a2-2022315

February 14, 2025

Import dependencies

[86]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Load images from mnist dataset

[87]:
```
def read_header(f):
    if (magic := int.from_bytes(f.read(4), byteorder='big')) != 2051:
        raise ValueError(f'Unexpected file format: magic number {magic}')
    return tuple(int.from_bytes(f.read(4), byteorder='big') for _ in range(3))

def read_image_data( num_images, num_rows, num_cols,f):
    return np.frombuffer(f.read(), dtype=np.uint8).reshape(num_images,␣
 ↪num_rows, num_cols)

def load_images(file_path):
    with open(file_path, 'rb') as f:
        num_images, num_rows, num_cols = read_header(f)
        images = read_image_data(num_images, num_rows, num_cols,f)
    return images
```

Read labels

[88]:
```
def read_magic_number(f):
    magic = int.from_bytes(f.read(4), byteorder='big')
    if magic != 2049:
        raise ValueError(f'Invalid magic number {magic} in MNIST label file')
    return magic

def read_num_labels(f):
    return int.from_bytes(f.read(4), byteorder='big')

def read_labels(f, num_labels):
    return np.frombuffer(f.read(num_labels), dtype=np.uint8)

def load_labels(file_path):
    with open(file_path, 'rb') as f:
        read_magic_number(f)
```

```
        num_labels = read_num_labels(f)
        labels = read_labels(f, num_labels)
    return labels
```

Preprocess data

[89]:
```
def load_data(image_path, label_path):
    return load_images(image_path), load_labels(label_path)

def filter_digits(images, labels, digits=None):
    if digits is None:
        digits = [0, 1, 2]
    return images[np.isin(labels, digits)], labels[np.isin(labels, digits)]

def preprocess_images(images):
    return images.reshape(images.shape[0], -1) / 255.0

def sample_images(images, labels, num_samples=100, seed=42):
    np.random.seed(seed)
    selected_indices = np.concatenate([
        np.random.choice(np.where(labels == c)[0], num_samples, replace=False)␣
 ↪for c in np.unique(labels)
    ])
    return images[selected_indices], labels[selected_indices]
```

Load train images

[90]:
```
train_images, train_labels = load_data(
    'archive/train-images-idx3-ubyte/train-images-idx3-ubyte',
    'archive/train-labels-idx1-ubyte/train-labels-idx1-ubyte')
```

Load test images

[91]:
```
test_images, test_labels = load_data('archive/t10k-images-idx3-ubyte/
 ↪t10k-images-idx3-ubyte',
                                     'archive/t10k-labels-idx1-ubyte/
 ↪t10k-labels-idx1-ubyte')
```

Filter required digits

[92]:
```
x_train, y_train = filter_digits(train_images, train_labels)
x_test, y_test = filter_digits(test_images, test_labels)
```

[93]:
```
x_train = preprocess_images(x_train)
x_test = preprocess_images(x_test)
```

[94]:
```
x_train, y_train = sample_images(x_train, y_train)
x_test, y_test = sample_images(x_test, y_test)
```

PCA Functions

[95]:
```
def compute_covariance(X):
    return np.cov(X, rowvar=False)

def center_data(X):
    return X - np.mean(X, axis=0), np.mean(X, axis=0)
def eigen_decomposition(cov):
    eigenvalues, eigenvectors = np.linalg.eigh(cov)
    return eigenvalues, eigenvectors

def sort_eigenpairs(eigenvalues, eigenvectors):

    idx = np.argsort(eigenvalues)[::-1]
    return eigenvalues[idx], eigenvectors[:, idx]

def enforce_sign_convention(components):
    for i in range(components.shape[1]):
        if components[0, i] < 0:
            components[:, i] *= -1
    return components

def select_n_components(eigenvalues, variance_ratio):
    total_var = np.sum(eigenvalues)
    cumulative_variance = np.cumsum(eigenvalues) / total_var
    return np.searchsorted(cumulative_variance, variance_ratio) + 1

def fit_pca(X, n_components=None, variance_ratio=None):
    X_centered, mean = center_data(X)
    cov = compute_covariance(X_centered)
    eigenvalues, eigenvectors = eigen_decomposition(cov)
    eigenvalues, components = sort_eigenpairs(eigenvalues, eigenvectors)
    components = enforce_sign_convention(components)

    if variance_ratio is not None:
        n_components = select_n_components(eigenvalues, variance_ratio)
    elif n_components is None:
        n_components = X.shape[1]

    return mean, components, eigenvalues, n_components

def transform_pca(X, mean, components, n_components):
    X_centered = X - mean
    return np.dot(X_centered, components[:, :n_components])
```

[96]:
```
mean, components, eigenvalues, n_components = fit_pca(x_train, variance_ratio=0.
↪95)
```

[97]:
```
x_test_pca = transform_pca(x_test, mean, components, n_components)
x_train_pca = transform_pca(x_train, mean, components, n_components)
```

[98]:
```
print(f"Reduced dimension-> {n_components}")
print(f"Original dimension-> {x_train.shape[1]}")
```

```
Reduced dimension-> 82
Original dimension-> 784
```

LDA Functions

[99]:
```
def compute_class_statistics(X, y):
    classes = np.unique(y)
    means = {}
    priors = {}

    for c in classes:
        Xc = X[y == c]
        means[c] = np.mean(Xc, axis=0)
        priors[c] = Xc.shape[0] / X.shape[0]

    return classes, means, priors

def compute_shared_covariance(X, y, means):
    n_features = X.shape[1]
    S_w = np.zeros((n_features, n_features))

    for c in means:
        Xc = X[y == c]
        cov_c = np.cov(Xc, rowvar=False, bias=True)
        S_w += Xc.shape[0] * cov_c

    S_w = S_w / X.shape[0]
    inv_cov = np.linalg.inv(S_w + 1e-6 * np.eye(n_features))

    return S_w, inv_cov

def fit_lda(X, y):
    classes, means, priors = compute_class_statistics(X, y)
    cov, inv_cov = compute_shared_covariance(X, y, means)
    return classes, means, priors, cov, inv_cov

def predict_lda(X, classes, means, priors, inv_cov):
    preds = []

    for x in X:
        scores = []
        for c in classes:
```

```
            mean = means[c]
            score = x.dot(inv_cov).dot(mean) - 0.5 * mean.dot(inv_cov).
 ↪dot(mean) + np.log(priors[c])
            scores.append(score)
        preds.append(classes[np.argmax(scores)])

    return np.array(preds)

def lda_projection(X, classes, means, priors, inv_cov):
    projections = []

    for x in X:
        scores = []
        for c in classes:
            mean = means[c]
            score = x.dot(inv_cov).dot(mean) - 0.5 * mean.dot(inv_cov).
 ↪dot(mean) + np.log(priors[c])
            scores.append(score)
        projections.append(np.max(scores))

    return np.array(projections)

def compute_accuracy(y_true, y_pred):
    return np.sum(y_true == y_pred) / len(y_true)

def compute_precision(y_true, y_pred, class_label):
    predicted_positives = np.sum(y_pred == class_label)
    return (np.sum((y_pred == class_label) & (y_true == class_label))) /␣
 ↪predicted_positives if predicted_positives > 0 else 0

def compute_recall(y_true, y_pred, class_label):
    true_positives = np.sum((y_pred == class_label) & (y_true == class_label))
    actual_positives = np.sum(y_true == class_label)
    return true_positives / actual_positives if actual_positives > 0 else 0

def compute_f1_score(y_true, y_pred, class_label):
    precision = compute_precision(y_true, y_pred, class_label)
    recall = compute_recall(y_true, y_pred, class_label)
    return 2 * (precision * recall) / (precision + recall) if (precision +␣
 ↪recall) > 0 else 0


classes, means, priors, cov, inv_cov = fit_lda(x_train_pca, y_train)

y_pred_lda = predict_lda(x_test_pca, classes, means, priors, inv_cov)

lda_accuracy = compute_accuracy(y_test, y_pred_lda)
```

