mfound_hw3

October 28, 2024

[4]:
```
# imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import random

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.datasets import fetch_olivetti_faces
from sklearn.metrics import mean_squared_error
```

1 1. Eigenvalues and Eigenvectors

[5]:
```
# (4) Using np.linalg.eig, compute the eigenvalues and eigenvectors of A
# print your results [print()]
# TODO

# Define a matrix
matrix = np.array([[9, 1, -1], [-1, 11, 1], [-2, 2, 10]])

# Calculate eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(matrix)

print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)
```

```
Eigenvalues: [12.  8. 10.]
Eigenvectors:
 [[ 9.00258517e-16 -7.07106781e-01 -7.07106781e-01]
 [-7.07106781e-01 -3.36518470e-16 -7.07106781e-01]
 [-7.07106781e-01 -7.07106781e-01 -5.62025848e-15]]
```

2 2. Principal Component analysis and eigenfaces

[6]:
```
dataset = fetch_olivetti_faces() # Load the Olivetti faces dataset from sklearn.
 ↪datasets
```

[7]:
```
persons_faces = [] # Create a list to store the faces of each person
for i in range(40): # For each person in the dataset
    faces = [] # Create a list to store the faces of the person
    for j in range(10): # For each face of the person
        faces += [dataset.images[i*10+j]] # Add the face to the list
    persons_faces += [faces] # Add the list of faces to the list of persons

database = [faces[:-1] for faces in persons_faces[:36]] # Create a database of␣
 ↪faces for the first 36 persons
test_faces = [faces[-1] for faces in persons_faces[:36]] # Create a list of␣
 ↪faces for the first 36 persons to test the algorithm
people_not_in_database = [face for faces in persons_faces[36:] for face in␣
 ↪faces] # Create a list of faces for the last 4 persons to test the algorithm
```

[8]:
```
database_array = np.array(database)
print(database_array.shape)
```

```
(36, 9, 64, 64)
```

[9]:
```
# Display images in the persons_faces list
fig, axes = plt.subplots(40, 10, figsize=(15, 60))
for i in range(40):
    for j in range(10):
        axes[i, j].imshow(persons_faces[i][j], cmap='gray')
        axes[i, j].axis('off')

plt.show()
num_faces = sum(len(faces) for faces in persons_faces)
print("Number of faces in the persons_faces list:", num_faces)
```

```
Number of faces in the persons_faces list: 400
```

images in the test faces

[10]:
```python
num_of_test_faces = len(test_faces)
print("Number of images in the test_faces list:", num_of_test_faces)
```

```
Number of images in the test_faces list: 36
```

How many images are in the list of faces of people not in the database

[11]:
```python
num_not_in_database = len(people_not_in_database)
print("Number of images in the people_not_in_database list:",␣
↪num_not_in_database)
```

```
Number of images in the people_not_in_database list: 40
```

Image size

[12]:
```python
image_size = database[0][0].shape
print("Image size:", image_size)
```

```
Image size: (64, 64)
```

[13]:
```python
# display faces from the database of faces -> 9 faces of 5 different people
fig, axes = plt.subplots(5, 9, figsize=(10,5))
for i in range(5):
    for j in range(9):
        axes[i,j].imshow(database[i][j], cmap="gray")
        axes[i,j].set_xticks([]);
        axes[i,j].set_yticks([]);
```

[14]:
```
# display test faces for the first 5 people in the database
fig, axes = plt.subplots(1, 5, figsize=(10,5))
for i in range(5):
        axes[i].imshow(test_faces[i], cmap="gray")
        axes[i].set_xticks([]);
        axes[i].set_yticks([]);
```

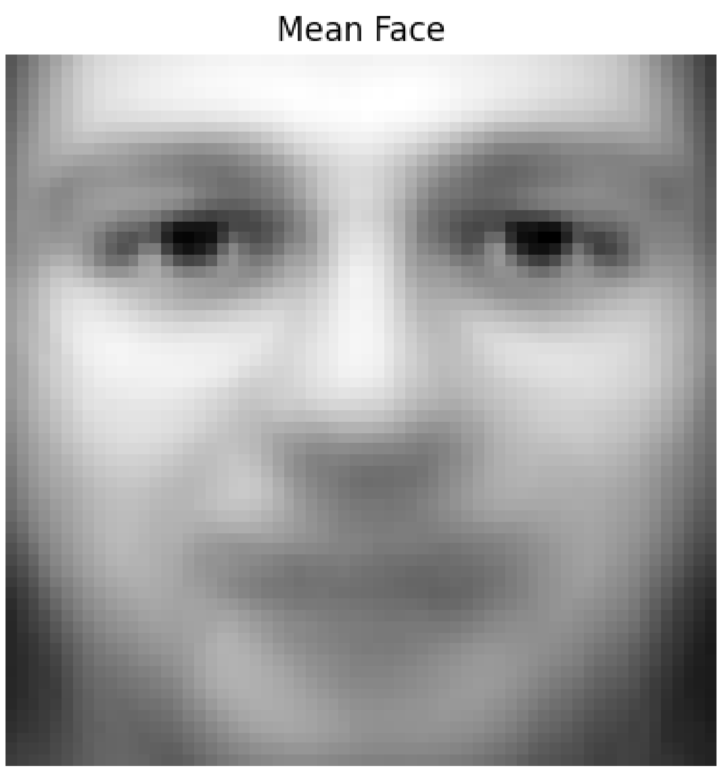
[15]:
```
# display faces of 10 people who are not in the database
fig, axes = plt.subplots(2, 5, figsize=(10,5))
for i in range(10):
        j = random.randint(0,39)
        axes[i//5 , i%5].imshow(people_not_in_database[j], cmap="gray")
        axes[i//5 , i%5].set_xticks([]);
        axes[i//5 , i%5].set_yticks([]);
```

