# Documentation for Mapping Creation and Classification

Karim El-Sharkawy

February, 2025

The code titled *Mapping Creation and Classification* generates $4 \times 4$ matrices (mappings) satisfying certain conditions, classifies them based on extendability using linear programming, and applies Support Vector Machines (SVMs) to find classifiers that linearly separate extendable and non-extendable mappings. 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` is used for solving linear programming problems.
- `random`: Used for generating random numbers and shuffling within the `backtracking` function.
- `sklearn`: Specifically, the `svm` module from `sklearn` is used for Support Vector Machine (SVM) classification.

## 1 Creating positive mappings and classifying extensions

The program generates $4 \times 4$ matrices that satisfy properties of $E_{2,2}$. The key conditions checked during generation are:

- **Row Condition**: The sum of the first two elements in each row must equal the sum of the last two elements.
- **Column Condition**: The sum of the elements in the first and fourth rows must equal the sum of the second and third rows.
- **Positivity Check**: The sum of any two elements in a row must be positive.

This Python code generates matrices that meet specific conditions and saves them to a `.npy` file so we could manipulate them using Numpy without making new mappings everytime.

The first function, `get_random_number()` simply generates and returns a random integer between $-9$ and 9. The first significant function is `is_valid(matrix, rows, cols, row, col, num)` which checks whether a given number `num` can be placed at position $(row, col)$ in the `matrix` based on three conditions:

- **Row condition**: Ensures that for each row, the sum of the first two elements should equal the sum of the last two elements. This is checked when `col == 3` (the last column).
- **Column condition**: The sum of the elements in the 1st and 4th rows should equal the sum of elements in the 2nd and 3rd rows. This condition is checked when `row == 3` (the last row). This is our convention and what we used in our theory, but it doesn't have to be this configuration.
- **Positivity check**: Ensures that adding any two elements in the row results in a positive value (i.e., no negative sums). This is checked when `col == 3` (the last column).

If any of these conditions are violated, the function returns `False`, preventing the number from being placed at that position. If all conditions are satisfied, it returns `True`.

### 1.1 create_matrix(rows, cols)

```
def create_matrix(rows, cols):
    matrix = [[None for _ in range(cols)] for _ in range(rows)]

    backtrackFailedAttempts = 0
    def backtrack(row, col):
        nonlocal backtrackFailedAttempts
        if backtrackFailedAttempts > 1000:
            return False

        if row == rows:
            return True

        nums = [i for i in range(-9, 10)]  # range [-9, 10)
        random.shuffle(nums)

        for num in nums:
            if is_valid(matrix, rows, cols, row, col, num):
                matrix[row][col] = num

                next_row = row
                next_col = col + 1
                if next_col == cols:
                    next_row += 1
                    next_col = 0

                if backtrack(next_row, next_col):
                    return True

        matrix[row][col] = None
        backtrackFailedAttempts = backtrackFailedAttempts + 1
        return False

    backtrack(0, 0)  # recursion

    return matrix
```

This function generates a random valid matrix using a backtracking approach. Initially, the matrix is created with all elements set to `None`. It then attempts to fill the matrix row by row and column by column, trying random numbers from the range $[-9, 9]$. To ensure randomness, the numbers are shuffled using `random.shuffle(nums)`. For each number, the function checks its validity by calling `is_valid` to determine if the number can be placed at the current position. If the number passes the validation, the function proceeds to the next column. When the last column is reached (`col == 3`), it moves to the next row. If the entire matrix is successfully filled (`row == rows`), the function returns `True`, indicating a valid matrix has been generated.

If an invalid number is placed at any point, it backtracks by removing the number and trying another value. If no valid configuration is found after 1000 failed attempts, it backtracks and tries a new configuration.

### 1.2 Main Code Block

```
numberOfMappingsToCreate = 10000
listOfMappings = []
for i in range(numberOfMappingsToCreate):
    rows = 4
    cols = 4
    isAValidMatrix = False
    while not isAValidMatrix:
        isAValidMatrix = True
```

```
        matrix = create_matrix(rows, cols)
        for row in matrix:
            for element in row:
                if element is None:
                    isAValidMatrix = False
    matrix[1], matrix[3] = matrix[3], matrix[1]
    listOfMappings.append(matrix)
```

This block of code brings everything together by creating numberOfMappingsToCreate number of mappings and adds them to the list `listOfMappings`. After generating a valid matrix, the second and fourth rows are swapped. The reason for swtiching rows is to preserve the column condition. Having the code determine the the second row when it hasn't completed the third or fourth leads to long computation times, so we instead had it calculate each value in the fourth row to staisfy all conditions and then switched it with the second row. Hence, The sum of the first and fourth rows equals the sum of the second and third rows.

```
listOfMappings_array = np.array(listOfMappings)
np.save('/content/listOfMappings.npy', listOfMappings_array)
```

Finally, we convert the list of mappings into a `numpy` array and save them to a `.npy` file so that we can easily load and manipulate them later.