QDA Functions

[100]:
```
def compute_means(X, y):
    classes = np.unique(y)
    means = {c: np.mean(X[y == c], axis=0) for c in classes}
    return means

def compute_priors(X, y):
    classes = np.unique(y)
    priors = {c: np.sum(y == c) / X.shape[0] for c in classes}
    return priors

def compute_covariances(X, y, reg_param):
    classes = np.unique(y)
    n_features = X.shape[1]
    covariances = {}
    inv_covariances = {}
    log_det_cov = {}

    for c in classes:
        Xc = X[y == c]
        cov_c = np.cov(Xc, rowvar=False, bias=True) + reg_param * np.
 ↪eye(n_features)
        covariances[c] = cov_c
        inv_covariances[c] = np.linalg.inv(cov_c)
        log_det_cov[c] = np.log(np.linalg.det(cov_c))

    return covariances, inv_covariances, log_det_cov

def fit_qda(X, y, reg_param=0.1):
    means = compute_means(X, y)
    priors = compute_priors(X, y)
    covariances, inv_covariances, log_det_cov = compute_covariances(X, y,␣
 ↪reg_param)
    return means, priors, covariances, inv_covariances, log_det_cov

def predict_qda(X, means, priors, inv_covariances, log_det_cov, classes):
    preds = []
    for x in X:
        scores = []
        for c in classes:
            mean = means[c]
            inv_cov = inv_covariances[c]
            log_det = log_det_cov[c]
            prior = priors[c]
            diff = x - mean
            score = -0.5 * log_det - 0.5 * diff.dot(inv_cov).dot(diff) + np.
 ↪log(prior)
```

```
            scores.append(score)
        preds.append(classes[np.argmax(scores)])
    return np.array(preds)

# Manual implementation of evaluation metrics
def compute_accuracy(y_true, y_pred):
    return np.sum(y_true == y_pred) / len(y_true)


# Train QDA
means, priors, covariances, inv_covariances, log_det_cov = fit_qda(x_train_pca,␣
 ↪y_train, reg_param=0.1)

# Predict
y_pred_qda = predict_qda(x_test_pca, means, priors, inv_covariances,␣
 ↪log_det_cov, np.unique(y_train))

# Compute and print metrics manually
qda_accuracy = compute_accuracy(y_test, y_pred_qda)
```

FDA Functions

[101]:
```
def compute_scatter_matrices(X, y, reg_param=1e-6):

    classes = np.unique(y)
    n_features = X.shape[1]
    mu_overall = np.mean(X, axis=0)

    S_B = np.zeros((n_features, n_features))
    S_W = np.zeros((n_features, n_features))

    for c in classes:
        Xc = X[y == c]
        N_c = Xc.shape[0]
        mu_c = np.mean(Xc, axis=0)
        diff_mu = (mu_c - mu_overall).reshape(-1, 1)
        S_B += N_c * diff_mu.dot(diff_mu.T)

        diff = Xc - mu_c
        S_W += diff.T.dot(diff)

    # Regularization for numerical stability
    S_W += reg_param * np.eye(n_features)

    return S_B, S_W, classes

def solve_generalized_eigen(S_B, S_W):
```

```
    inv_SW = np.linalg.inv(S_W)
    mat = inv_SW.dot(S_B)
    eigenvalues, eigenvectors = np.linalg.eig(mat)
    return eigenvalues.real, eigenvectors.real

def sort_eigen_components(eigenvalues, eigenvectors):

    idx = np.argsort(eigenvalues)[::-1]
    eigenvalues_sorted = eigenvalues[idx]
    eigenvectors_sorted = eigenvectors[:, idx]
    return eigenvalues_sorted, eigenvectors_sorted

def select_fda_components(eigenvectors, classes, n_components):

    max_components = len(classes) - 1
    if n_components is None or n_components > max_components:
        n_components = max_components
    W = eigenvectors[:, :n_components]
    return W, n_components

def fda_fit(X, y, n_components=None, reg_param=1e-6):

    S_B, S_W, classes = compute_scatter_matrices(X, y, reg_param)
    eigenvalues, eigenvectors = solve_generalized_eigen(S_B, S_W)
    eigenvalues, eigenvectors = sort_eigen_components(eigenvalues, eigenvectors)
    W, n_components = select_fda_components(eigenvectors, classes, n_components)

    model = {
        'W': W,
        'eigenvalues': eigenvalues,
        'classes': classes,
        'n_components': n_components,
        'reg_param': reg_param
    }
    return model

def fda_transform(X, fda_model):

    return np.dot(X, fda_model['W'])

def fda_fit_transform(X, y, n_components=None, reg_param=1e-6):

    model = fda_fit(X, y, n_components, reg_param)
    X_transformed = fda_transform(X, model)
    return X_transformed, model
```

```
def compute_centroids(X_fda, y):

    classes = np.unique(y)
    centroids = {}
    for c in classes:
        centroids[c] = np.mean(X_fda[y == c], axis=0)
    return centroids

def classify_sample(x, centroids):

    distances = {c: np.linalg.norm(x - centroid) for c, centroid in centroids.
 ↪items()}
    predicted_class = min(distances, key=distances.get)
    return predicted_class

def fda_classifier(X_train_fda, y_train, X_test_fda):
    centroids = compute_centroids(X_train_fda, y_train)
    predictions = [classify_sample(x, centroids) for x in X_test_fda]
    return np.array(predictions)


def compute_accuracy(y_true, y_pred):
    return np.sum(y_true == y_pred) / len(y_true)
```

[102]:
```
fda_model = fda_fit(x_train_pca, y_train)

# Transform both training and test data using the computed FDA projection matrix
x_train_fda = fda_transform(x_train_pca, fda_model)
x_test_fda = fda_transform(x_test_pca, fda_model)

# Classify the test data using a nearest centroid classifier in FDA space
y_pred_fda = fda_classifier(x_train_fda, y_train, x_test_fda)

# Compute evaluation metrics manually
fda_accuracy = compute_accuracy(y_test, y_pred_fda)
```

[ ]:
```

```

Printing results

[103]:
```
print(f"\nScratch LDA Accuracy: {lda_accuracy:.4f}")

print("Scratch LDA Classification Report:")
for c in classes:
    precision = compute_precision(y_test, y_pred_lda, c)
    recall = compute_recall(y_test, y_pred_lda, c)
```

```
    f1 = compute_f1_score(y_test, y_pred_lda, c)
    print(f"Class {c}: Precision={precision:.4f}, Recall={recall:.4f},␣
↪F1-Score={f1:.4f}")
```

```
Scratch LDA Accuracy: 0.9500
Scratch LDA Classification Report:
Class 0: Precision=0.9900, Recall=0.9900, F1-Score=0.9900
Class 1: Precision=0.8981, Recall=0.9700, F1-Score=0.9327
Class 2: Precision=0.9674, Recall=0.8900, F1-Score=0.9271
```

0.0.1 Overall Accuracy

The LDA model achieved an accuracy of **95%**, meaning it correctly classified 95 out of every 100 digits.

