img3 = apply_banding(img, f = 10, A=0.1)
```

Getting e's

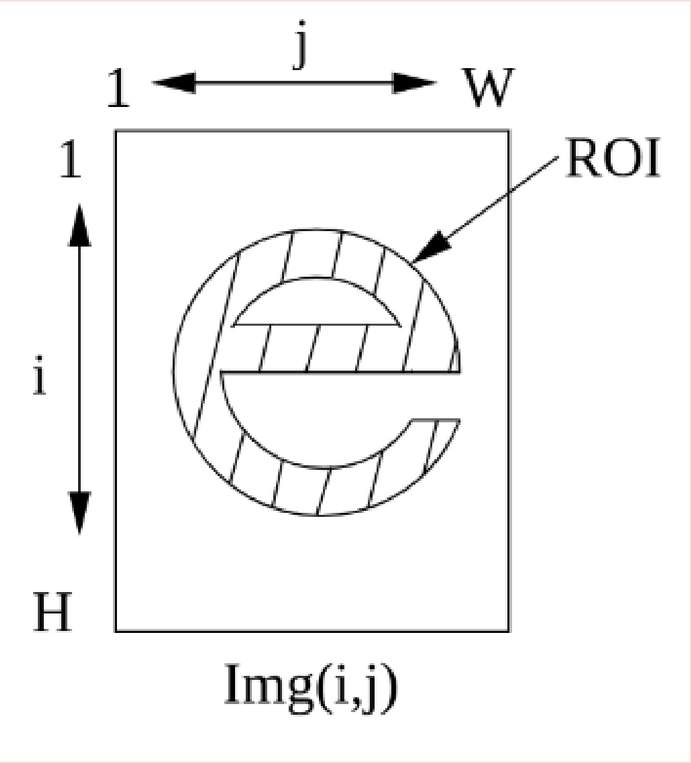
- Characters are extracted - e's most common
- Pytesseract used for extraction of e's from Images using bounding boxes of size 14 x 14 pixel - conscious decision
- stored as images → processed

Ever experienced the eerie elegance of endless serene evenings where every bree
ethereal, enveloping everything effortlessly?
These elements, while ephemeral, exert extreme energy over even the most reser
Eager eyes perceive these serene scenes,
every element evoking endless excitement. Elsewhere, evergreen meadows stretc
sheltering deer, bees, and gentle creatures
beneath the evergreen trees. Here, serenity settles, rendering even the deepest res
obsolete. Every step, even the feeblest,
feels deliberate, delibe like the careful etchings engraved into eternal stones.
wherever essence prevails, p e
emerges, despite deep-seated f s. Hence, whenever events render experiences e
remember the essence embedd etween fleeting
moments. The ever-present energy enveloping every breeze, every leaf, every wh
eternally. Gentle energies elev
everywhere—believe.

Beneath the velvet expanse, e cenes emerge, enveloped in nebulo
experience exceeds expectatio
scene, ever-changing, ever-mo eriences, expressed through endles
generate eternal memories. Se
the breeze delivers essence, ephemeral yet everlasting. Every traveler re
decades later. Evenings bleed into nights,
then reemerge anew, eternal in their rhythm. The gentle essence enters the deepes
erasing the remnants of yesterday. Every
being deserves serene experiences; everyone breathes, believes, and becomes mo
Elevate existence, embrace every element,

Feature Extraction

REGION OF INTEREST - R



Region of interest :

1. Calculated using thresholding, that is the number of pixels that make up the charecter ‘e’
2. Denoted by R throughout the code
3. Used to normlaize GLCM histograms as shown next

```python
# get region of interest ROI
def extract_roi(img, threshold):
    if len(img.shape) != 2:
        raise ValueError("Input image must be grayscale")
    roi_mask = (img < threshold).astype(np.uint8)
    # roi_pixels = np.sum(roi_mask)
    return roi_mask
```

Printer identification based on graylevel co-occurrence features for security and forensic applications

Feature Extraction

GLCMS

final vector of size - 20 x 10, captures texture-based features

$$glcm(n,m) = \sum_{(i,j),(i+dr,j+dc)\epsilon ROI} 1_{\{Img(i,j)=n, Img(i+dr,j+dc)=m\}}$$

$$R_{glcm} = \sum_{(i,j),(i+dr,j+dc)\epsilon ROI} 1$$

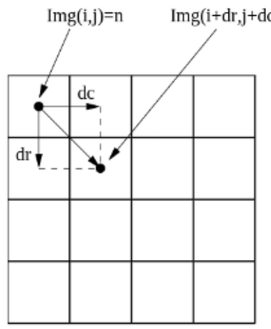


Figure 5. Generation of $glcm(n,m)$

$$p_{glcm}(n,m) = \frac{1}{R_{glcm}} glcm(n,m)$$

GLCM is a 4D matrix that is a histogram of 4 things - value pairs, number of occurances, distances and angles.

1. no angle considered in this paper
2. we use a 3d matrix
3. distances upto 10 taken

Features calculated :

A. Marginal Probabalities - GLCM -
   1. mean of row
   2. mean of column
   3. variance of row
   4. variance of column

B. Direct GLCM Metrics - GLCM
   1. Energy (sum of squares of marginal probs)
   2. hxy1: Cross entropy using marginal product
   3. hxy2: Entropy of marginal product
   4. hglcm: Entropy of the GLCM itself
   5. Max prob in GLCM values P_max
   6. correlation
   7.diagonal correlation

C. From Difference Matrices - Diff GLCM
   1. Energy
   2.Entropy
   3.Inertia
   4. d. Local Homogenteity H_d

D. From Sum Matrices - Sum GLCM
   1.Energy
   2.Entropy
   3. Variance
   4. cluster shade
   5. cluster prominence

Printer identification based on graylevel co-occurrence features for security and forensic applications

Feature Extraction

DFT Features

- 15-point DFT features are extracted from the e's extracted.
- The banding artifacts shows up in row wise. So, banding projection helps in converting it to interpretable 1D signal.

$$b(i) = \frac{\sum_{(i,j)\epsilon ROI} Img(i,j)}{\sum_{(i,j)\epsilon ROI} 1}.$$

- And, 240 has been taken as the banded projection length.
- Instead of taking all the frequencies, frequency bins have been formed {10, 20, ...150} cycles/inches. At these frequencies, we calculate DFT instead of all the 240 points.

$$P_b(n) = \sum_{k=0}^{239} b(k) e^{-j\frac{2\pi kn}{240}}.$$

```python
def normalized_projection(img, roi_mask):
    proj = np.sum(img * roi_mask, axis = 1)
    norm = np.sum(roi_mask, axis = 1)
    b = np.zeros_like(proj, dtype=float)
    for i in range(len(b)):
        if norm[i]>0:
            b[i] = proj[i]/norm[i]
    return b


def dft_features(b):
    """More direct implementation of Equation 6"""
    if len(b) < 240:
        b_padded = np.pad(b, (0, 240 - len(b)), mode='constant')
    else:
        b_padded = b[:240]