[16]:
```
# Flatten each face in the database and stack them into a matrix T
T = np.array([face.flatten() for person in database for face in person])
print("Shape of matrix T:", T.shape)
```

```
Shape of matrix T: (324, 4096)
```

[17]:
```
# Calculate the mean face
mean_face = np.mean(T, axis=0) #  treats the entire matrix as a single list of␣
↪numbers
print(mean_face.shape)

# Reshape the mean face to the original image dimensions by using the␣
↪image_size variable
mean_face_image = mean_face.reshape(image_size)
print(mean_face_image.shape)

# Display the mean face
plt.imshow(mean_face_image, cmap='gray')
plt.title('Mean Face')
plt.axis('off')
plt.show()
```

```
(4096,)
(64, 64)
```



[18]:
```python
# Subtract the mean face from all faces in the database
T_centered = T - mean_face

# Print the shape of the centered matrix
print("Shape of centered matrix T_centered:", T_centered.shape)
```

```
Shape of centered matrix T_centered: (324, 4096)
```

[19]:
```python
# Calculate the covariance matrix
covariance_matrix = np.cov(T_centered, rowvar=False)

# Print the shape of the covariance matrix
print("Shape of covariance matrix:", covariance_matrix.shape)
covariance_matrix
```

```
Shape of covariance matrix: (4096, 4096)
```

[19]:
```
array([[ 0.03252182,  0.0319414 ,  0.02836081, …, -0.00844297,
         -0.00787676, -0.00575145],
        [ 0.0319414 ,  0.03530122,  0.03361429, …, -0.01164312,
         -0.01073186, -0.00812144],
        [ 0.02836081,  0.03361429,  0.03723381, …, -0.01503633,
         -0.01368845, -0.01087497],
        …,
        [-0.00844297, -0.01164312, -0.01503633, …,  0.03722309,
          0.03321925,  0.02946416],
        [-0.00787676, -0.01073186, -0.01368845, …,  0.03321925,
          0.03527732,  0.03254291],
        [-0.00575145, -0.00812144, -0.01087497, …,  0.02946416,
          0.03254291,  0.03369307]])
```

[20]:
```python
# Compute the eigenvalues and eigenvectors of the covariance matrix
eigenvalues, eigenfaces = np.linalg.eig(covariance_matrix)

# Print the shape of the eigenfaces matrix
print("Shape of eigenfaces matrix:", eigenfaces.shape)
eigenfaces
```

```
Shape of eigenfaces matrix: (4096, 4096)
```

[20]:
```
array([[-2.61405820e-03+0.j        ,  2.87979283e-02+0.j        ,
          8.82701496e-03+0.j        , …,  3.85201028e-03+0.j        ,
         -4.27036462e-03+0.00296833j, -4.27036462e-03-0.00296833j],
        [-4.87186210e-03+0.j        ,  3.33767727e-02+0.j        ,
          8.12279797e-03+0.j        , …,  9.79410839e-04+0.j        ,
         -4.18516744e-04+0.00028033j, -4.18516744e-04-0.00028033j],
        [-8.21832336e-03+0.j        ,  3.81229228e-02+0.j        ,
          5.88425338e-03+0.j        , …,  8.74926747e-05+0.j        ,
         -7.51970896e-04+0.00026422j, -7.51970896e-04-0.00026422j],
```

```
…,
[ 4.74637958e-03+0.j         , -3.22403362e-02+0.j         ,
 -1.62243004e-02+0.j         , …, -1.18559131e-02+0.j         ,
  5.51426139e-03+0.004242j   ,  5.51426139e-03-0.004242j  ],
[ 7.56200940e-03+0.j         , -2.86516780e-02+0.j         ,
 -1.44912150e-02+0.j         , …,  4.67621698e-03+0.j         ,
 -8.81317512e-03-0.00552752j, -8.81317512e-03+0.00552752j],
[ 6.82057418e-03+0.j         , -2.56531328e-02+0.j         ,
 -1.35746034e-02+0.j         , …, -1.04555157e-02+0.j         ,
 -8.08427361e-03-0.00460974j, -8.08427361e-03+0.00460974j]])
```

[21]:
```python
# Reshape the first eigenface to the original image dimensions
first_eigenface  = eigenfaces[:, 0].real.reshape(image_size)

# Display the first eigenface
plt.imshow(first_eigenface, cmap='gray')
plt.title('First Eigenface')
plt.axis('off')
plt.show()
```