Model	Accuracy
Scratch LDA	0.9500

0.0.2 Detailed Performance (Classification Report)

Here's a breakdown of how well the model performed for each digit class:

Digit (Class)	Precision	Recall	F1-Score
0	99.00%	99.00%	99.00%
1	89.81%	97.00%	93.27%
2	96.74%	89.00%	92.71%

What These Numbers Mean:

- **Precision**: Measures how many of the model's predictions for a digit were actually correct.
  - For example, the model was **89.81% precise** for the digit **1**, meaning that when it predicted a "1," it was correct about **89.81% of the time**.
- **Recall**: Measures how many actual instances of a digit the model correctly identified.
  - The recall for **1** is **97%**, meaning that out of all the actual "1"s in the dataset, the model correctly found **97%** of them.
- **F1-Score**: A balanced combination of precision and recall. The closer to **1 (or 100%)**, the better.

Observations:

- The model performed **exceptionally well** on class **0**, with nearly perfect scores.
- For class **1**, the recall is very high (**97%**), meaning the model is good at detecting "1"s, but the lower precision (**89.81%**) suggests some other digits might be misclassified as "1".
- Class **2** has **great precision (96.74%)** but a slightly lower recall (**89%**), meaning it sometimes misses actual "2"s.

Final Thoughts: Overall, the model is **strong and reliable**, with room for improvement in classifying certain digits more accurately. A possible next step could be fine-tuning the model to reduce misclassifications.

[104]:
```
print(f"\nScratch QDA Accuracy: {qda_accuracy:.4f}")

print("Scratch QDA Classification Report:")
for c in np.unique(y_train):
    precision = compute_precision(y_test, y_pred_qda, c)
    recall = compute_recall(y_test, y_pred_qda, c)
    f1 = compute_f1_score(y_test, y_pred_qda, c)
    print(f"Class {c}: Precision={precision:.4f}, Recall={recall:.4f},␣
 ↪F1-Score={f1:.4f}")
```

```
Scratch QDA Accuracy: 0.9933
Scratch QDA Classification Report:
Class 0: Precision=1.0000, Recall=0.9900, F1-Score=0.9950
Class 1: Precision=0.9900, Recall=0.9900, F1-Score=0.9900
Class 2: Precision=0.9901, Recall=1.0000, F1-Score=0.9950
```

0.0.3 MNIST Dataset - QDA Model Performance

0.0.4 Overall Accuracy

The QDA model achieved an **amazing accuracy of 99.33%**! This means that out of every 1,000 digits, it correctly classified **993 of them**.

Model	Accuracy
Scratch QDA	0.9933

0.0.5 Detailed Performance (Classification Report)

Here's a breakdown of how well the model performed for each digit class:

Digit (Class)	Precision	Recall	F1-Score
0	100.00%	99.00%	99.50%
1	99.00%	99.00%	99.00%
2	99.01%	100.00%	99.50%

What These Numbers Mean:

- **Precision**: How often the model's predictions for a digit were correct.
  - Class **0** has **100% precision**, meaning every time the model predicted a "0," it was **always right**.
- **Recall**: How many actual instances of a digit were correctly identified.
  - Class **2** has **100% recall**, meaning the model **never missed a "2"** in the dataset.

- **F1-Score**: A balanced measure of precision and recall. Closer to **1 (or 100%)**, the better.

Observations:

- The QDA model is **extremely accurate**, nearly perfect across all digit classes.
- It **never misclassified a "0" as another digit** (precision = 100%), though it did miss a very small number of actual "0"s (recall = 99%).
- For class **1**, both precision and recall are **99%**, showing strong consistency in classification.
- Class **2** has a **perfect recall score (100%)**, meaning **every actual "2" in the dataset was correctly identified**.

Final Thoughts: This QDA model performs **exceptionally well**, reaching almost **perfect classification** across the board! The small margin of error suggests it's already highly optimized, but slight improvements could be made in ensuring **recall for class 0** reaches 100%. Overall, a **fantastic model for digit recognition!**

[105]:
```
print(f"FDA Accuracy: {fda_accuracy:.4f}\n")

print("FDA Classification Report:")
for c in np.unique(y_train):
    precision = compute_precision(y_test, y_pred_fda, c)
    recall = compute_recall(y_test, y_pred_fda, c)
    f1 = compute_f1_score(y_test, y_pred_fda, c)
    print(f"Class {c}: Precision={precision:.4f}, Recall={recall:.4f},␣
 ↪F1-Score={f1:.4f}")
```

```
FDA Accuracy: 0.9467

FDA Classification Report:
Class 0: Precision=0.9900, Recall=0.9900, F1-Score=0.9900
Class 1: Precision=0.8899, Recall=0.9700, F1-Score=0.9282
Class 2: Precision=0.9670, Recall=0.8800, F1-Score=0.9215
```

0.0.6 MNIST Dataset - FDA Model Performance

0.0.7 Overall Accuracy

The FDA model achieved an overall accuracy of **94.67%**, meaning that it correctly classified roughly 95 out of every 100 digits.

Model	Accuracy
FDA	0.9467

0.0.8 Detailed Performance (Classification Report)

Below is a breakdown of how well the model performed for each digit class:

Digit (Class)	Precision	Recall	F1-Score
0	99.00%	99.00%	99.00%
1	88.99%	97.00%	92.82%
2	96.70%	88.00%	92.15%

What These Numbers Mean:

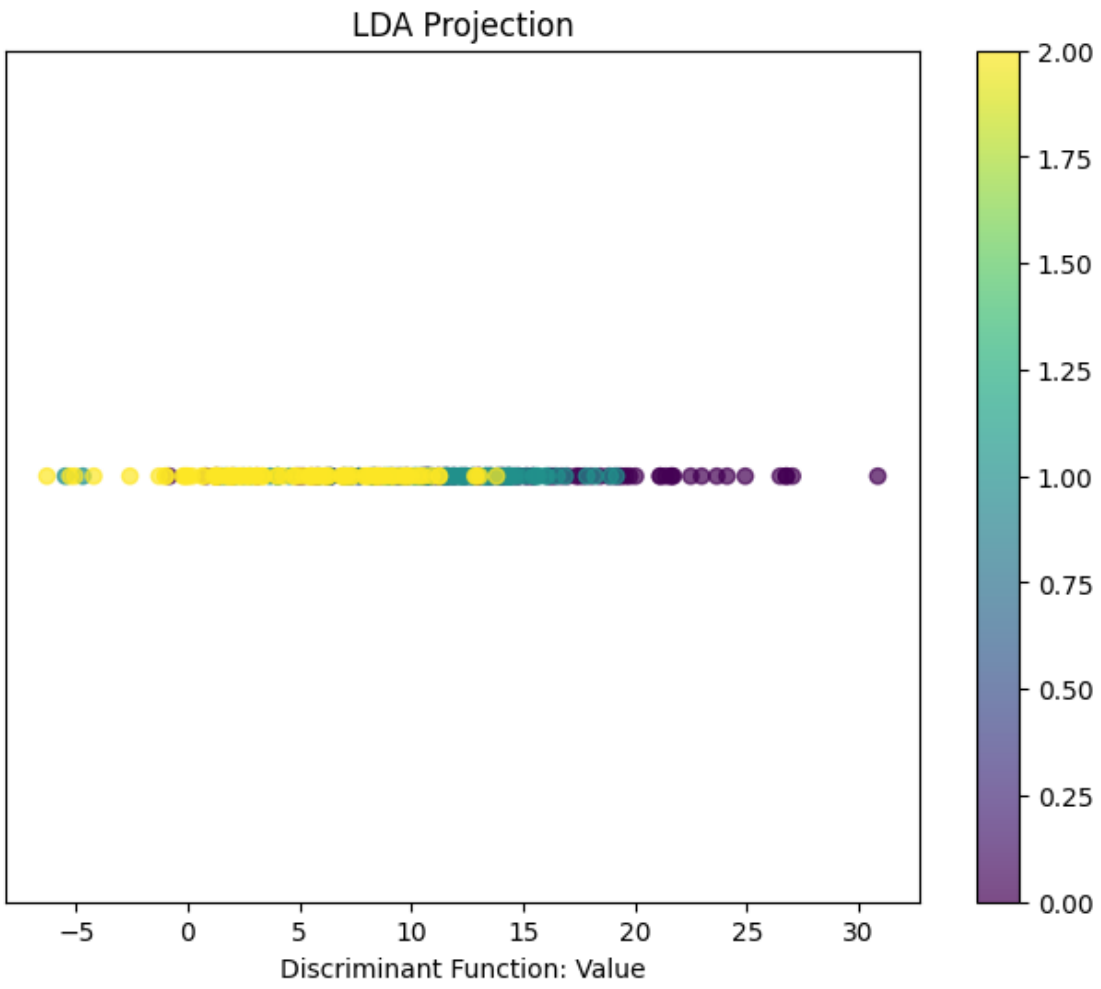
- **Precision**: Indicates how often the model's predictions for a digit were correct. For instance, for class **0**, the model has a precision of **99%**, meaning nearly every time it predicted a "0," it was right.
- **Recall**: Measures how many actual instances of a digit were correctly identified by the model. For class **1**, a recall of **97%** means it successfully identified 97% of all "1"s.
- **F1-Score**: A combined metric that balances precision and recall. A higher F1-score (closer to 100%) shows better performance.