    N = 240
    features = np.zeros(15)

    # Frequencies centered at [10,20,...,150] cycles/inch
    for i in range(15):
        n = (i + 1) * 10  # 10,20,...,150
        # Calculate the DFT at frequency n
        dft_val = 0
        for k in range(N):
            dft_val += b_padded[k] * np.exp(-1j * 2 * np.pi * n * k / N)
        features[i] = np.abs(dft_val)

    return features
```

Feature Extraction

Pixel Based Features

1. Variance

It calculates variance by calculating the mean of the ROI(region of interest) and then calculate the variance(how much the pixel values deviate from the mean)

$$\mu_{Img} = \frac{1}{R} \sum_{(i,j)\epsilon ROI} Img(i,j)$$

$$\sigma^2_{Img} = \frac{1}{R} \sum_{(i,j)\epsilon ROI} (Img(i,j) - \mu_{Img})^2$$

2. Entropy

This calculates the intensity occurs at the ROI and then get the probability distribution function and find the entropy(randomness or complexity of pixel intensity distribution).

$$p_{Img}(\alpha) = \frac{1}{R} \sum_{(i,j)\epsilon ROI} 1_{\{Img(i,j)=\alpha\}}$$

$$h_{Img} = -\sum_{\alpha=0}^{255} p_{Img}(\alpha) \log_2 p_{Img}(\alpha)$$

```python
def compute_variance(img, roi_mask):
    roi_indices = np.where(roi_mask)
    roi_pixels = img[roi_indices]
    R = len(roi_pixels)
    if R == 0:
        return 0
    mu = np.mean(roi_pixels)
    variance = np.sum((roi_pixels - mu)**2)/R # variance
    return variance


def compute_entropy(img, roi_mask):
    roi_indices = np.where(roi_mask)
    roi_pixels = img[roi_indices]
    R = len(roi_pixels)
    if R == 0:
        return 0