## 1.3 Checking for Extendability using Linear Programming

To classify mappings, a linear programming (LP) problem is set up using the `linprog` function from the `scipy.optimize` library. LP determines the feasibility of the constraints for each matrix. The problem is formulated as:

$$\min \mathbf{c}^T \mathbf{x}, \quad \text{subject to:}$$
$$A_{ub}\mathbf{x} \leq \mathbf{b}_{ub}, \quad A_{eq}\mathbf{x} = \mathbf{b}_{eq}$$

Where:

- $\mathbf{x}$ is our decision vector $t$, representing the variables to be optimized.
- $\mathbf{c} = [1, 1, 1, 1]$ are the objective coefficients, which in this case are constant.
- $A_{ub}$ and $\mathbf{b}_{ub}$ define bounds derived from the matrix entries using the min and max functions.
- $A_{eq} = [[1, 1, -1, -1]]$ and $\mathbf{b}_{eq} = [0]$ impose the constraint that $x_1 + x_2 = x_3 + x_4$.

The solution $\mathbf{x}$ determines the extendability of the matrix. If the solution is feasible, the matrix is *extendable*. If not, it is *non-extendable*. The theory behind this is provided in the paper *Insights into the untanglement mapping, the properties of positive mappings, and the conditions for extension* by Sinclair, El-Sharkawy, Rajamani, and Luschwitz.

The code uses the `linprog` function from the `scipy.optimize` library to solve the linear program. The objective function is the vector $\mathbf{c} = [1, 1, 1, 1]$, which is minimized over the decision variables $\mathbf{x}$. The constraints are defined for each mapping as follows:

### 1.3.1 Upper Bound Constraints $A_{ub}\mathbf{x} \leq \mathbf{b}_{ub}$

These constraints ensure that the values of $x_1, x_2, x_3, x_4$ stay within bounds derived from the matrix entries. The bounds are calculated based on the minimum and maximum values from the matrix entries:

$$A_{ub} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

$$\mathbf{b}_{ub} = \begin{bmatrix} \min(matrix[0][2], matrix[0][3]) \\ \min(matrix[1][2], matrix[1][3]) \\ \min(matrix[2][2], matrix[2][3]) \\ \min(matrix[3][2], matrix[3][3]) \\ -\max(-matrix[0][0], -matrix[0][1]) \\ -\max(-matrix[1][0], -matrix[1][1]) \\ -\max(-matrix[2][0], -matrix[2][1]) \\ -\max(-matrix[3][0], -matrix[3][1]) \end{bmatrix}$$

#### 1.3.2 Equality Constraints $A_{eq}\mathbf{x} = \mathbf{b}_{eq}$

The equality constraint ensures that the sum of the first two variables equals the sum of the last two:

$$A_{eq} = \begin{bmatrix} 1 & 1 & -1 & -1 \end{bmatrix}, \quad \mathbf{b}_{eq} = \begin{bmatrix} 0 \end{bmatrix} \leftrightarrow x_1 + x_2 - x_3 - x_4 = 0$$

If `result.success` is `True` (all conditions are satisfied), the matrix is considered feasible and is classified as *extendable*. If `result.success` is `False`, the matrix is considered *non-extendable*. The matrices are then stored in two separate lists: `extendableMappings` and `nonExtendableMappings`.

Finally, the extendable mappings are similarly saved to `.npy` files. The code also includes commented-out print statements, which display each matrix and its extendability status.

## 2 Farthest Bs and Making Classifiers using SVMs

An understanding of Classification is key to understanding the later code. Briefly, binary classification using Support Vector Machines (SVMs) involves finding a hyperplane that best separates data points from two classes. SVMs work by mapping the input data into a higher-dimensional space (if necessary) using a kernel function, enabling the algorithm to find a linear boundary even for data that isn't linearly separable in its original form. The goal is to identify the hyperplane that maximizes the margin, which is the distance between the closest data points (support vectors) from each class. A larger margin typically leads to better generalization and reduced risk of overfitting.

Once the optimal hyperplane is found, SVM uses this boundary to classify new data points. Data points on one side of the hyperplane are assigned to one class, while those on the other side are assigned to the other class. SVM is particularly powerful for high-dimensional datasets and is effective even when data points are not linearly separable, thanks to its ability to use kernel functions.