Observations:

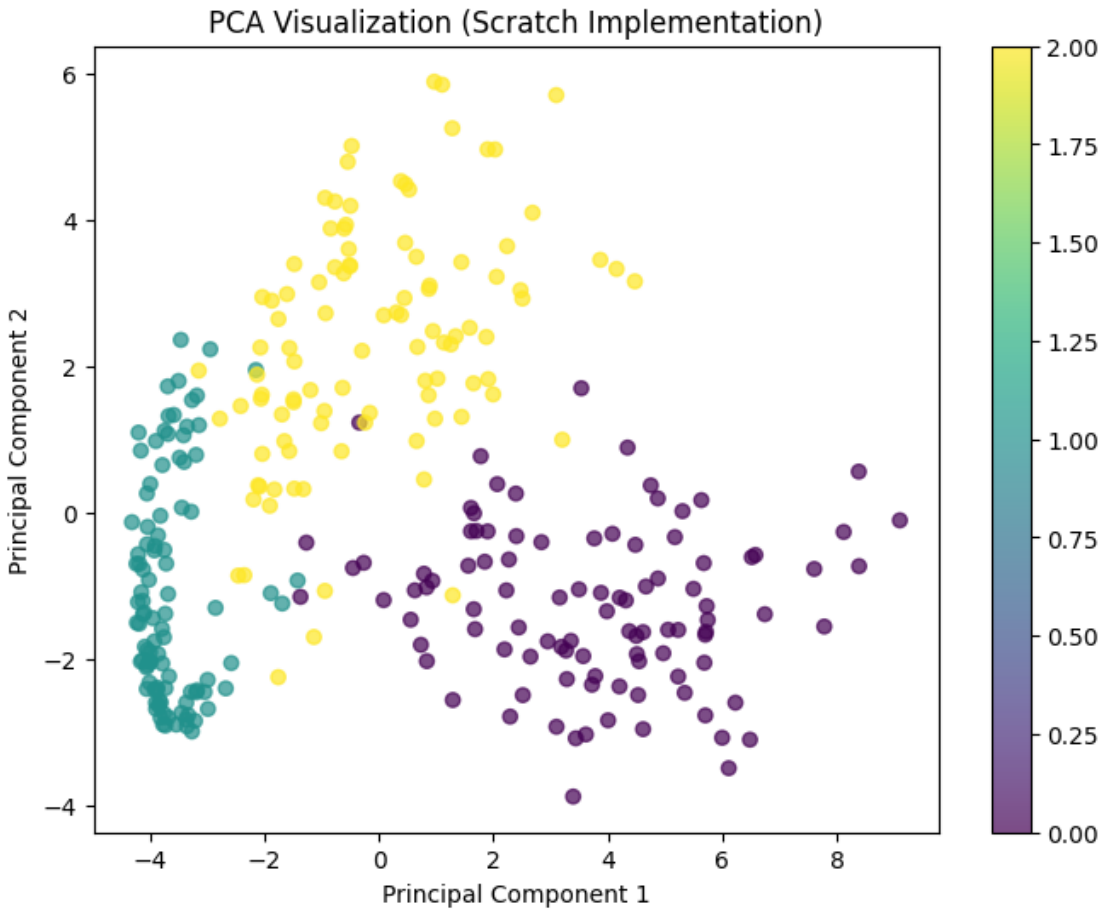
- **Class 0**: Shows excellent performance across all metrics.
- **Class 1**: While the recall is very high (97%), the precision is a bit lower (about 89%), which means some digits that aren't "1" might occasionally be misclassified as such.
- **Class 2**: Exhibits high precision (96.70%), but the recall is a bit lower (88%), suggesting that the model may miss some actual "2"s.

Final Thoughts: Overall, the FDA model performs very well, with a solid accuracy and strong class-specific metrics. While there's a slight room for improvement in balancing precision and recall for classes 1 and 2, the model is robust and reliable for digit recognition.

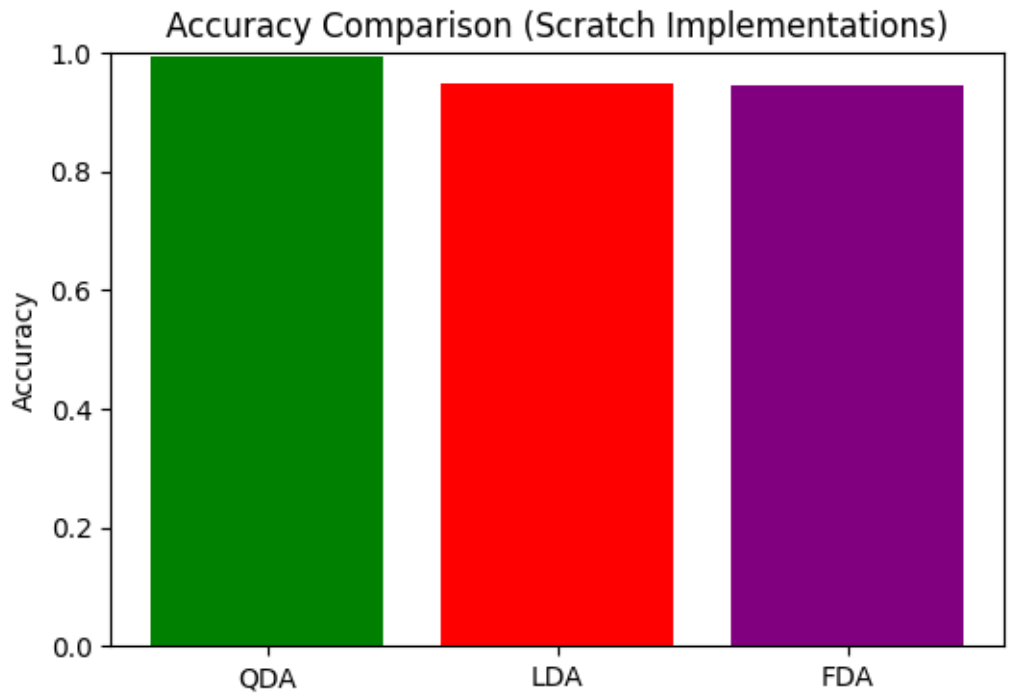
[106]:
```
lda_proj = lda_projection(x_train_pca, classes, means, priors, inv_cov)


plt.figure(figsize=(8, 6))
plt.scatter(lda_proj, np.zeros_like(lda_proj), c=y_train, cmap='viridis',␣
 ↪alpha=0.7)
plt.title("LDA Projection")
plt.xlabel("Discriminant Function: Value")
plt.yticks([])
plt.colorbar()
plt.show()
```



