Documentation for Pattern Recognition & Visualization Code

Karim El-Sharkawy

February 2025

Please note that the code, and so by extension this documentation, builds off of the previous documentation, Documentation for Mapping Creation and Classification Code, so it would be good to skim those first before reading this. For this code, we had many goals in mind. First, we wanted to visualize the cone created by all the mappings. Second, we wanted to find ways of differentiating between extendable and nonextendable mappings via methods of visualization and their algebraic properties. Finally, we wanted to test any theories we had. This was a way for us to gain intuition on our proofs or ideas. All papers, code, and other information (including this documentation) can be found in the GitHub repository. The following Python libraries and packages are required to run the code on Google Colab:

- `numpy`: Required for handling arrays and saving matrices to `.npy` files.
- `scipy`: Specifically `scipy.optimize.linprog`, `scipy.spatial.distance`, and `scipy.linalg`.
- `matplotlib`: Specifically `pyplot` and `mpl_toolkits.mplot3d`.

1 3D Cone Visualization

After loading our necessary datasets (`Extendables`, `Nonextendables`, `Classifiers`), we create a 3D scatter plot to gain insights into the distribution of these mappings using `pyplot` and `mpl_toolkits.mplot3d`:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Assign colors and labels to each set of vectors
colors = ['r', 'g', 'b', 'y']
labels = ['Classifier', 'Extendable', "Nonextendable", 'Farthest B']

# Create a new figure and axis for 3D plotting
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot classifiers
for i in range(25):
    soa0 = goodclassifier_class[i]
    soa0 = soa0[:, :-1]  # delete the last column bc we don't use it to graph
    ax.scatter(soa0[:, 0], soa0[:, 1], soa0[:, 2], color=colors[0], alpha=0.2)

# similarly for (non)extendables

ax.set_xlim([-20, 20])
ax.set_ylim([-20, 20])
ax.set_zlim([-20, 20])

ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Z-axis')
```

```
plt.title('3D Vector Plot')

scatter1_proxy = plt.Line2D([0],[0], linestyle="none", c='r', marker = 'o')
scatter2_proxy = plt.Line2D([0],[0], linestyle="none", c='g', marker = 'o')
scatter3_proxy = plt.Line2D([0],[0], linestyle="none", c='b', marker = 'o')
ax.legend([scatter1_proxy, scatter2_proxy, scatter3_proxy], labels)

plt.show()
```

This visualization provides a spatial representation of the mappings and lets us visualize the cones shape created by the mappings (check theory work for explanation on cones). We get the following as a result:

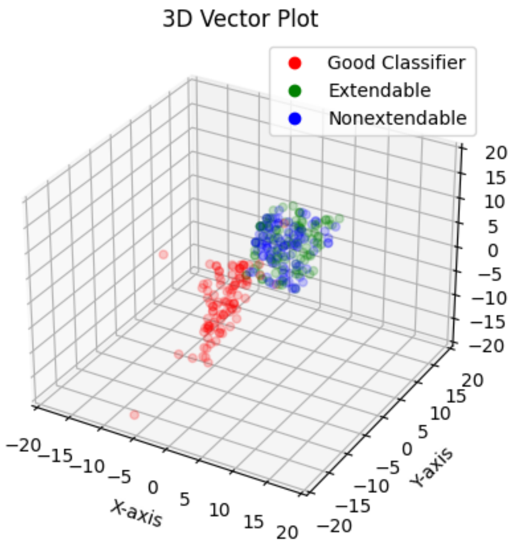
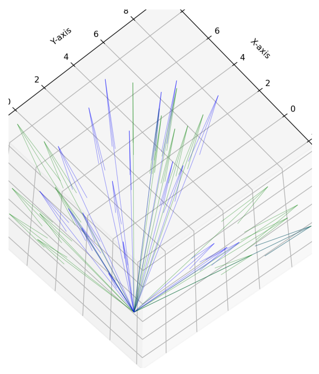
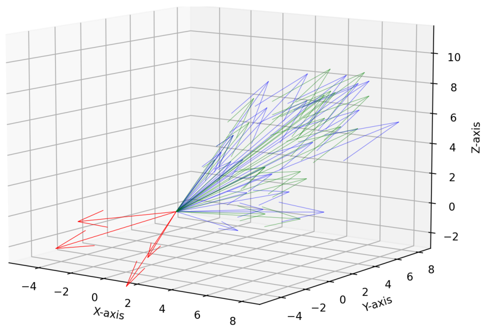


Figure 1: Positive Cone

To better see the cone shape, one can switch the points to arrows pointing towards the points. We can see:



(a) Visualization of arrows from below



(b) Visualization including classifiers

Figure 2: Cone visualized via arrows (same color scheme as Fig 1)

Notice that in the previous documentation, we mentioned that the nonextendable mappings intertwine with some of the extendable mappings, that can be confirmed visually here as well.

2 Finding Differences

2.1 Distances

2.1.1 Computing Average Distances

This function computes the average distances between two datasets, `data1` and `data2`, by calculating dot products for each pair of data points from both datasets. It then visualizes these distances in a 3D plot.

Function Definition

The function `calculate_average_distances_and_plot` takes two datasets and calculates the average distances between all pairs of data points, and generates a 3D plot to visualize the results.

