TOPSIS in Python: Sharing Insights on Effective Decision-Making

Anthonius Valentino B. P
JCDS 0408 BDG

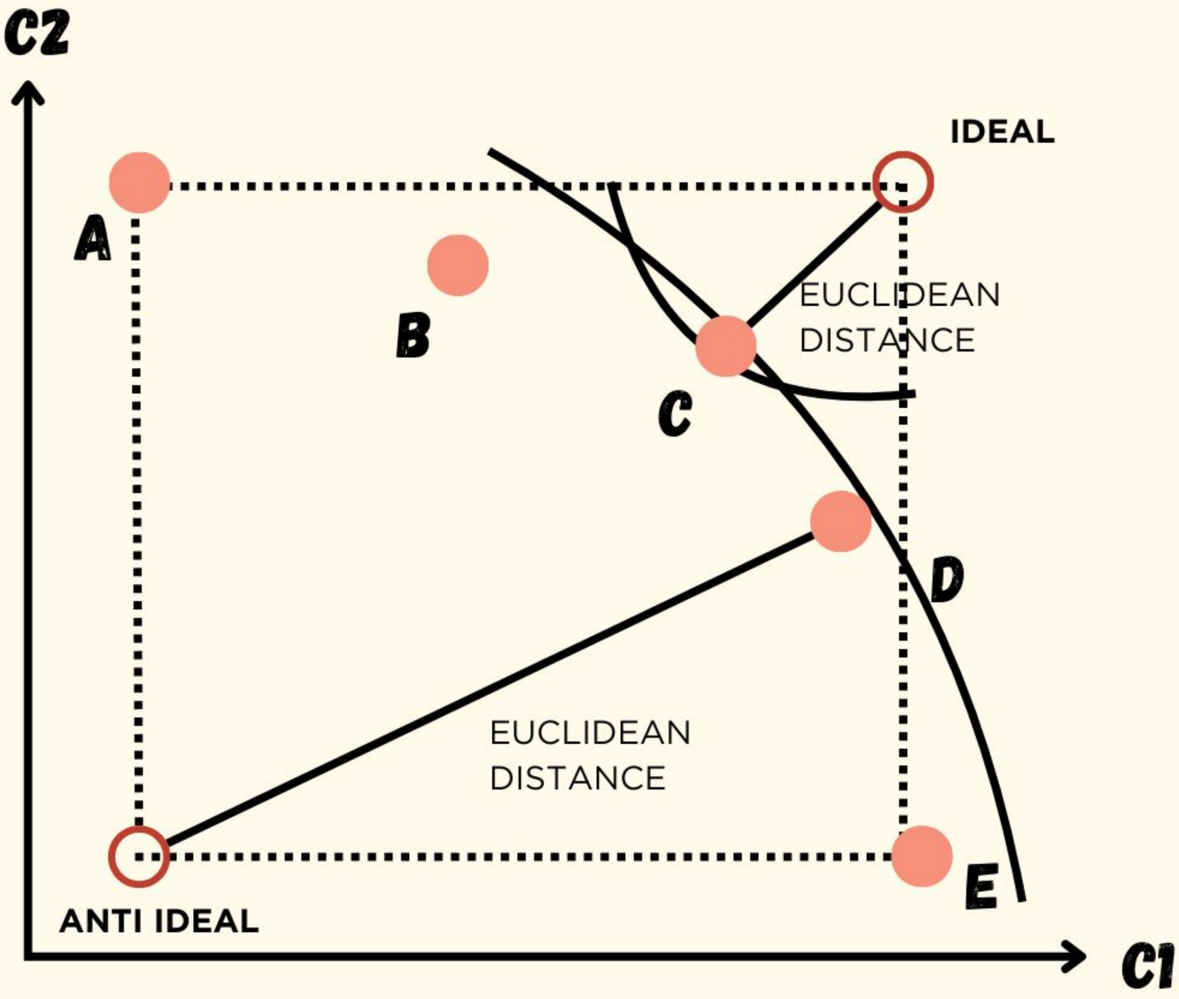
ALL ROADS LEAD TO ROME?

MCDA

Multiple Criteria Decision Analysis

TOPSIS

Technique for Order of Preference by Similarity to Ideal Solution


C2
IDEAL
A
B
EUCLIDEAN
DISTANCE
C
D
EUCLIDEAN
DISTANCE
E
ANTI IDEAL
C1

WHY THIS TOPIC?