    hist, _ = np.histogram(roi_pixels, bins=256, range=(0, 255)) # probability density functio
    p = hist / R  # p_Img(alpha)
    entropy = -np.sum(p * np.log2(p + 1e-10)) # this is added to avoid log 0 ...can think of it
    return entropy
```

FEATURE CONCATENATION

Features have to be concatenated to form a final feature vector per charceter extracted:

1. DFT features repeated along 10 columns as one only 1D feature is extracted per 'e' - final dimension 15 x 1
2. GLCM has a dimension of 10, since we have 10 distances considered - final dimension 22 x 10
3. Additional 2 vectors have dimension 2 x 1

Final vector = (20+ 15 + 2) x 10 = 37 x 10

- Since there are 50 characters per image that is extracted, the number of vector we get is 50 x 37 x 10.
- The matrix is then divided into 10 matrixes such that we have 10 matrixes each of dimension 50 x 37
- Here 37 means 37 different features, each of which play a different part

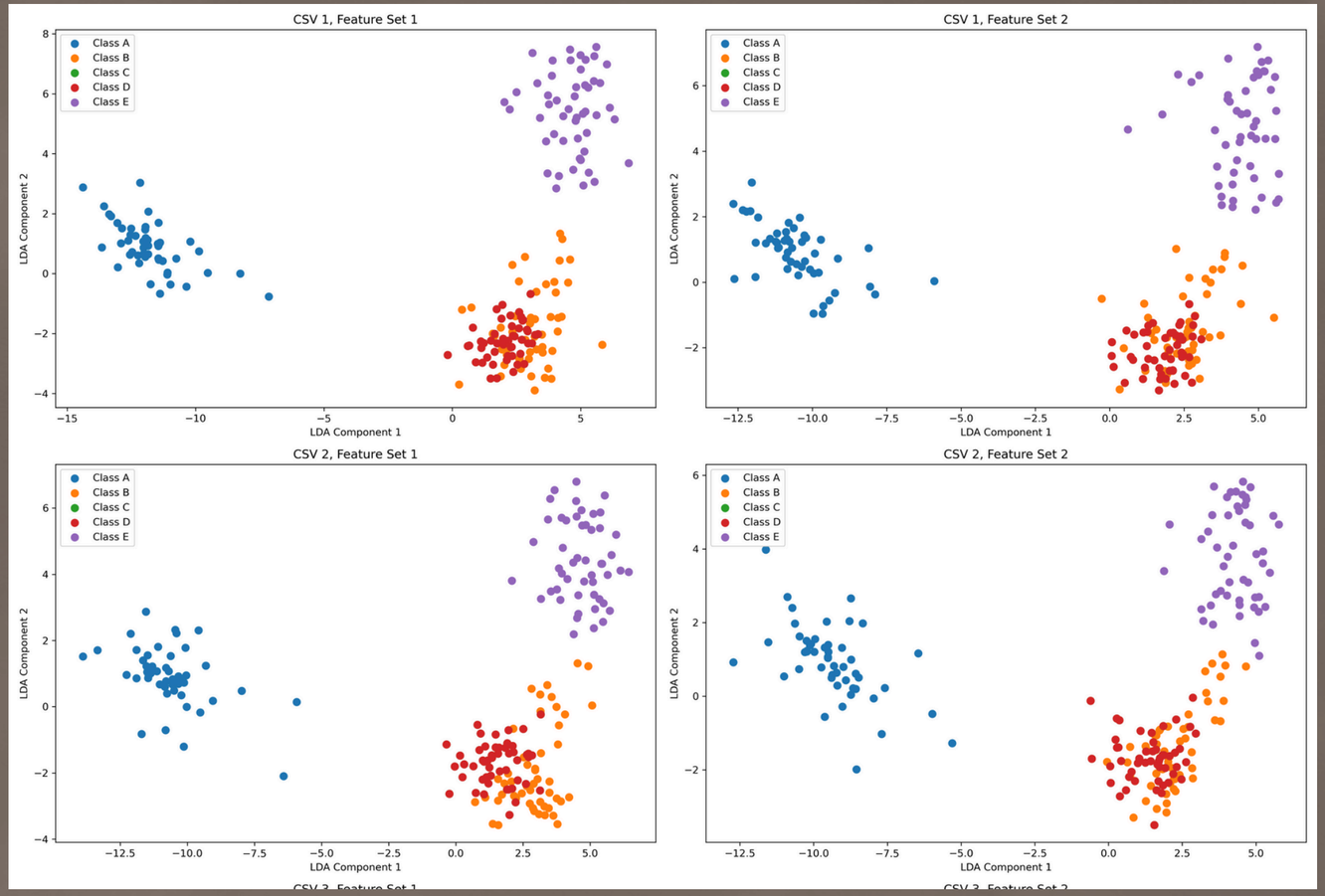
LDA