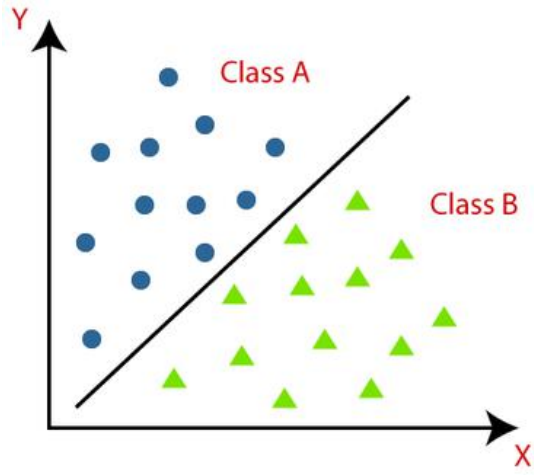


Figure 1: Example of simple binary classification (paperswithcode)

### 2.1 Farthest $B$ from $A$

However, in our case, Class B (nonextendables) is intertwined with some points of Class A (extendables), meaning there is no linear classifier that can separate the classes. To set up the proper environment to find linear separable

classifiers, we first need to find the farthest mappings from the matrices in set $A$. The function computes the total distance from each nonextendable to all extendables and then sorts them by distance using `np.linalg.norm`.

```
def farthestBFromA(setB, setA):
  totalDistancesFromA = []
  for B in setB:
    currentBTotalDistance = 0
    for A in setA:
      currentBTotalDistance = currentBTotalDistance + (np.linalg.norm(np.subtract(B, A)) ** 2)
    totalDistancesFromA.append(currentBTotalDistance)

  orderedBsByDistance = []
  for i in range(len(totalDistancesFromA)):
    index = np.argmax(totalDistancesFromA)
    orderedBsByDistance.append(setB[index])
    totalDistancesFromA[index] = -np.inf
  return orderedBsByDistance

farthestBs = farthestBFromA(nonExtendableMappings, extendableMappings)
```

## 2.2 SVM Classifier Creation

The function `CreateClassifier(farthestBIndex)` builds and trains a linear Support Vector Machine (SVM) classifier to properly separate mappings that extend from those that don't extend.

```
def CreateClassifier(farthestBIndex):
  global extendableMappings
  global nonExtendableMappings
  global farthestBs

  extendableMappings = np.array(extendableMappings)
  nonExtendableMappings = np.array(nonExtendableMappings)
  farthestBs = np.array(farthestBs)

  features = np.concatenate((
      extendableMappings,
      [farthestBs[farthestBIndex]],
      [extendableMappings[0]/1000],
      [farthestBs[farthestBIndex]/1000],
      [np.zeros((4, 4))]
  ))

  features = features.reshape(len(features), -1)
  labels = [0]*len(extendableMappings) + [1,0,1,1]

  model = svm.SVC(kernel='linear', C=1e10, coef0=0.0, tol=1e-5)
  model.fit(features, labels)

  model.intercept_ = [0.0]

  predictions = model.predict(features)

  accuracy = accuracy_score(labels, predictions)

  coefficients = model.coef_
  intercept = model.intercept_

  return coefficients
```

The features for the classifier are created by concatenating several components (which are reshaped into a 2D array):

1. `extendableMappings`
2. A single value from `farthestBs` indexed by `farthestBIndex`
3. Scaled versions of the first element of `extendableMappings` and `farthestBs[farthestBIndex]`
4. A fixed-size zero matrix (`np.zeros((4, 4))`)

The labels are composed of zeros, matching the length of `extendableMappings`, and a fixed array $[1, 0, 1, 1]$. An SVM model is then instantiated with a linear kernel and trained on the features and labels using `svm.SVC()`. The regularization parameter (`C`) is set to a very high value to avoid misclassifications.