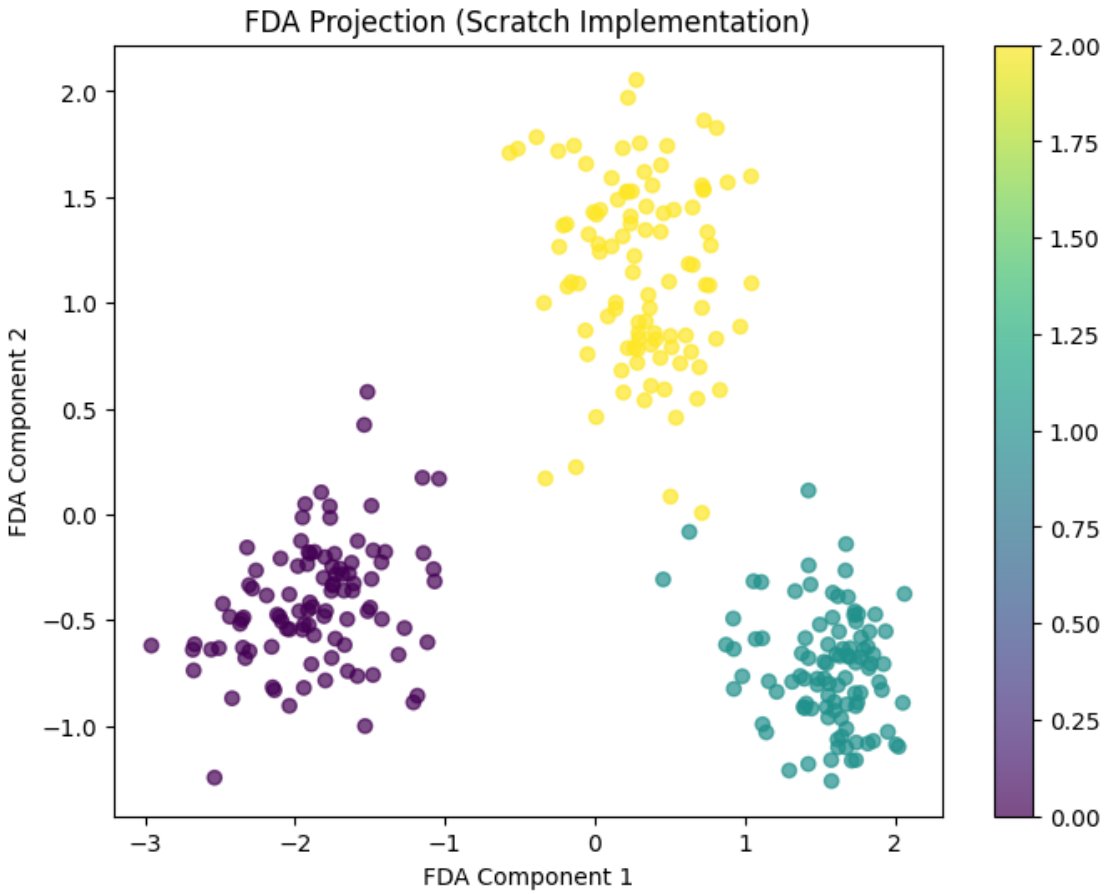
[107]:
```
plt.figure(figsize=(8, 6))
plt.scatter(x_train_pca[:, 0], x_train_pca[:, 1], c=y_train, cmap='viridis',␣
↪alpha=0.7)
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("PCA Visualization (Scratch Implementation)")
plt.colorbar()
plt.show()
```



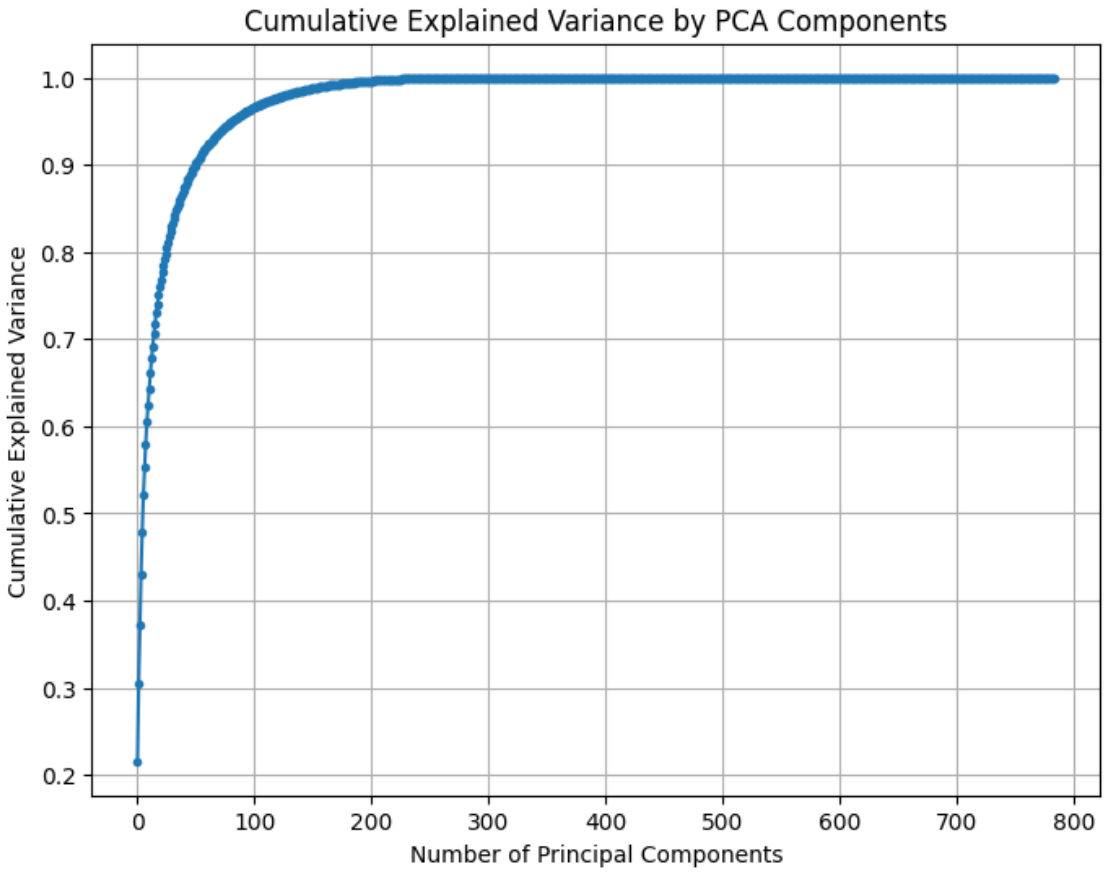
[108]:
```
plt.figure(figsize=(6, 4))
plt.bar(['QDA' , 'LDA','FDA'], [qda_accuracy, lda_accuracy, fda_accuracy],␣
↪color=['green', 'red', 'purple'])
plt.ylabel("Accuracy")
plt.title("Accuracy Comparison (Scratch Implementations)")
plt.ylim(0, 1)
plt.show()
```