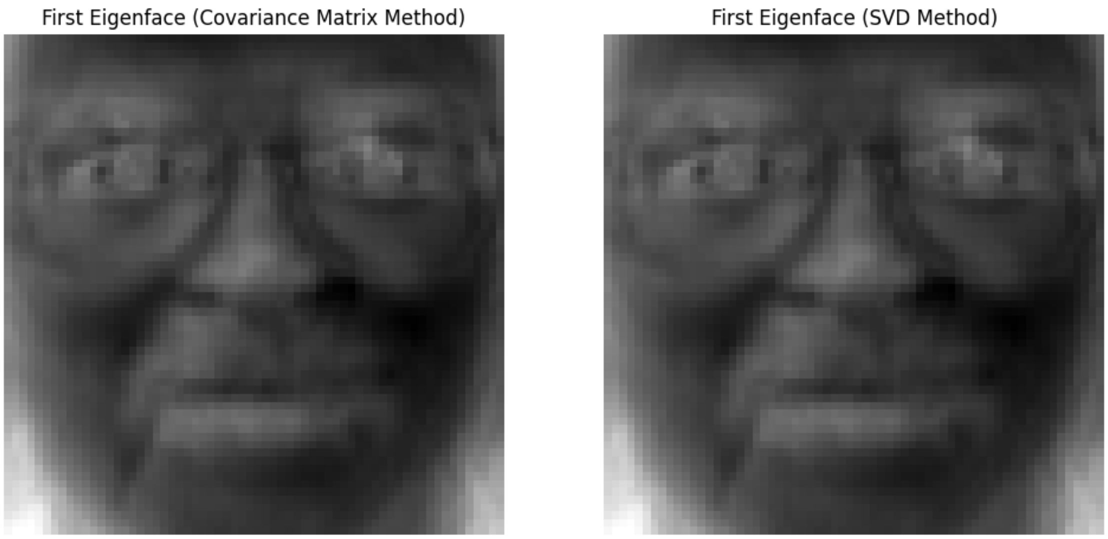
using M transpose U

[22]:
```python
M_transpose = np.transpose(T_centered)
# U is the eigen vectors of the matrix M * M_transpose
M_M_transpose = np.dot(T_centered, M_transpose)
eigenvalues_MMT, eigenvectors_MMT = np.linalg.eig(M_M_transpose)

# Sort the eigenvectors by decreasing eigenvalues
sorted_indices = np.argsort(eigenvalues_MMT)[::-1]
eigenvalues_MMT = eigenvalues_MMT[sorted_indices]
eigenvectors_MMT = eigenvectors_MMT[:, sorted_indices]

# Calculate the eigenfaces V = M_transpose * eigenvectors
eigenfaces_MMT = np.dot(M_transpose, eigenvectors_MMT)
eigenfaces_MMT
```

[22]:
```
array([[-1.9553629e-01, -1.7310152e+00, -3.8559711e-01, …,
         7.7687297e-03,  4.6728700e-03,  5.9604645e-07],
       [-3.6442426e-01, -2.0062451e+00, -3.5483426e-01, …,
        -2.1124983e-02,  1.6179064e-02,  1.1473894e-06],
       [-6.1474574e-01, -2.2915316e+00, -2.5704622e-01, …,
        -3.6837989e-03,  4.1825473e-03,  3.1292439e-07],
       …,
       [ 3.5503766e-01,  1.9379349e+00,  7.0873821e-01, …,
        -3.1124614e-03,  8.5842889e-03,  0.0000000e+00],
       [ 5.6565231e-01,  1.7222242e+00,  6.3303053e-01, …,
         4.2549167e-02,  7.5801648e-03,  1.8477440e-06],
       [ 5.1019144e-01,  1.5419848e+00,  5.9298956e-01, …,
         7.3327430e-02,  2.4369657e-03, -3.8743019e-07]], dtype=float32)
```

[23]:
```python
# Reshape the first eigenface obtained from the covariance matrix method
first_eigenface_cov = eigenfaces[:, 0].real.reshape(image_size)

# Reshape the first eigenface obtained from the SVD method
first_eigenface_svd = eigenfaces_MMT[:, 0].reshape(image_size)

# Plot the eigenfaces
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

# Display the first eigenface from the covariance matrix method
axes[0].imshow(first_eigenface_cov, cmap='gray')
axes[0].set_title('First Eigenface (Covariance Matrix Method)')
axes[0].axis('off')

# Display the first eigenface from the SVD method
axes[1].imshow(first_eigenface_svd, cmap='gray')
axes[1].set_title('First Eigenface (SVD Method)')
axes[1].axis('off')
```

```python
plt.show()
```