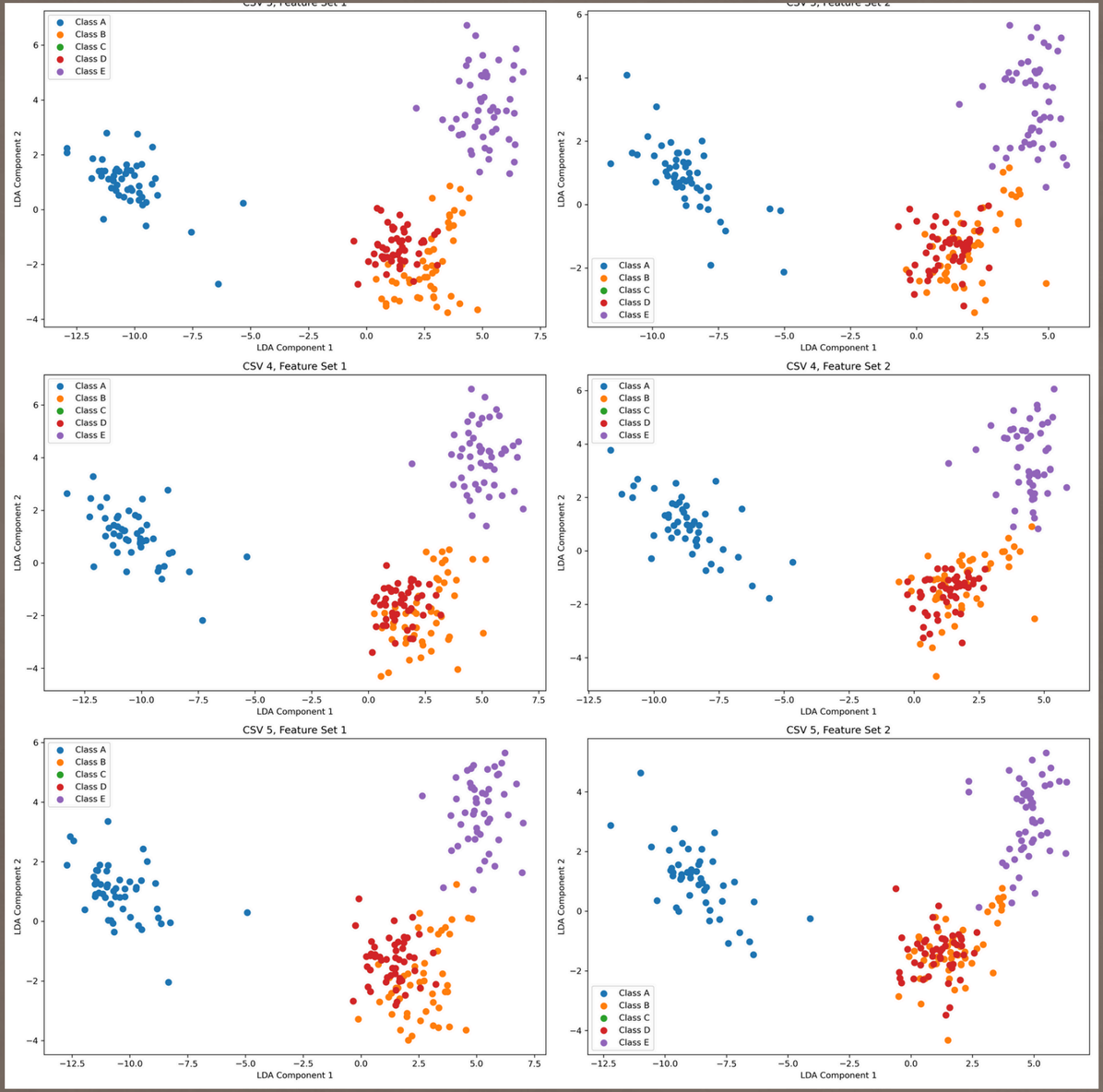
LDA is a supervised, dimension reduction techniques, that we use here to visualize the differences between the clusters.

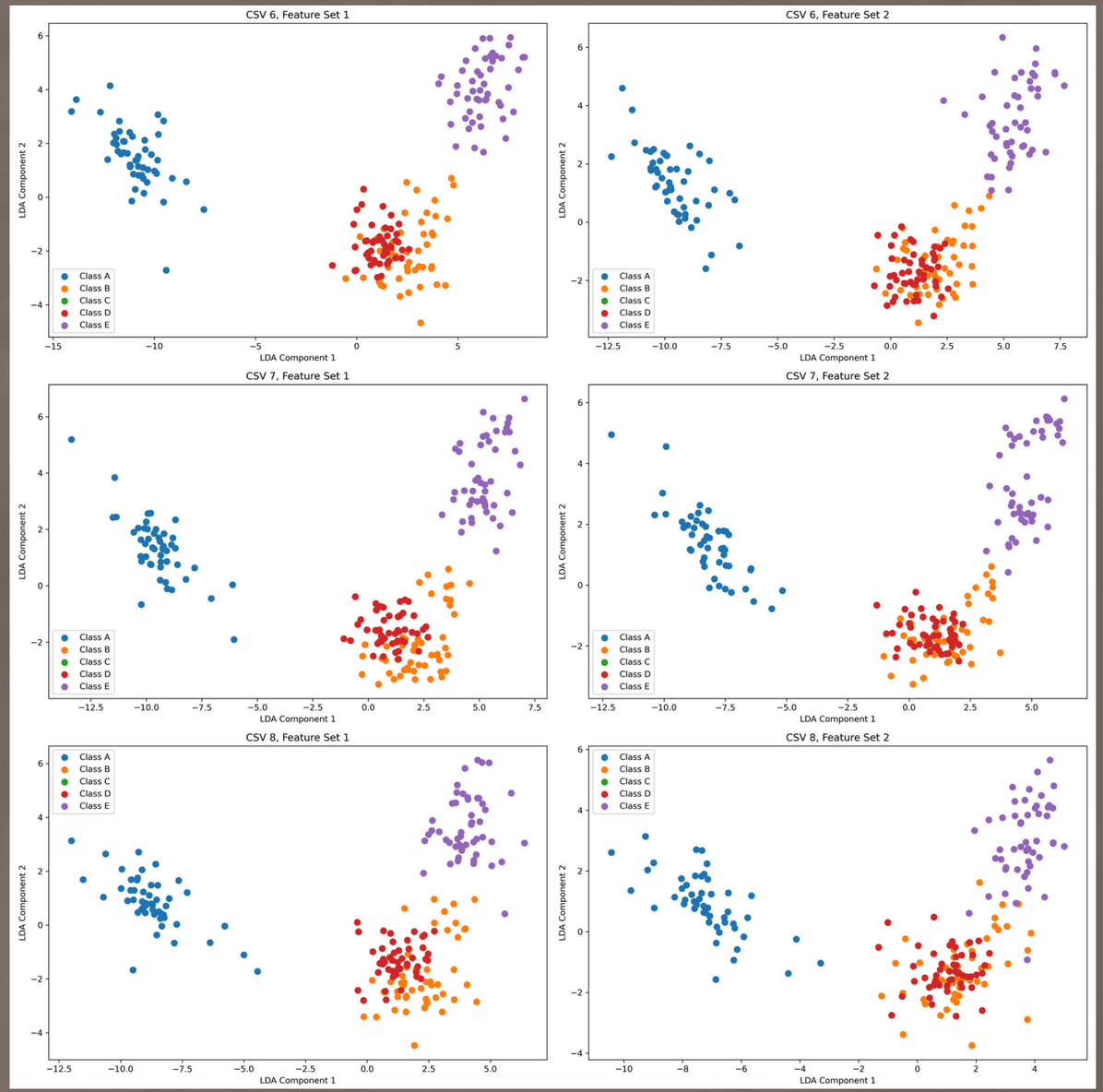
Training Phase

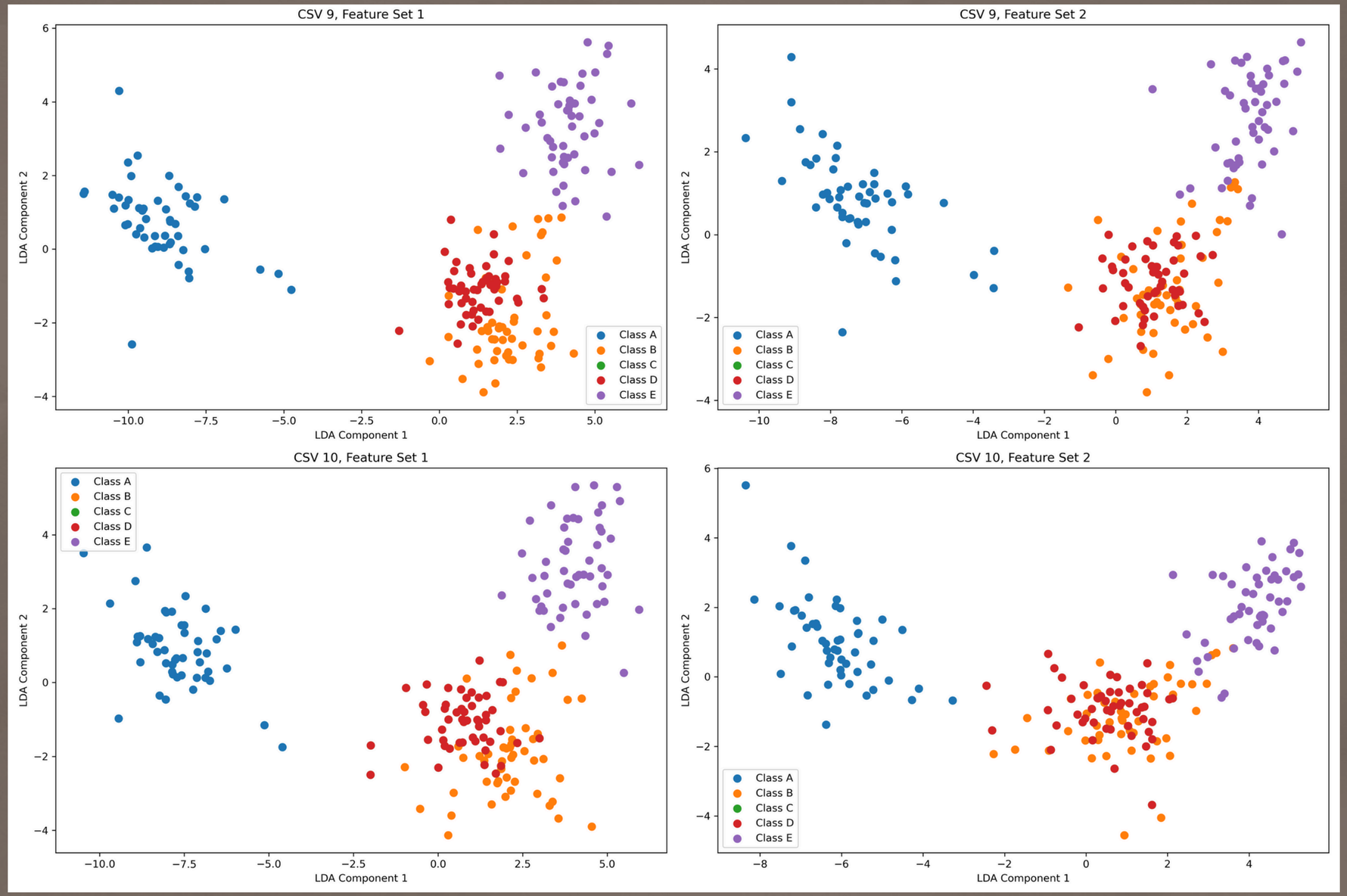
We can visualize the effect on different features based on the classes of printers that are being considered. Here is apparent that the GLCM clusters are more distinguishable than DFT.

Arranged in increasing order ofdistances we have:




CSV 4, Feature Set 1
CSV 4, Feature Set 2
CSV 5, Feature Set 1
CSV 5, Feature Set 2
Class A
Class B
Class C
Class D
Class E
LDA Component 1
LDA Component 2

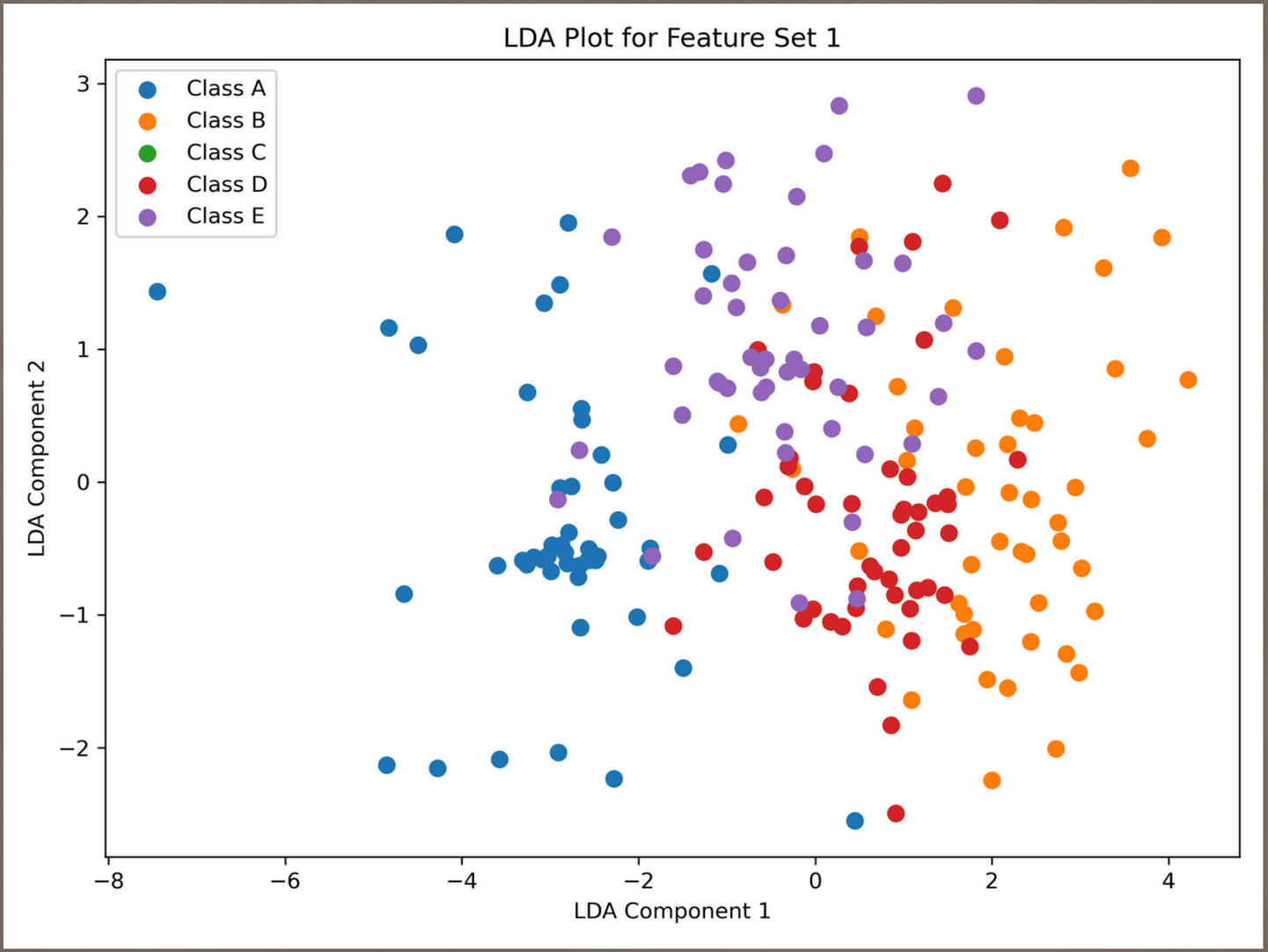

CSV 6, Feature Set 1
CSV 6, Feature Set 2
CSV 7, Feature Set 1
CSV 7, Feature Set 2
CSV 8, Feature Set 1
CSV 8, Feature Set 2
Class A
Class B
Class C
Class D
Class E
LDA Component 1
LDA Component 2



It visible that the clusters blue, purple and red are very well seperable.

It is also visible that the clusters red and orange are very overlapping, but still have most seperablity possible in distcance 10.

Whereas the DFT feature plot looks like:



There is some degree of seperablity in most of them, yet they are very overlapping.

Attacks

Attack 1: Fixed Frequency Sinusoidal Signal

- Adds a grayscale sinusoid to printed (black) text lines:

$$s_1(i) = \frac{A}{2}\left[1 + cos\left(\frac{2\pi f i}{R_p}\right)\right]$$

- Only affects pixels where I(i,j) = 0 (text).

Attack 2: Binarized Sinusoidal Attack

- Converts Attack 1 to binary:
  - Uses a sinusoid-based threshold:

$$s_2(i) = \frac{1}{16}\left[1 + cos\left(\frac{2\pi f i}{R_p}\right)\right].$$

  - For text pixels, flip to white (255) if a random number < $s_2(I)$.

Attack 3: Frequency Hopping Sinusoid

- Adds variable-frequency sinusoid:

$$s_3(i) = \frac{A}{2}\left[1 + cos\left(\phi(i)\right)\right],$$

$$\phi(i) = \phi(i-1) + \frac{2\pi f(i)}{R_p},$$

- f(i) hops randomly across rows (30–120 cycles/inch), staying constant over α rows.