[109]:
```
if fda_model['n_components'] >= 2:
    plt.figure(figsize=(8, 6))
    plt.scatter(x_train_fda[:, 0], x_train_fda[:, 1], c=y_train,␣
 ↪cmap='viridis', alpha=0.7)
    plt.title("FDA Projection (Scratch Implementation)")
    plt.xlabel("FDA Component 1")
    plt.ylabel("FDA Component 2")
    plt.colorbar()
    plt.show()
```



[110]:
```
cumulative_variance = np.cumsum(eigenvalues) / np.sum(eigenvalues)
plt.figure(figsize=(8, 6))
plt.plot(cumulative_variance, marker='.')
plt.title("Cumulative Explained Variance by PCA Components")
plt.xlabel("Number of Principal Components")
plt.ylabel("Cumulative Explained Variance")
plt.grid()
plt.show()
```



With 90 percent variance

[111]:
```
mean, components, eigenvalues, n_components = fit_pca(x_train, variance_ratio=0.
↪95)
```

[112]:
```
x_test_pca = transform_pca(x_test, mean, components, n_components)
x_train_pca = transform_pca(x_train, mean, components, n_components)
```

[113]:
```
print(f"Reduced dimension-> {n_components}")
print(f"Original dimension-> {x_train.shape[1]}")
```

```
Reduced dimension-> 82
Original dimension-> 784
```

[114]:
```
classes, means, priors, cov, inv_cov = fit_lda(x_train_pca, y_train)

y_pred_lda = predict_lda(x_test_pca, classes, means, priors, inv_cov)

lda_accuracy = compute_accuracy(y_test, y_pred_lda)
```

[115]:
```
means, priors, covariances, inv_covariances, log_det_cov = fit_qda(x_train_pca,␣
 ↪y_train, reg_param=0.1)

y_pred_qda = predict_qda(x_test_pca, means, priors, inv_covariances,␣
 ↪log_det_cov, np.unique(y_train))

qda_accuracy = compute_accuracy(y_test, y_pred_qda)
```

[116]:
```
fda_model = fda_fit(x_train_pca, y_train)


x_train_fda = fda_transform(x_train_pca, fda_model)
x_test_fda = fda_transform(x_test_pca, fda_model)

y_pred_fda = fda_classifier(x_train_fda, y_train, x_test_fda)

fda_accuracy = compute_accuracy(y_test, y_pred_fda)
```

[117]:
```
print(f"\nScratch LDA Accuracy: {lda_accuracy:.4f}")

print("Scratch LDA Classification Report:")
for c in classes:
    precision = compute_precision(y_test, y_pred_lda, c)
    recall = compute_recall(y_test, y_pred_lda, c)
    f1 = compute_f1_score(y_test, y_pred_lda, c)
    print(f"Class {c}: Precision={precision:.4f}, Recall={recall:.4f},␣
 ↪F1-Score={f1:.4f}")
```

```
Scratch LDA Accuracy: 0.9500
Scratch LDA Classification Report:
Class 0: Precision=0.9900, Recall=0.9900, F1-Score=0.9900
Class 1: Precision=0.8981, Recall=0.9700, F1-Score=0.9327
Class 2: Precision=0.9674, Recall=0.8900, F1-Score=0.9271
```

[118]:
```
print(f"\nScratch QDA Accuracy: {qda_accuracy:.4f}")

print("Scratch QDA Classification Report:")
for c in np.unique(y_train):
    precision = compute_precision(y_test, y_pred_qda, c)
    recall = compute_recall(y_test, y_pred_qda, c)
    f1 = compute_f1_score(y_test, y_pred_qda, c)
    print(f"Class {c}: Precision={precision:.4f}, Recall={recall:.4f},␣
 ↪F1-Score={f1:.4f}")
```

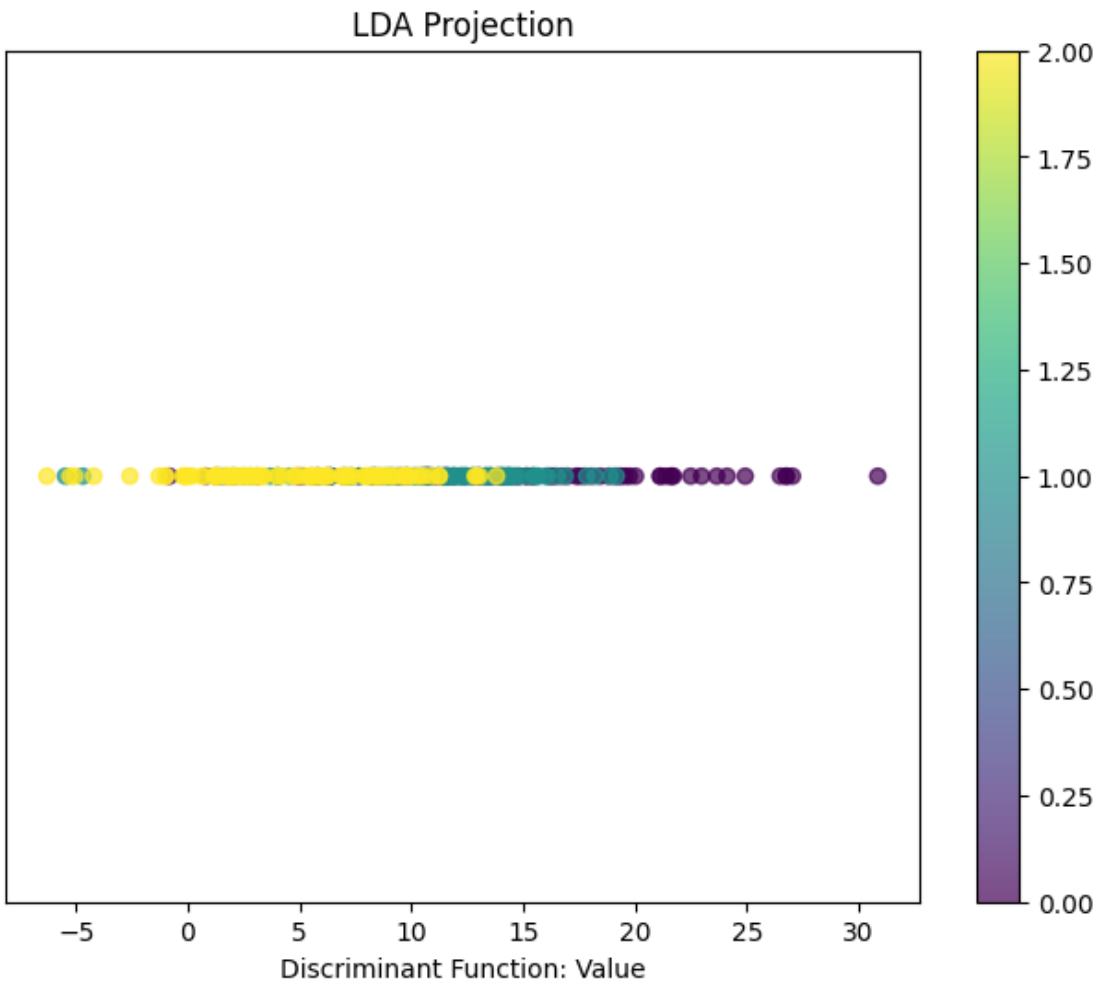
```
Scratch QDA Accuracy: 0.9933
```

```
Scratch QDA Classification Report:
Class 0: Precision=1.0000, Recall=0.9900, F1-Score=0.9950
Class 1: Precision=0.9900, Recall=0.9900, F1-Score=0.9900
Class 2: Precision=0.9901, Recall=1.0000, F1-Score=0.9950
```