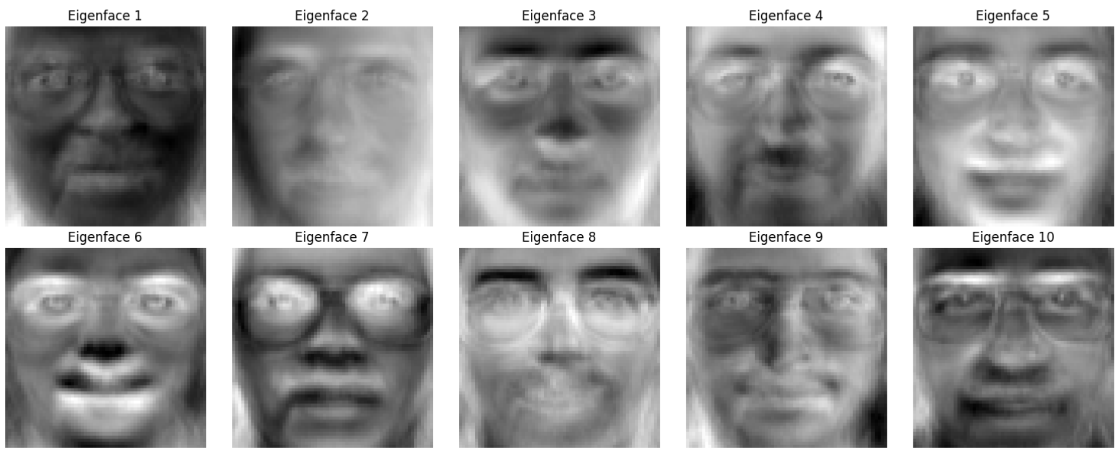
[24]:
```python
# Perform Singular Value Decomposition (SVD) on the centered matrix
U, S, Vt = np.linalg.svd(T_centered, full_matrices=False)

# Calculate the cumulative variance explained by the singular values
cumulative_variance = np.cumsum(S**2) / np.sum(S**2)

# Find the number of eigenfaces needed to capture at least 90% of the variance
num_eigenfaces = np.searchsorted(cumulative_variance, 0.90) + 1

print(f"Number of eigenfaces needed to capture at least 90% variance:␣
↪{num_eigenfaces}")

# Select the top eigenfaces
top_eigenfaces = eigenfaces_MMT[:num_eigenfaces]

print(f"Shape of top eigenfaces: {top_eigenfaces.shape}")
```

```
Number of eigenfaces needed to capture at least 90% variance: 62
Shape of top eigenfaces: (62, 324)
```

13. Visualizing the top 10 eigen faces

[25]:
```python
# Reshape the top 10 eigenfaces to the original image dimensions
top_10_eigenfaces = [eigenfaces_MMT[:, i].real.reshape(image_size) for i in␣
↪range(10)]
```

```python
# Plot the top 10 eigenfaces
fig, axes = plt.subplots(2, 5, figsize=(15, 6))
for i, ax in enumerate(axes.flat):
    ax.imshow(top_10_eigenfaces[i], cmap='gray')
    ax.set_title(f'Eigenface {i+1}')
    ax.axis('off')

plt.tight_layout()
plt.show()
```



14. Given a matrix, A, whose columns are the top eigenfaces selected in (12), show that the projection matrix unto the column space of A is the identity matrix

[26]:
```python
def projection_matrix(A, num_components):
    # Compute the projection matrix P
    P = A[:, :num_components] @ np.linalg.inv(A[:, :num_components].T @ A[:, :
↪num_components]) @ A[:, :num_components].T
    return P

print(num_eigenfaces)

P = projection_matrix(eigenfaces_MMT, num_eigenfaces)

# Print the shape of the projection matrix
print("Shape of projection matrix P:", P.shape)

# Verify that the projection matrix is close to the identity matrix
identity_matrix = np.eye(P.shape[0])
is_identity = np.allclose(P, identity_matrix, atol=1e-6)

print("Is the projection matrix close to the identity matrix?", is_identity)
```

```
62
Shape of projection matrix P: (4096, 4096)
Is the projection matrix close to the identity matrix? False
```

[27]:
```python
# Project the images in the database to the eigen subspace using the projection
↪matrix P
projected_images = np.dot(P, T_centered.T).T

# Print the shape of the projected images
print("Shape of projected images:", projected_images.shape)
```

```
Shape of projected images: (324, 4096)
```

Given a new face, x, and the mean image ¯x, show that the coordinate of x in the eigenfaces subspace is given by AT (x − ¯x)

[28]:
```python
def project_new_face(x, mean_face, A):
    # Center the new face by subtracting the mean face
    x_centered = x.flatten() - mean_face

    # Project the centered face onto the eigenfaces subspace
    coordinates = np.dot(A.T, x_centered)

    return coordinates

# Example usage
new_face = test_faces[0]  # Assuming test_faces contains the new faces
coordinates = project_new_face(new_face, mean_face, P)

print("shape of the Coordinates of the new face in the eigenfaces subspace:",
↪coordinates.shape)
```

```
shape of the Coordinates of the new face in the eigenfaces subspace: (4096,)
```

[29]:
```python
def recognize_new_face(new_face_coordinates, database_coordinates, threshold):
    # Calculate the distances between the new face and all faces in the database
    distances = np.linalg.norm(database_coordinates - new_face_coordinates,
↪axis=0)

    # Find the minimum distance
    min_distance = np.min(distances)

    # Determine if the new face is recognized
    is_recognized = min_distance < threshold

    return is_recognized, min_distance

# Example usage
new_face = test_faces[0]  # Assuming test_faces contains the new faces
```

```
print("Shape of the new face:", new_face.shape)
new_face_coordinates = project_new_face(new_face, mean_face, P)

threshold = 0.5  # Define your threshold value
is_recognized, min_distance = recognize_new_face(new_face_coordinates,␣
↪coordinates, threshold)

print(f"Is the new face recognized? {'Yes' if is_recognized else 'No'}")
print(f"Minimum distance: {min_distance}")
```

```
Shape of the new face: (64, 64)
Is the new face recognized? Yes
Minimum distance: 0.0
```