```python
#Attack 1
def attack_fixed_freq(image, A, f, Rp):
    height, width = image.shape
    s1 = (A/2) * (1 + np.cos(2 * np.pi * f * np.arange(height)/Rp))

    s1 = s1[:, np.newaxis]
    attacked = np.where(image == 0, image+s1, 255)
    return attacked

#Attack 2
def attack_fixed_freq_binarized(image, A, f, Rp):
    height, width = image.shape
    s2 = (1/16) * (1 + np.cos(2 * np.pi * f * np.arange(height)/Rp))
    s2 = s2[:, np.newaxis]
    rand = np.random.rand(height, width)  # this is a random number for thresholding
    attacked = np.where((image == 0) & (rand < s2), 255, image)
    return attacked

#Attack 3
def attack_freq_hopping(image, A, Rp):
    height, width = image.shape
    phi = np.zeros(height)
    f_values = []
    alpha_values = []

    # Generate frequency segments
    i = 0
    while i < height:
        # Randomly select frequency and duration
        f = np.random.randint(30, 121)  # Frequency in [30,120]
        alpha = np.random.randint(0, 101)  # Duration in [0,100] rows
        alpha = max(1, alpha)

        # Apply same frequency for alpha rows
        segment_end = min(i + alpha, height)
        for row in range(i, segment_end):
            if row == 0:
                phi[row] = 0      #phi(0) = 0
            else:
                phi[row] = phi[row-1] + (2 * np.pi * f) / Rp

        i = segment_end

    s3 = (A / 2) * (1 + np.cos(phi))
    s3 = s3[:, np.newaxis]
    attacked = np.where(image == 0, image + s3, 255)
    return attacked
```

Contd...

Attack 4: Binarized Frequency Hopping

- Binary version of Attack 3:
  - Decision threshold:

$$s_4(i) = \frac{1}{2^\nu}\left[1 + cos\left(\phi(i)\right)\right]$$

  - Larger ν reduces noise power.
- Offers stronger stealth in binary documents.

Attack 5: Gaussian Noise Injection

- Adds bounded Gaussian noise to black pixels:

$$\tilde{I}(i,j) = \begin{cases} I(i,j) + \mathbf{X} & , \quad I(i,j) = 0 \quad and \quad \mathbf{X} \in [\mathbf{0}, \mathbf{3}\sigma] \\ I(i,j) & , \quad else \end{cases}$$

- Randomizes banding without frequency structure.