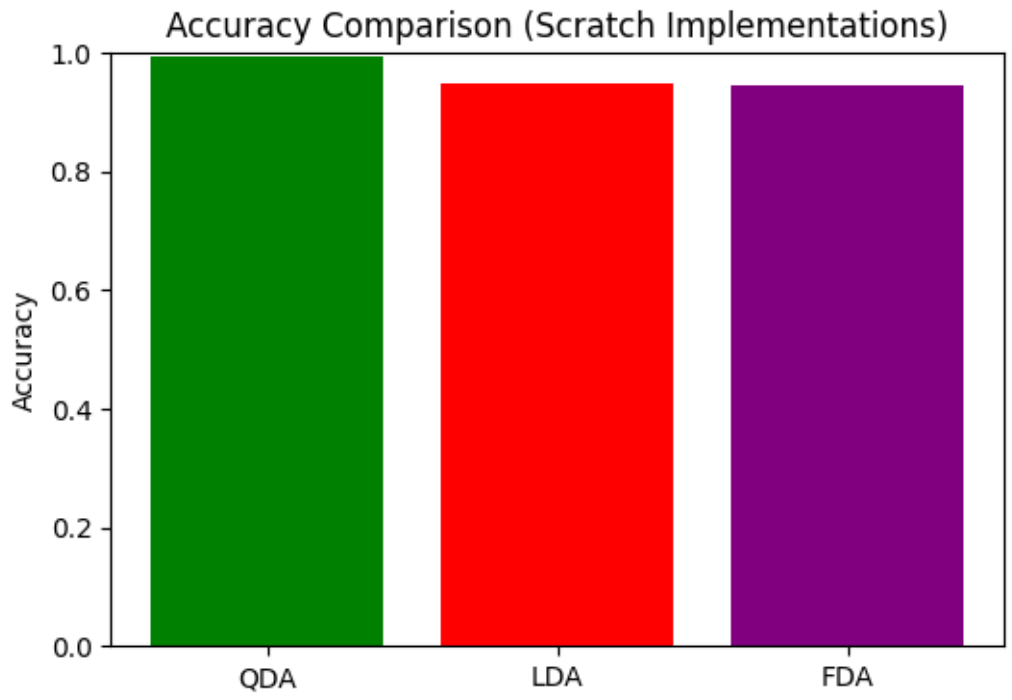
[119]:
```
print(f"FDA Accuracy: {fda_accuracy:.4f}\n")

print("FDA Classification Report:")
for c in np.unique(y_train):
    precision = compute_precision(y_test, y_pred_fda, c)
    recall = compute_recall(y_test, y_pred_fda, c)
    f1 = compute_f1_score(y_test, y_pred_fda, c)
    print(f"Class {c}: Precision={precision:.4f}, Recall={recall:.4f},␣
 ↪F1-Score={f1:.4f}")
```

```
FDA Accuracy: 0.9467

FDA Classification Report:
Class 0: Precision=0.9900, Recall=0.9900, F1-Score=0.9900
Class 1: Precision=0.8899, Recall=0.9700, F1-Score=0.9282
Class 2: Precision=0.9670, Recall=0.8800, F1-Score=0.9215
```

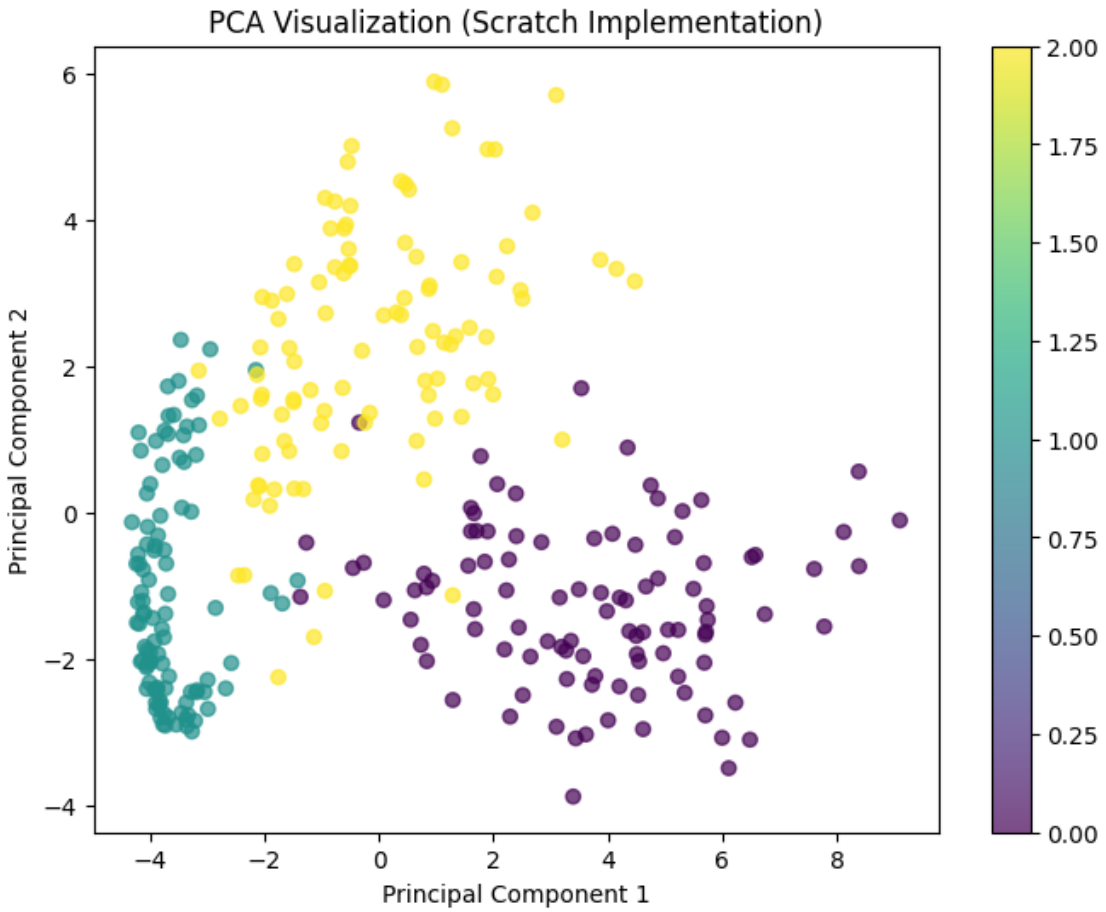
[120]:
```
lda_proj = lda_projection(x_train_pca, classes, means, priors, inv_cov)


plt.figure(figsize=(8, 6))
plt.scatter(lda_proj, np.zeros_like(lda_proj), c=y_train, cmap='viridis',␣
 ↪alpha=0.7)
plt.title("LDA Projection")
plt.xlabel("Discriminant Function: Value")
plt.yticks([])
plt.colorbar()
plt.show()
```