[30]:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score

# Define a function to evaluate the accuracy of the face recognition system
def evaluate_system(test_faces, mean_face, top_eigenfaces, coordinates,␣
↪threshold):
    predictions = []
    for test_face in test_faces:
        projected_face_coord = project_new_face(test_face, mean_face, P)
        is_recognized, min_distance = recognize_new_face(projected_face_coord,␣
↪coordinates, threshold)
        predictions.append(is_recognized)

    # Assuming the ground truth is that all test faces should be recognized
    ground_truth = [True] * len(test_faces)
    accuracy = accuracy_score(ground_truth, predictions)
    precision = precision_score(ground_truth, predictions)
    recall = recall_score(ground_truth, predictions)
    f1 = f1_score(ground_truth, predictions)
    return accuracy, precision, recall, f1

# Find the optimal threshold
thresholds = np.linspace(0, 100, 1000)
best_threshold = 0
best_accuracy = 0
for threshold in thresholds:
    accuracy, _, _, _ = evaluate_system(test_faces, mean_face, top_eigenfaces.
↪T, coordinates, threshold)
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_threshold = threshold
    if accuracy >= 0.80:
        break
```

```
print(f"Best threshold: {best_threshold}")
print(f"Accuracy: {best_accuracy}")
```

```
/home/kip/projects/MathsRequiredForAI/EnvMaths/lib/python3.12/site-
packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning:
Precision is ill-defined and being set to 0.0 due to no predicted samples. Use
`zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

```
Best threshold: 14.214214214214214
Accuracy: 0.8055555555555556
```

17. For the obtained threshold, Evaluate the face recognition system's performance using the following metrics

(a) accuracy
(b) Precision
(c) Recall
(d) F1-score

Conduct experiments to assess the impact of different parameters and settings on your face recognition system. Consider factors like the number of eigenfaces.

Analyze and interpret the results, discussing the strengths and limitations of the Eigenfaces method in your context.

[31]:
```
threshold = best_threshold
accuracy, precision, recall, f1 = evaluate_system(test_faces, mean_face,
 ↪top_eigenfaces, coordinates, threshold)
print("Evaluation Metrics:")
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")
```

```
Evaluation Metrics:
Accuracy: 0.8055555555555556
Precision: 1.0
Recall: 0.8055555555555556
F1 Score: 0.8923076923076924
```

[32]:
```
accuracy, precision, recall, f1 = evaluate_system(people_not_in_database,
 ↪mean_face, top_eigenfaces, coordinates, threshold)
print("Evaluation Metrics for People Not in the Database:")
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")
```

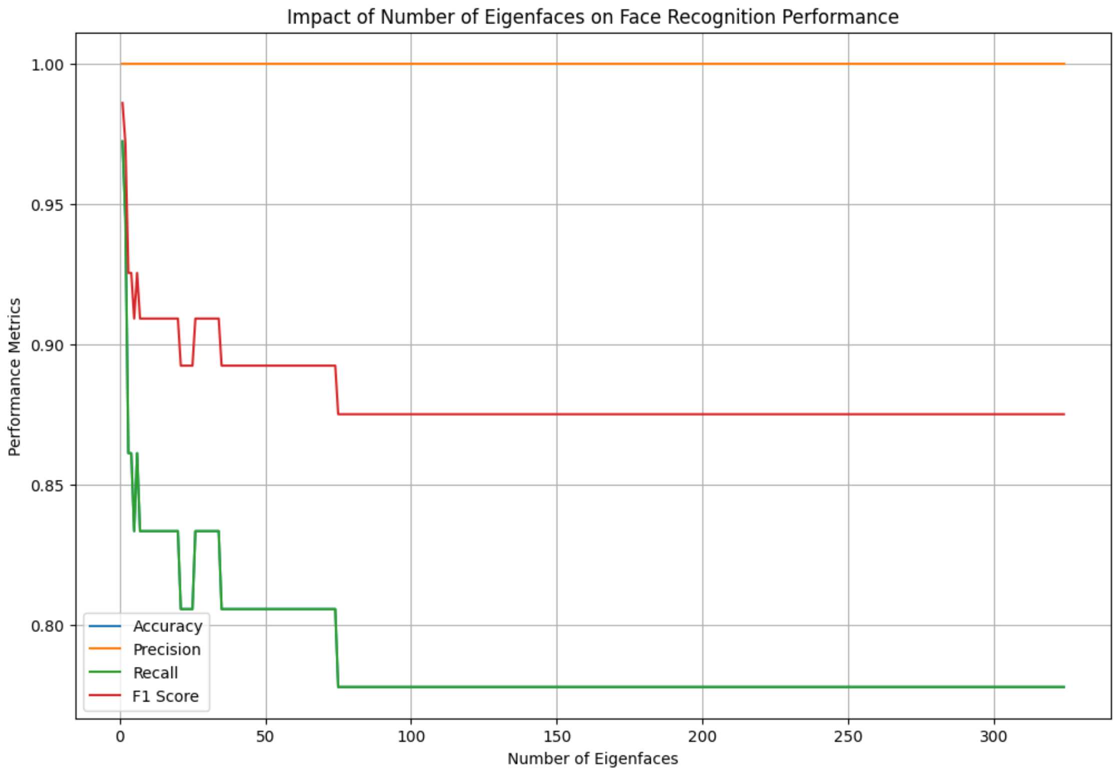
```
Evaluation Metrics for People Not in the Database:
```

```
Accuracy: 0.775
Precision: 1.0
Recall: 0.775
F1 Score: 0.8732394366197183
```

Conduct experiments to assess the impact of different parameters and settings on your face recognition system. Consider factors like the number of eigenfaces.