```
def calculate_average_distances_and_plot(data1, data2, plot_title, xlabel, ylabel):
```

Inputs:

- `data1`, `data2`: The two datasets (likely matrices or lists of vectors).
- `plot_title`, `xlabel`, `ylabel`: Labels and title for the 3D plot.

Procedure

1. **Initialization**: Three lists are initialized: `average_distances`, `positions`, and `colors`.
2. **Looping Through Data**: The function iterates through all pairs of data points from `data1` and `data2`.
3. **Distance Calculation**: For each pair of points, the dot product is computed using np.dot(row_d2, row_d1), and stored in a list `distances`.
4. **Average Distance**: The average distance is calculated using:

$$\text{average\_distance} = \sqrt{\text{mean}(\text{square}(distances))}$$

5. **Storing Results**: The average distance, as well as the positions for plotting, are stored. The points are colored based on their index.

Plotting

A 3D scatter plot is generated using `matplotlib` where:

```
ax.scatter(positions[:, 0], positions[:, 1], positions[:, 2], c=colors, marker='o')
```

This plots the calculated average distances, with colors differentiating data from different halves of the datasets.

Result

The function returns the list of average distances and displays the plot. An example call might look like:

```
average_distances = calculate_average_distances_and_plot(data1, data2)
```

Additionally, summary statistics such as the mean, maximum, and minimum of the average distances are printed:

$$\text{print}(\text{mean}(\text{average\_distances}))$$

To quantify relationships between mappings, we compute pairwise average distances:

```
def calculate_average_distances(data1, data2):
    distances = []
    for d1 in data1:
        for d2 in data2:
            dot_product = np.dot(d1.flatten(), d2.flatten())
            distances.append(np.sqrt(np.mean(np.square(dot_product))))
    return distances

average_distances = calculate_average_distances(Extendables[:500], Nonextendables[:500])
print(f"Mean average distance: {np.mean(average_distances)}")
```

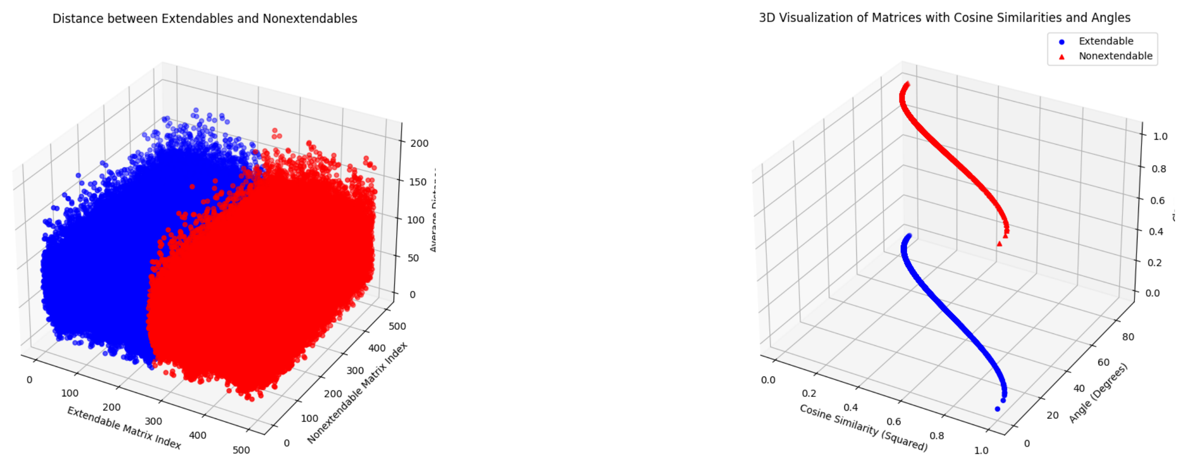
2.1.2 Cosine Distance Analysis

Another useful metric is the cosine similarity between mappings:

```
from scipy.spatial.distance import cosine

def calculate_cosine_similarity(m1, m2):
    return 1 - cosine(m1.flatten(), m2.flatten())

cosine_distances = [calculate_cosine_similarity(e, n) for e in Extendables[:100] for n in Nonextendables[:1
print(f"Mean cosine similarity: {np.mean(cosine_distances)}")
```



(a) Visualization of arrows from below

(b) Visualization including classifiers

Figure 3: Cone visualized via arrows (same color scheme as Fig 1)

3 Algebraic Properties

To analyze the algebraic properties of the matrices, we compute their rank, determinant, eigenvalues, and null space:

```
from numpy.linalg import matrix_rank
from scipy.linalg import lu, inv

def matrix_properties(matrix):
    rank = matrix_rank(matrix)
    determinant = np.linalg.det(matrix)
    eigenvalues, eigenvectors = np.linalg.eig(matrix)
    _, s, V = np.linalg.svd(matrix)
    tolerance = max(matrix.shape) * np.spacing(np.max(s))
    null_space = V.T[:, s < tolerance]
    Q, _ = np.linalg.qr(matrix)
    column_space = Q[:, :rank]
    P, L, U = lu(matrix)
    L_inv = inv(L)
    RREF = np.dot(L_inv, np.dot(P, matrix))
    return {
        'rank': rank,
        'determinant': determinant,
        'eigenvalues': eigenvalues,
        'eigenvectors': eigenvectors,
        'null_space': null_space,
        'column_space': column_space,
        'rref': RREF
    }
```

3.1 Coplanarity and Collinearity

We also examine the coplanarity and collinearity of the matrices:

```
coplanerCountEX = sum(matrix_rank(matrix) < 3 for matrix in extendable_class)
coplanerCountNEX = sum(matrix_rank(matrix) < 3 for matrix in nonextendable_class)
coplanerCountCLA = sum(matrix_rank(matrix) < 3 for matrix in goodclassifier_class)

print(f"Percentage of coplanar extendables: {coplanerCountEX/len(extendable_class)*100}%")
print(f"Percentage of coplanar nonextendables: {coplanerCountNEX/len(nonextendable_class)*100}%")
print(f"Percentage of coplanar classifiers: {coplanerCountCLA/len(goodclassifier_class)*100}%")
```

3.2 Collinearity Scores

Finally, we compute collinearity scores to determine linear dependencies among rows:

```
def row_collinearity_score(matrix):
    min_abs_det = np.inf
    for i in range(4):
        submatrix = np.delete(matrix, i, axis=0)
        for j in range(4):
            if i != j:
                submatrix_temp = np.delete(submatrix, j, axis=1)
                if submatrix_temp.shape[0] == submatrix_temp.shape[1]:
                    det = np.abs(np.linalg.det(submatrix_temp))
                    min_abs_det = min(min_abs_det, det)
    return min_abs_det

equalszero = sum(row_collinearity_score(matrix) == 0 for matrix in goodclassifier_class)
nozero = len(goodclassifier_class) - equalszero
print(equalszero, nozero)
```

4 Classifier Verification

We check whether the classifiers meet specified conditions:

```
passed_matrices = []
failed_matrices = []

for matrix in goodclassifier_class:
    passes_all_conditions = all(
        row[0] + row[1] - row[2] - row[3] >= 0 for row in matrix
    )

    if passes_all_conditions:
        passed_matrices.append(matrix)
    else:
        failed_matrices.append(matrix)

print(f"{len(passed_matrices)} Classifiers passed the condition")
print(f"{len(failed_matrices)} Classifiers failed the condition")
```

4.1 Assumption Bin

We analyze positivity and negativity distributions across different classes:

```
def all_positive_nested(matrix):
    return all(value > 0 for row in matrix for value in row)

def nonNeg_nested(matrix):
    return all(value >= 0 for row in matrix for value in row)

positive = sum(all_positive_nested(matrix) for matrix in extendable_class)
nonnegative = sum(nonNeg_nested(matrix) for matrix in extendable_class)
positive1 = sum(all_positive_nested(matrix) for matrix in nonextendable_class)
nonnegative1 = sum(nonNeg_nested(matrix) for matrix in nonextendable_class)
positive2 = sum(all_positive_nested(matrix) for matrix in goodclassifier_class)
nonnegative2 = sum(nonNeg_nested(matrix) for matrix in goodclassifier_class)

print(f"Number of positive extendable matrices: {positive}")
print(f"Number of nonnegative extendable matrices: {nonnegative}\n")
print(f"Number of nonnegative nonextendable matrices: {nonnegative1}")
print(f"Number of nonnegative classifiers: {nonnegative2}")
```