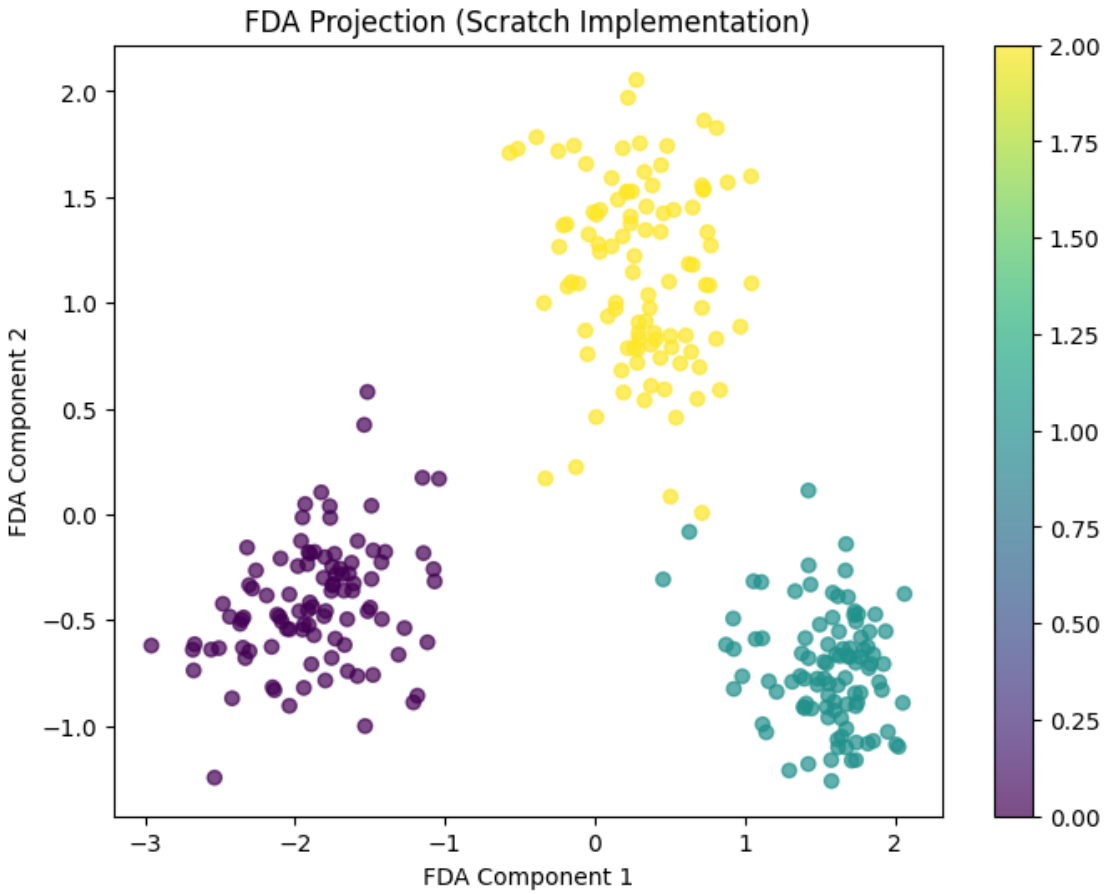
[121]:
```
plt.figure(figsize=(6, 4))
plt.bar(['QDA' , 'LDA','FDA'], [qda_accuracy, lda_accuracy, fda_accuracy],␣
↪color=['green', 'red', 'purple'])
plt.ylabel("Accuracy")
plt.title("Accuracy Comparison (Scratch Implementations)")
plt.ylim(0, 1)
plt.show()
```



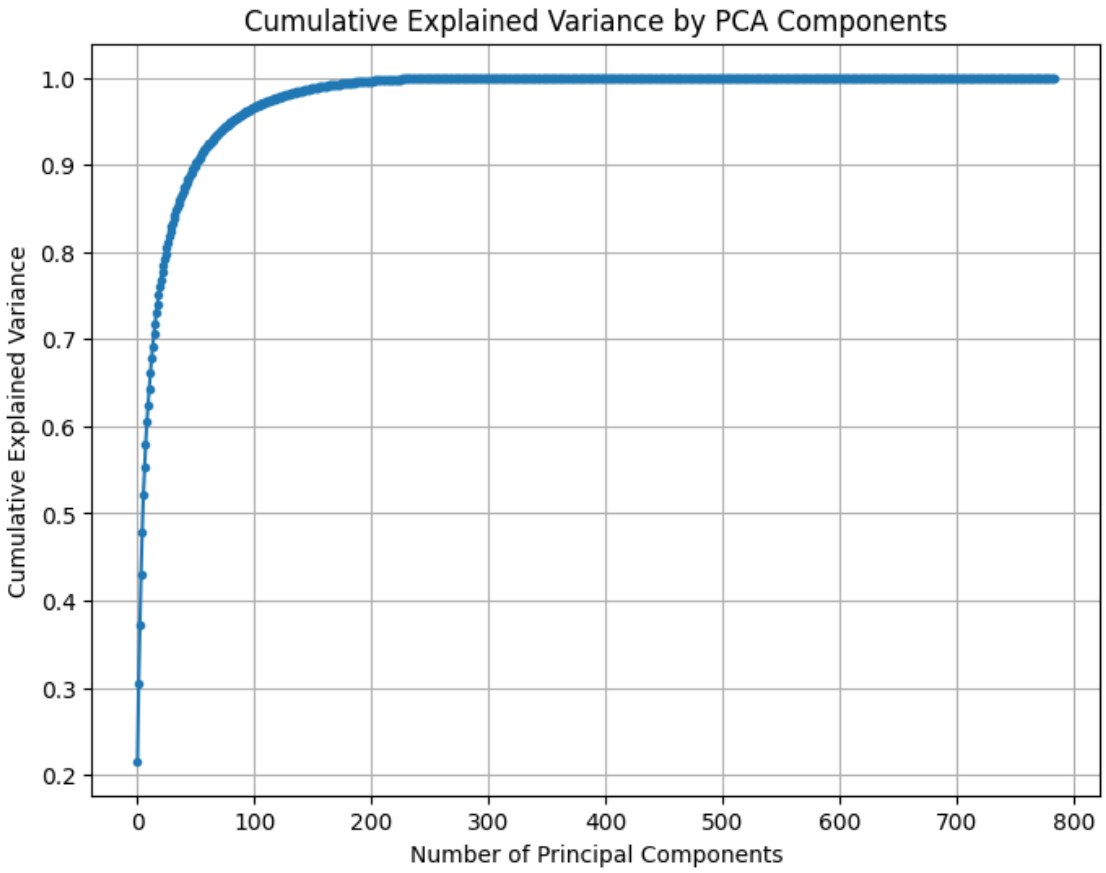
[122]:
```
plt.figure(figsize=(8, 6))
plt.scatter(x_train_pca[:, 0], x_train_pca[:, 1], c=y_train, cmap='viridis',␣
↪alpha=0.7)
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("PCA Visualization (Scratch Implementation)")
plt.colorbar()
plt.show()
```



[123]:
```
if fda_model['n_components'] >= 2:
    plt.figure(figsize=(8, 6))
    plt.scatter(x_train_fda[:, 0], x_train_fda[:, 1], c=y_train,␣
 ↪cmap='viridis', alpha=0.7)
    plt.title("FDA Projection (Scratch Implementation)")
    plt.xlabel("FDA Component 1")
    plt.ylabel("FDA Component 2")
    plt.colorbar()
    plt.show()
```



[124]:
```
cumulative_variance = np.cumsum(eigenvalues) / np.sum(eigenvalues)
plt.figure(figsize=(8, 6))
plt.plot(cumulative_variance, marker='.')
plt.title("Cumulative Explained Variance by PCA Components")
plt.xlabel("Number of Principal Components")
plt.ylabel("Cumulative Explained Variance")
plt.grid()
plt.show()
```


Cumulative Explained Variance by PCA Components
Cumulative Explained Variance
Number of Principal Components
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
100
200
300
400
500
600
700
800

Changing of variance doesn't have any effect on accuracy