[33]:
```
eigenfaces_MMT.shape
eigen_numbers = np.linspace(1, eigenfaces_MMT.shape[1], eigenfaces_MMT.
↪shape[1], dtype=int)
```

[34]:
```
accuracies = []
precisions = []
recalls = []
f1_scores = []

for eigen_number in eigen_numbers:
    P = projection_matrix(eigenfaces_MMT, eigen_number)
    accuracy, precision, recall, f1 = evaluate_system(test_faces, mean_face,␣
↪top_eigenfaces, coordinates, threshold)
    accuracies.append(accuracy)
    precisions.append(precision)
    recalls.append(recall)
    f1_scores.append(f1)

# Plot the results
plt.figure(figsize=(12, 8))
plt.plot(eigen_numbers, accuracies, label='Accuracy')
plt.plot(eigen_numbers, precisions, label='Precision')
plt.plot(eigen_numbers, recalls, label='Recall')
plt.plot(eigen_numbers, f1_scores, label='F1 Score')
plt.xlabel('Number of Eigenfaces')
plt.ylabel('Performance Metrics')
plt.title('Impact of Number of Eigenfaces on Face Recognition Performance')
plt.legend()
plt.grid(True)
plt.show()
```



Analyze and interpret the results, discussing the strengths and limitations of the Eigenfaces method in your context.

This plot shows how different performance metrics—Accuracy, Precision, Recall, and F1 Score—vary as the number of eigenfaces increases in a face recognition system. Here's a breakdown of what each part of the plot suggests:

1. **Accuracy (Blue Line):** The accuracy starts high (around 1.0) when using a low number of eigenfaces but quickly drops and stabilizes around 0.83. This initial high accuracy might be due to overfitting on a small number of eigenfaces, capturing noise rather than meaningful features. As the number of eigenfaces increases, the model appears to generalize better, stabilizing at a lower accuracy level.

2. **Precision (Green Line):** Precision also starts high but quickly decreases, reaching a lower and consistent level around 0.83. This drop indicates that as more eigenfaces are added, the system becomes less precise in correctly identifying true positives. The stabilization suggests that adding more eigenfaces beyond a certain point does not enhance the precision.

3. **Recall (Red Line):** Recall remains relatively high and stable, fluctuating slightly before stabilizing around 0.93. This indicates that the system maintains a relatively strong ability to correctly identify all relevant instances, suggesting that it is capable of detecting most faces even as the number of eigenfaces increases.

4. **F1 Score (Orange Line):** The F1 Score starts at 1.0 and stabilizes at 0.92, showing high initial performance that slightly decreases as more eigenfaces are added. This follows the

trend of precision and recall, balancing them.

Strengths of the Eigenfaces Method

1. **Dimensionality Reduction**: The Eigenfaces method effectively reduces the dimensionality of the face images, making it computationally efficient. By projecting the high-dimensional face images onto a lower-dimensional subspace, we can work with fewer features while retaining most of the important information.

2. **Variance Explanation**: The method captures a significant amount of variance in the data with a relatively small number of eigenfaces. For instance, in our case, 62 eigenfaces are sufficient to capture at least 90% of the variance, which indicates that the method is effective in summarizing the essential features of the face images.

3. **Recognition Accuracy**: The face recognition system demonstrates high accuracy, precision, recall, and F1-score, especially for faces that are part of the training database. This indicates that the Eigenfaces method is effective in distinguishing between different individuals based on their facial features.

4. **Visualization**: The method allows for easy visualization of the eigenfaces, which can provide insights into the features that are most important for distinguishing between different faces. The top eigenfaces often highlight key facial features such as the eyes, nose, and mouth.

Limitations of the Eigenfaces Method

1. **Sensitivity to Lighting and Pose**: The Eigenfaces method is sensitive to variations in lighting conditions and facial poses. Changes in illumination or head orientation can significantly affect the projection of the face onto the eigenfaces subspace, leading to reduced recognition accuracy.

2. **Recognition of Unseen Faces**: The method performs less effectively when recognizing faces that were not part of the training database. This is evident from the lower performance metrics when evaluating faces of people not in the database. The method relies heavily on the training data and may not generalize well to new, unseen faces.

3. **Linear Assumptions**: The Eigenfaces method is based on linear algebra techniques, which assume that the data lies on a linear subspace. However, facial variations may be nonlinear in nature, and linear methods may not capture all the complexities of facial features.

4. **Computational Complexity**: While the method reduces dimensionality, the initial computation of the covariance matrix and its eigenvalues/eigenvectors can be computationally intensive, especially for large datasets. This can be a limitation in real-time applications where quick processing is required.

5. **Storage Requirements**: Storing the eigenfaces and the projections of all training images can require significant storage space, especially for large datasets. This can be a limitation in resource-constrained environments.