After training, the intercept of the model is manually set to [0.0] so the classifiers are linear. Predictions are made on the features, and the model's coefficients are returned as the output of the function. The SVM model returns classifiers with very small numbers that are difficult to work with, so we rounded each value to ensure they contain only whole numbers:

```
def RoundClassifier(classifier):
  classifier = np.array(classifier)
  c = classifier.copy()
  return np.round(c/10)
```

## 2.3 Checking Classifier Validity

This function checks the classifier against the sample mappings to verify if it properly classifies all the data into extendable and non-extendable categories.

```
def CheckClassifierAgainstSamples(classifier):
  allMappings = np.concatenate((extendableMappings, nonExtendableMappings), axis=0)
  allMappings = allMappings.reshape(len(allMappings), -1)

  totalMappingsInClass1 = 0
  totalMappingsInClass2 = 0
  classifiedMappingResults = []

  for i in range(len(allMappings)):
    dotProduct = np.dot(classifier, allMappings[i])
    dotProductNormalized = dotProduct/(np.linalg.norm(classifier) * np.linalg.norm(allMappings[i]))
    extendableOrNot = "E" if i<len(extendableMappings) else "N"
    if dotProduct <= 0:
      classifiedMappingResults.append((allMappings[i], extendableOrNot, 1, i, dotProduct, dotProductNormali
      totalMappingsInClass1 += 1
    else:
      classifiedMappingResults.append((allMappings[i], extendableOrNot, 2, i, dotProduct, dotProductNormali
      totalMappingsInClass2 += 1

  largestMisclassification = 0
  isGoodClassifier = True
  for mapping, extendability, _class, mapID, dotProduct, dotProductNormalized in classifiedMappingResults:
    if (extendability == "E" and _class == 2):
      isGoodClassifier = False
      if dotProductNormalized > largestMisclassification:
        largestMisclassification = dotProductNormalized

  return isGoodClassifier
```

### 2.4 Linear Programming to Check True Classifier

We use linear programming again to verify if the found classifier is truly separating the extendable from the non-extendable mappings.

```
from scipy.optimize import linprog

def CheckIfTrueClassifier(classifier):
  classifier = np.array(classifier)
  c = -classifier.reshape(1, 16)
  c = np.concatenate((c, np.zeros((1, 4))), axis=1)

  for_A_ub = [[-1, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [-1, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
              [0, -1, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

  for_A_eq = [[1, 0, 0, 0, 1, 0, 0, 0, -1, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0],
              [0, 1, 0, 0, 0, 1, 0, 0, 0, -1, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0]]

  return linprog(c=c, A_ub=for_A_ub, b_ub=for_b_ub, A_eq=for_A_eq, b_eq=for_b_eq, bounds=(None, None))
```

## 3 Removing Duplicates

Looking through the files of mappings and classifiers, we noticed there were some duplicates, so the following function takes an array of matrices, flattens each matrix, and removes duplicates by storing already seen matrices in a set.

```
def remove_duplicates(matrix_array):
    seen = set()
    unique_matrices = []

    for matrix in matrix_array:
        matrix_tuple = tuple(matrix.flatten())
        if matrix_tuple not in seen:
            seen.add(matrix_tuple)
            unique_matrices.append(matrix)

    return np.array(unique_matrices)
```

### 3.1 Reshaping and Saving Classifiers

Finally, it was also noticed that the classifiers in `goodclassifier_class` weren't formatted in a consistent manner with the (non)extendable mappings, which made manipulation difficult using `numpy`, so we reshaped them to have dimensions of $4 \times 4$.

```
extendable_class = np.load("/content/extendableMappings.npy")
nonextendable_class = np.load("/content/nonExtendableMappings.npy")
goodclassifier_class = np.load("/content/2900trueGoodClassifiers.npy")

reshaped_matrices = []
for i in range(len(goodclassifier_class)):
    reshaped_matrix = goodclassifier_class[i].reshape((4, 4))
    reshaped_matrices.append(reshaped_matrix)
reshaped_goodclassifier_class = np.array(reshaped_matrices)
reshaped_goodclassifier_class = reshaped_goodclassifier_class.astype(int)
goodclassifier_class = reshaped_goodclassifier_class

Extendables = np.array(extendable_class)
```

```
Nonextendables = np.array(nonextendable_class)
Classifiers = np.array(goodclassifier_class)

print("Extendable count: ", len(Extendables))
print("Nonextendable count: ", len(Nonextendables))
print("Classifier count: ", len(Classifiers))
```

By the end of this code, you now have three files: Extendables, Nonextendables, and Classifiers.

# 4 Contributions

**Professor Thomas Sinclair** served as the principal investigator, guiding the team in shaping the direction of the project and translating theoretical concepts into practical code. **Luke Luschwitz** was the lead programmer, focusing on the implementation of Support Vector Machines (SVMs) and the linear programming (linprog) sections, which were essential for developing and optimizing the classifiers. **Raja Darshini Darshini** played a key role in creating the initial mappings with the three constraints. She also helped implement the matrix duplication removal and managing unique matrices. **Karim El Sharkawy** translated mathematical theory into practical code, helped develop the linear programming solution to determine extendability, and took the lead in matrix reshaping and removal of duplicates.