```python
#Attack 4
def attack_freq_hopping_binarized(image, Rp, nu=4):
    height, width = image.shape
    phi = np.zeros(height)
    f_values = []
    alpha_values = []

    # Generate frequency segments (same as Attack 3)
    i = 0
    while i < height:
        f = np.random.randint(30, 121)
        alpha = np.random.randint(0, 101)
        alpha = max(1, alpha)

        segment_end = min(i + alpha, height)
        for row in range(i, segment_end):
            if row == 0:
                phi[row] = 0
            else:
                phi[row] = phi[row-1] + (2 * np.pi * f) / Rp

        i = segment_end

    s4 = (1 / (2**nu)) * (1 + np.cos(phi))
    s4 = s4[:, np.newaxis]
    rand = np.random.rand(height, width)
    attacked = np.where((image == 0) & (rand < s4), 255, image)
    return attacked

#Attack 5
def attack_gaussian_noise(image, sigma = 5):
    height, width = image.shape
    noise = np.random.normal(3 * sigma/2, sigma, size=(height, width))
    attacked = np.where(image == 0, image+noise, image)
    return attacked
```

SVM Training and Testing

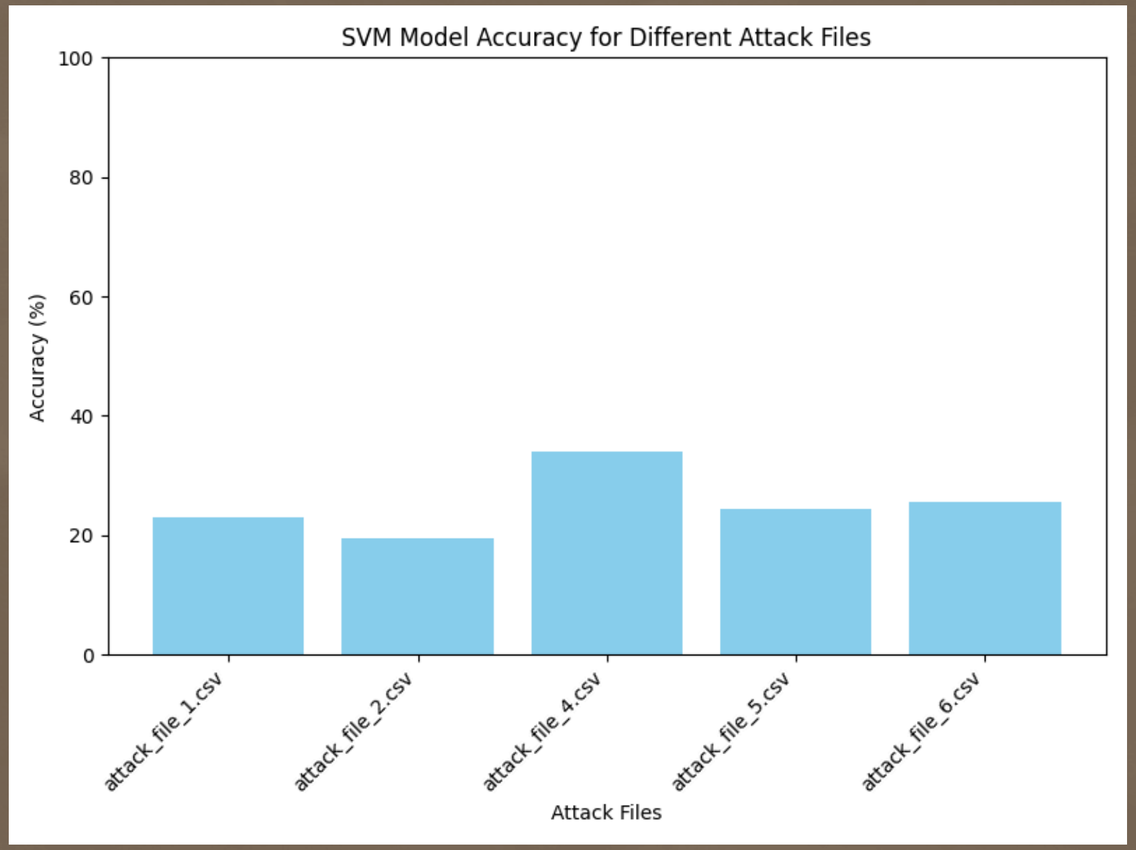
- As there we have 10 different features vectors for each column representing the distance in GLCM, so need to train 10 different SVM models to get the best performing dist.

- On visualizing the LDA plots of raw data, we found that frequency 10 and amplitude 0.04 was underperforming.

- Then again, trained models based on 4 classes and found that column 10 was the best performing one among all.

- So, took that model and test it on all the attacked version feature vectors of 5 different attacks.

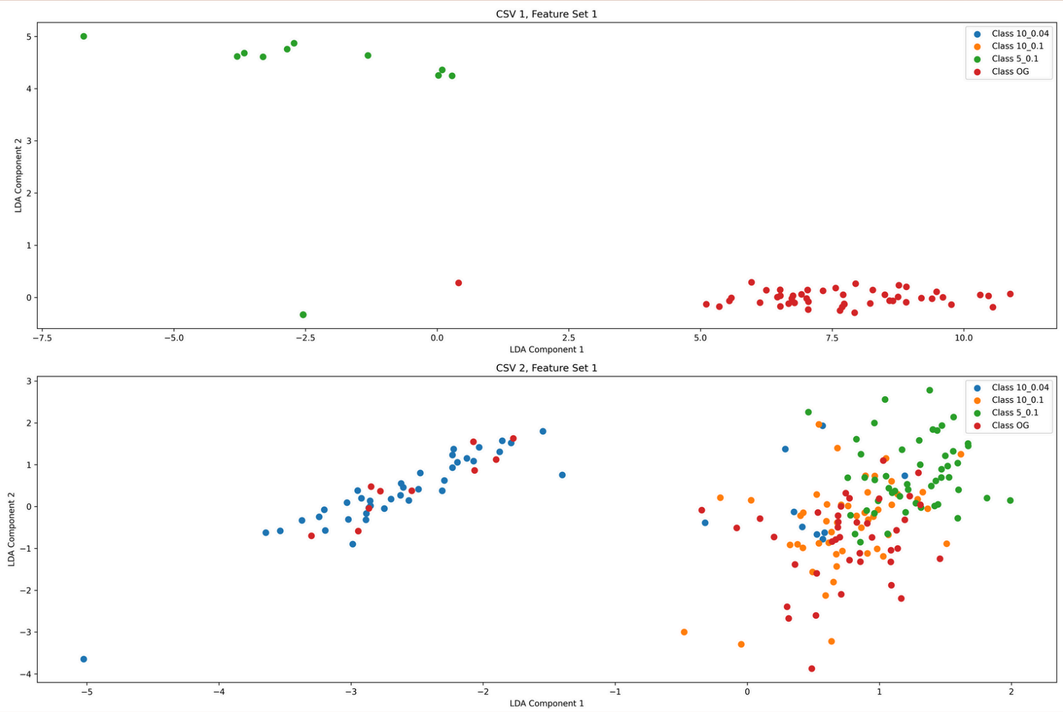
```python
import pandas as pd
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
import joblib

def train_and_save_svm_model(feature_csv_path, label_csv_path, model_save_path='svm_model.joblib'):