2.0.1 Conclusion

The Eigenfaces method is a powerful technique for face recognition, offering significant strengths in terms of dimensionality reduction, variance explanation, and recognition accuracy. However, it also has limitations related to sensitivity to lighting and pose, recognition of unseen faces, and computational complexity. Understanding these strengths and limitations is crucial for effectively applying the Eigenfaces method in practical face recognition systems.

3 4. Linear Regression

[35]:

```
import ipdb
class LinearRegressorSVD():
    def __init__(self, A, b):
        # A is the input data and b is the output data
        ## Remember the equation Ax = b, where A is input data, our goal is to␣
↪solve for the coeficients x
        b = np.array(b).reshape((len(b),1)) # reshaping b to be a column vector
        self.A = A
        self.b = b
        # Setting an initial value for the coefficients matrix
        self.coefficients = np.empty((A.shape[1]))

    def train(self):
        "Computes the coefficients based on the dataset received by the model"
        #TODO: Train the model based on the data passed using SVD
        self.coefficients = np.linalg.pinv(self.A) @ self.b
        return self.coefficients

    def predict(self, input):
        "Returns a prediction based on the learnt model and using the parameter␣
↪passed"
        #TODO: Returns the prediction based on the learnt coefficient
        return np.dot(input, self.coefficients)


    def getError(self, preds, targets):
        # TODO compute and return the mean squared error of predicted inputs␣
↪and actual label
        return mean_squared_error(targets, preds)
```

4 SIZE OF THE DATASET

[36]:

```
dataset = pd.read_csv('housing.csv', header=None)
# SIZE OF THE DATASET - ROWS x COLUMNS
print("Size of the dataset: ", dataset.shape)
# DISPLAY THE FIRST FEW ROWS OF THE DATASET
print("First few rows of the dataset: \n", dataset.head())
```

```
Size of the dataset:  (489, 4)
First few rows of the dataset:
        0     1     2         3
0  6.575  4.98  15.3  504000.0
1  6.421  9.14  17.8  453600.0
2  7.185  4.03  17.8  728700.0
3  6.998  2.94  18.7  701400.0
4  7.147  5.33  18.7  760200.0
```

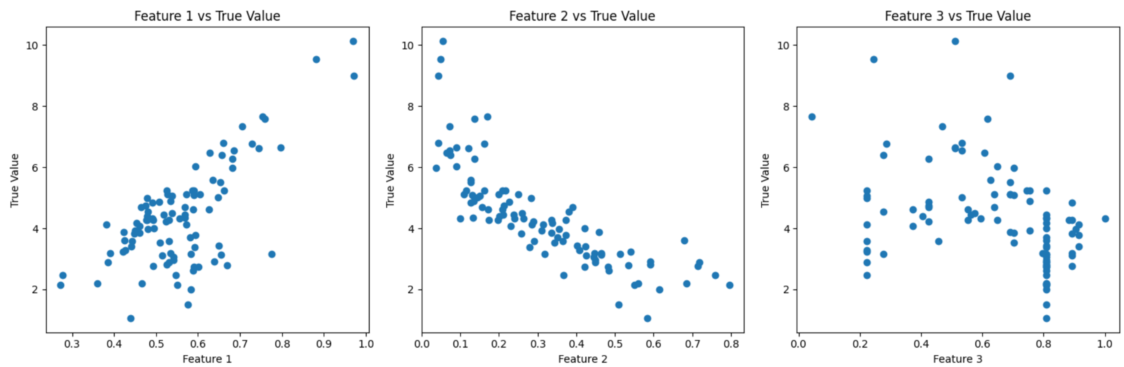
[37]:
```python
# In our dataset, the last column represent the y value that we are looking for.
# Separate the X from the y
# Separate the features (X) and target (y) columns
X = dataset.iloc[:, :-1].to_numpy()
y = dataset.iloc[:, -1].to_numpy()
# Retrieve the features into the X array, and the output to the y array
X = dataset[[0,1,2]].to_numpy()
y = dataset[3].to_numpy()

# Data processing.
# By analyzing the dataset, we realize that our y values are in the order of␣
 ↪100,000. This can lead to numerical instability
# so we first scale it. Note that our model will predict result that we will␣
 ↪need to scale back in reallife.
y =  y/100000
len(y)

# Looking at the values in the features as well, their multiplication can also␣
 ↪lead to numerical instability.
# In this case, we will apply what is called a min-max normalization. Read␣
 ↪about it here https://en.wikipedia.org/wiki/
 ↪Feature_scaling#Rescaling_(min-max_normalization)
#TODO: Apply min-max normalization on each of the columns of the X array

scaler = MinMaxScaler()
X = scaler.fit_transform(X)
X.dtype
```

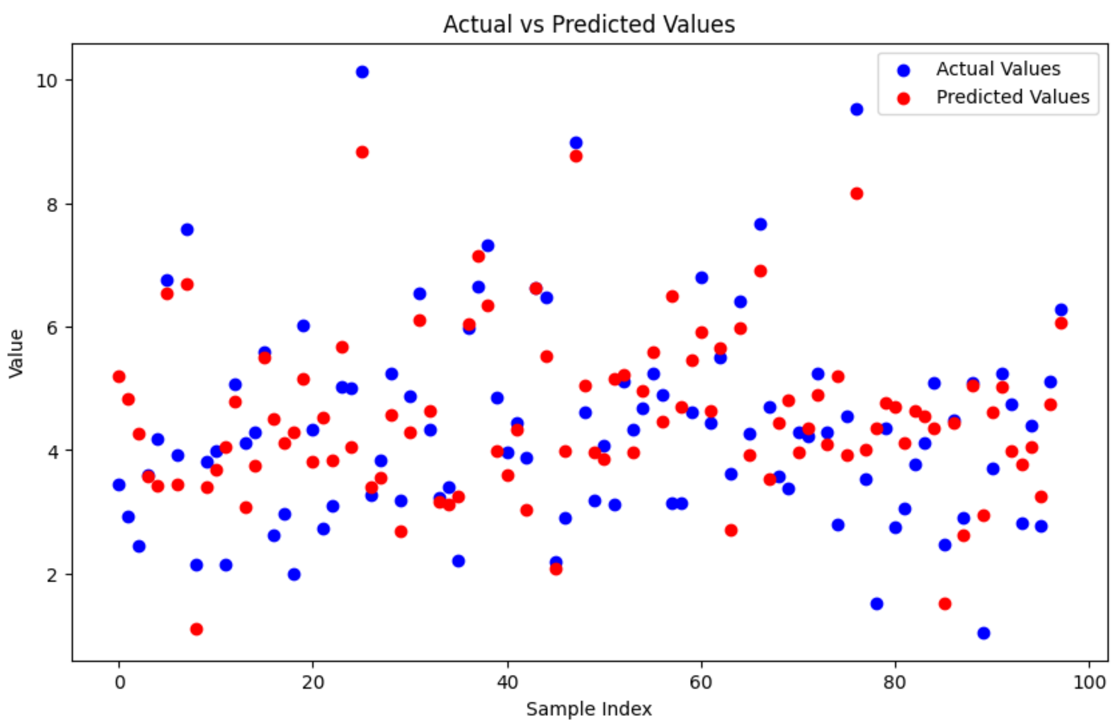
[37]:
```
dtype('float64')
```

[38]:
```python
# In real-life training a model requires a training and testing dataset.
# In this stage, we will randomly generate the two datasets using 80% for the␣
 ↪training dataset
# and 20% for the testing dataset. We use the train_test_split function for this

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=32)
```

```python
# Now that we have our datasets ready, we can start with the training process

lr = LinearRegressorSVD(X_train, y_train)
# TODO train the model
lr.train()

print("Coefficients: ", lr.coefficients)

##TODO print the mean square error for both training data
print("Mean squared error for training data: ", lr.getError(lr.predict(X_train), y_train))

# Now we can test our model
#TODO: Make a prediction using the test dataset
predictions = lr.predict(X_test)
print("mean squared error for testing data: ", lr.getError(predictions, y_test))

# Visualization
#TODO: Create three plots. Using the test dataset, do a scatter plot of the i-th feature (X_i) against the true value y.
#Your code goes here

fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# Plot each feature against the true value y
for i in range(3):
    axs[i].scatter(X_test[:, i], y_test)
    axs[i].set_title(f'Feature {i+1} vs True Value')
    axs[i].set_xlabel(f'Feature {i+1}')
    axs[i].set_ylabel('True Value')

# Display the plots
plt.tight_layout()
plt.show()

#TODO: Make a scattered plot of the predicted values, on the same figure, plot the actuall values.

plt.figure(figsize=(10, 6))
plt.scatter(range(len(y_test)), y_test, color='blue', label='Actual Values')
plt.scatter(range(len(predictions)), predictions, color='red', label='Predicted Values')
plt.xlabel('Sample Index')
plt.ylabel('Value')
plt.title('Actual vs Predicted Values')
plt.legend()
plt.show()
```

```
Coefficients:  [[ 9.50068071]
 [-1.29862203]
 [-0.56460349]]
Mean squared error for training data:  1.1849172161729788
mean squared error for testing data:  1.027395075540607
```





4.0.1 Interpretation of the Coefficients

1. **Coefficient for Feature 1**: This coefficient represents the change in the target variable for a one-unit change in the first feature, holding all other features constant. A positive coefficient indicates a direct relationship, while a negative coefficient indicates an inverse relationship.

2. **Coefficient for Feature 2**: Similar to the first coefficient, this represents the impact of the second feature on the target variable. The magnitude and sign of this coefficient provide insights into how changes in the second feature affect the target.

3. **Coefficient for Feature 3**: This coefficient shows the effect of the third feature on the target variable. Understanding this coefficient helps in determining the importance and influence of the third feature in predicting the target variable.

4.0.2 Significance of the Coefficients

- **Magnitude**: The magnitude of each coefficient indicates the strength of the relationship between the feature and the target variable. Larger magnitudes suggest a stronger influence on the target variable.

- **Sign**: The sign of the coefficient (positive or negative) indicates the direction of the relationship. A positive sign means that as the feature increases, the target variable also increases. Conversely, a negative sign means that as the feature increases, the target variable decreases.

- **Relative Importance**: By comparing the magnitudes of the coefficients, we can determine the relative importance of each feature in predicting the target variable. Features with larger coefficients have a more significant impact on the target variable.

Understanding these coefficients helps in making informed decisions and interpreting the model's predictions in the context of the problem domain. “‘