    X = pd.read_csv(feature_csv_path)
    y = pd.read_csv(label_csv_path).squeeze()

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    svm = SVC(kernel='rbf', probability=True)
    svm.fit(X_train, y_train)

    # Evaluate model
    print("\nEvaluating model...")
    train_accuracy = svm.score(X_train, y_train)
    test_accuracy = svm.score(X_test, y_test)

    print(f" Training Accuracy: {train_accuracy:.4f}")
    print(f" Testing Accuracy: {test_accuracy:.4f}\n")

    y_pred = svm.predict(X_test)
    print("Classification Report:")
    print(classification_report(y_test, y_pred))

    joblib.dump(svm, model_save_path)
    print(f"\n Model saved as {model_save_path}")

    return svm

svm_model = train_and_save_svm_model(
    'dist_csvs/trimmed_column_1.csv',
    'dist_csvs/trimmed_class_label.csv',
    model_save_path='svm_removedC_col1.joblib'
)
```

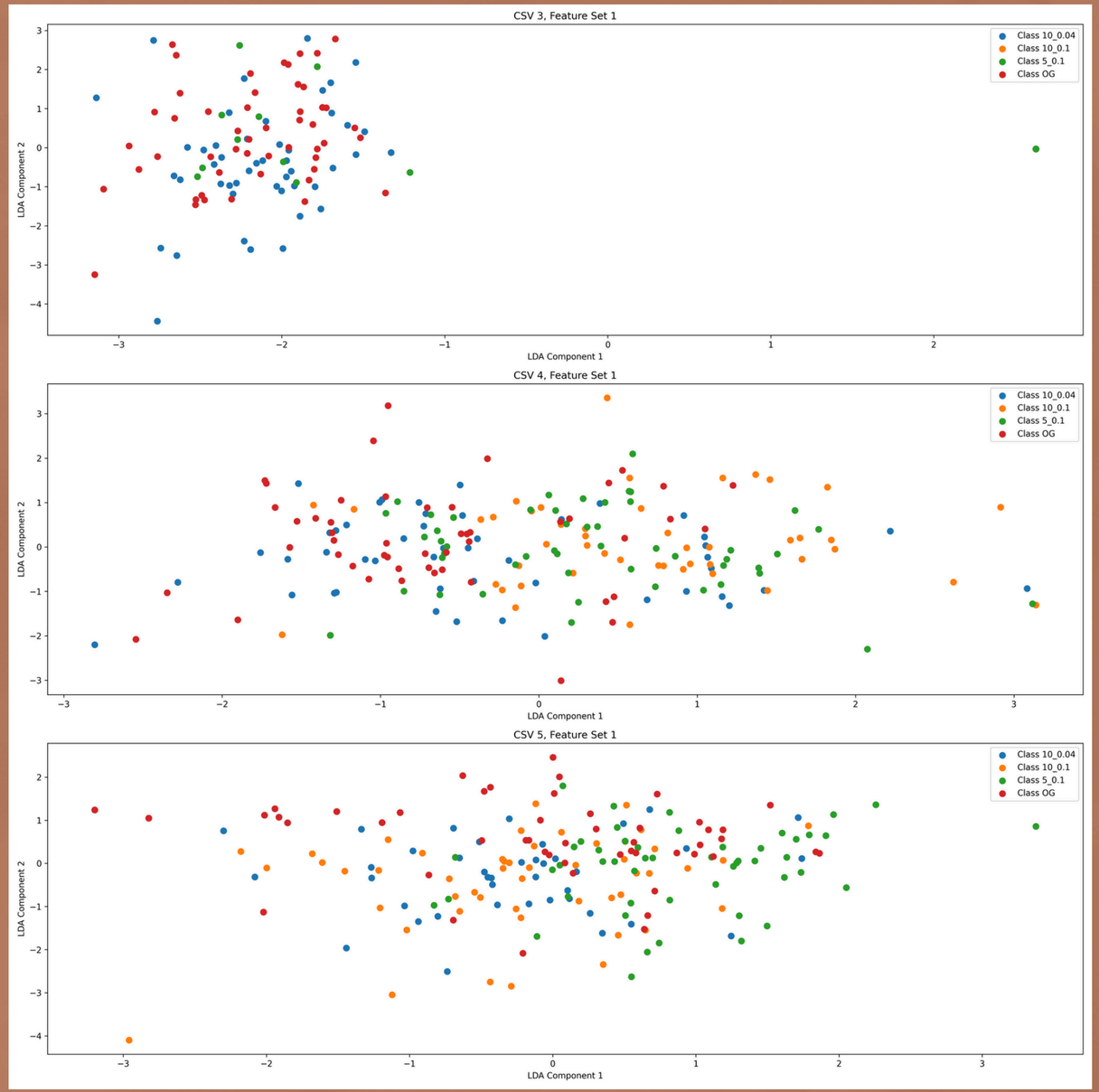


This is the bar graph of accuracies of all the attacked versions. The results are not satisfying because of poor image quality and the aggressive attacks have deteriorated the images which makes it difficult to extract the e's. The frequency hopping attack is easily identifiable.

Testing Phase

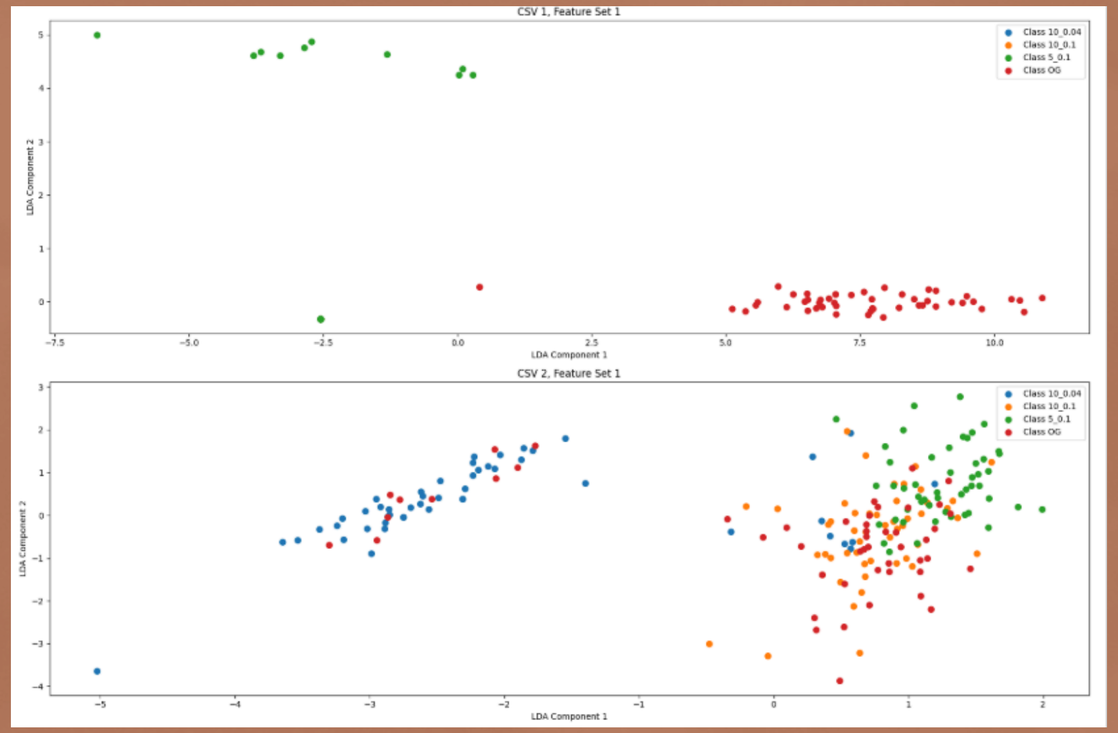
For testing phase, we need to see the attacked versions of the printer pages and what the effect on different features look like.

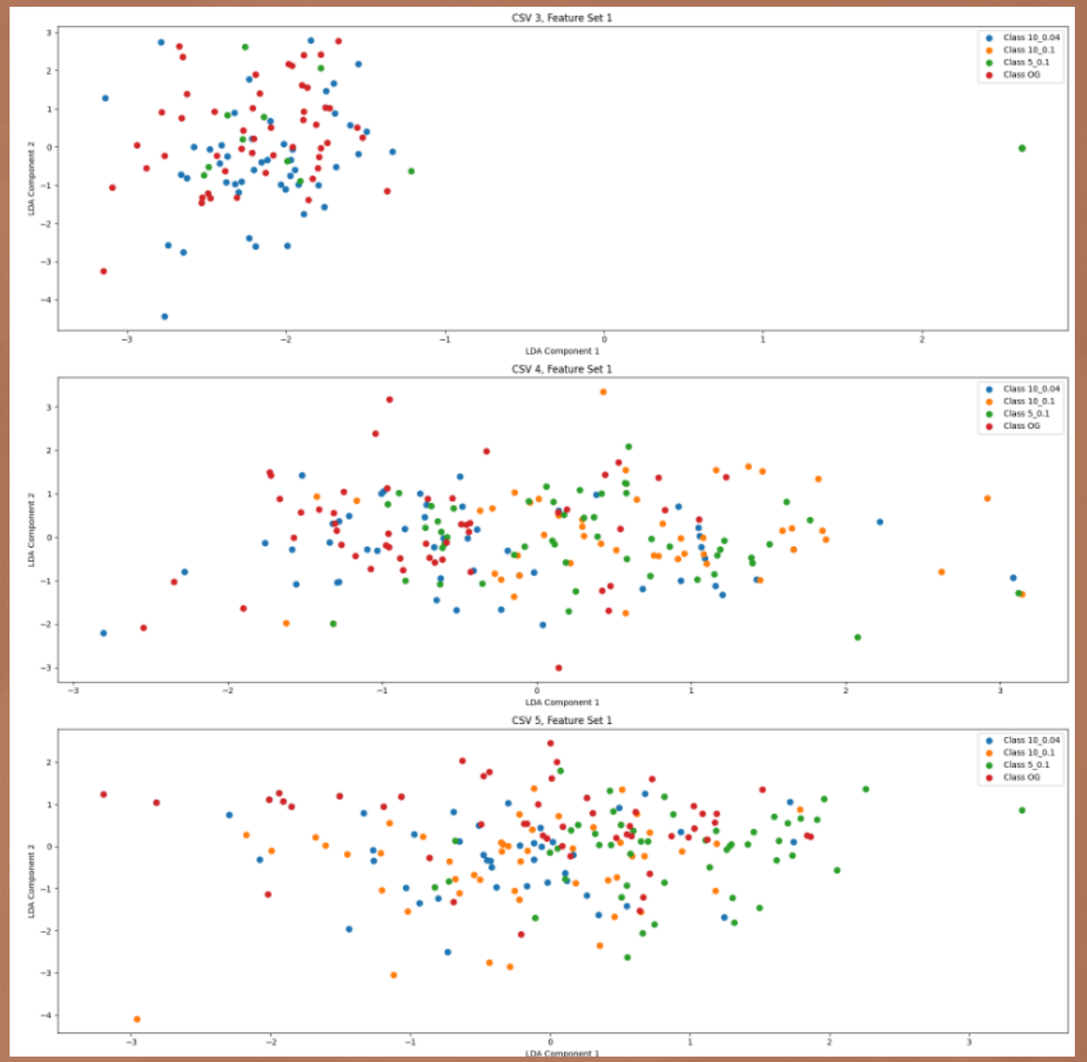




GLCM features are not very seperable anymore after the attacks.

DFT features





Seperablity is not very good for DFT after attacks as well.

ISSUES

- Dimensional Transformation Challenges: Significant effort was needed to transform data across dimensions — from 3D GLCM matrices to 3D vectors, and finally to 2D statistical features like mean and variance. The process was non-trivial and time-consuming.
- Character Extraction Issues: PyTesseract often misidentified characters; not all extracted symbols were 'e'.
- Image Enhancement: OpenCV filters were used to enhance resolution for better character recognition.
- Sample Size Strategy: At least 50 'e's per printer base page were extracted to ensure a reliable sample, even with some misclassifications.
- GLCM Normalization: Each character image's GLCM was normalized based on its region of interest (ROI).
- Low Resolution Constraints: Work was done on 14x14 pixel images (vs. 160–180 pixels in referenced papers), limiting texture information and affecting result fidelity.
- Manual Bounding Box Adjustments: Fine-tuning was necessary due to small character sizes and overlapping features from adjacent characters.
- Trade-off: There was a balance between quality and accuracy — priority was given to fitting the constraints of the project setup.
- Attack Sensitivity: Image degradation from attacks like fixed frequency and frequency hopping made accurate character extraction more difficult.
- Rp Value Note: Rp was arbitrarily chosen, as the project simulated printer artifacts rather than using multiple physical printers.

Thank You!!