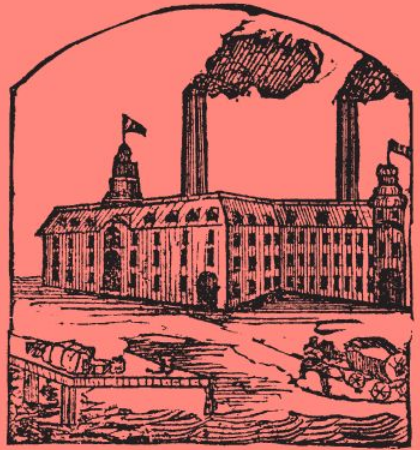
Evaluating Airline Competitiveness

Chang et al. (2001)

1. ***Cost*** *(unit operating cost)*
2. ***Productivity*** *(labour, fleet, passenger load factor)*
3. ***Service Quality*** *(on time, safety, flight frequemcy)*
4. ***Price*** *(average price)*
5. ***Management*** *(revenue growth, net profit margin, market share)*

Supplier Selection in Electronic Industry

Lin et al. (2011)



1. ***Price*** *(material, assembly, transportation, etc.)*
2. ***Quality*** *(yield rate, innovation, repairability, etc.)*
3. ***Service*** *(attitude, communication, response speed, etc.)*
4. ***Delivery*** *(accuracy, lead time, location)*
5. ***Trust****(credibility, capability)*

Case Study

"Imagine that you are part of the expansion department of a major ice cream chain. Business is booming, and now the company wants to open a new store"

Criteria

RENT COST ($) : Lower is better
FOOT TRAFFIC : Higher is better
NUMBER OF COMPETITION : Lower is better
PROXIMITY TO SCHOOL : Lower is better
PARKING SPOT : Higher is better

Criteria

	RENT COST ($)	FOOT TRAFFIC	NUMBER OF COMPETITION	PROXIMITY TO SCHOOL (KM)	PARKING SPOT
LOCATION A	2500	1500	3	1.2	15
LOCATION B	2000	1300	2	0.5	20
LOCATION C	3000	2000	5	2	10
LOCATION D	2200	1700	4	1	12

Step 1 : Performance Matrix

```
criteria_matrix = [
    [2500,1500,3,1.2,15],
    [2000,1300,2,0.5,20],
    [3000,2000,5,2,10],
    [2200,1700,4,1,12]
]

criteria_weight= [0.2,0.2,0.2,0.2,0.2]
criteria_preference = [-1,1,-1,-1,1]
```

Step 2 : Normalized Matrix

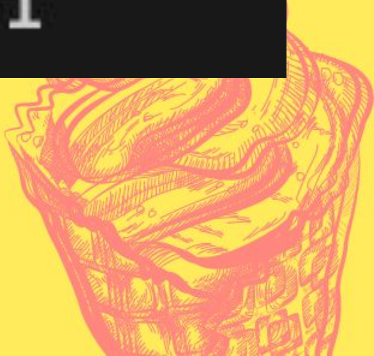
$$Normalized\,Value = \frac{Original\,Value}{\sqrt{\Sigma(Original\,value^2)}}$$

Step 2 : Normalized Matrix

```python
def normalize_decision_matrix(criteria_matrix):
    criteria_rows = len(criteria_matrix)
    criteria_columns = len(criteria_matrix[0])
    column_sums= [0] * criteria_columns

    for j in range(criteria_columns):
        for i in range(criteria_rows):
            column_sums[j] += criteria_matrix[i][j] ** 2
    column_sums = [value ** 0.5 for value in column_sums]

    normalized_matrix = []
    for i in range(criteria_rows):
        normalized_matrix_rows = []
        for j in range(criteria_columns):
            normalized_matrix_rows.append(criteria_matrix [i][j]/column_sums[j])
        normalized_matrix.append(normalized_matrix_rows)

    print('\nNormalized matrix from criteria matrix: ')
    for i in normalized_matrix:
        for j in i:
            print(f'{j:.4f}',end=' ')
        print()
    return normalized_matrix
```

Step 2 : Normalized Matrix

```
Normalized matrix from criteria matrix:
0.5094 0.4558 0.4082 0.4639 0.5088
0.4075 0.3950 0.2722 0.1933 0.6785
0.6112 0.6077 0.6804 0.7732 0.3392
0.4482 0.5166 0.5443 0.3866 0.4071
```



Step 3 : Weighted Matrix

```
def weighted_matrix(criteria_weight, normalized_matrix):
    normalized_rows = len(normalized_matrix)
    normalized_columns = len(normalized_matrix[0])

    weighted_matrix = []
    for i in range(normalized_rows):
        weighted_matrix_rows = []
        for j in range(normalized_columns):
            weighted_matrix_rows.append(normalized_matrix [i][j]* criteria_weigh
        weighted_matrix.append(weighted_matrix_rows)

    print('\nWeighted matrix from normalized matrix: ')
    for i in weighted_matrix:
        for j in i:
            print(f'{j:.4f}',end=' ')
        print()
    return weighted_matrix
```

Step 3 : Weighted Matrix

```
Weighted matrix from normalized matrix:
0.1019 0.0912 0.0816 0.0928 0.1018
0.0815 0.0790 0.0544 0.0387 0.1357
0.1222 0.1215 0.1361 0.1546 0.0678
0.0896 0.1033 0.1089 0.0773 0.0814
```

Step 4 : Positive and Negative Ideal Solution

```
def ideal_best_worst(weighted_matrix,criteria_preferences):
    weighted_column = len(weighted_matrix[0])
    positive_ideal= []
    negative_ideal = []

    for j in range(weighted_column):
        max_value = weighted_matrix[0][j]
        min_value = weighted_matrix[0][j]

        for i in range(len(weighted_matrix)):
            if weighted_matrix[i][j] > max_value:
                max_value = weighted_matrix [i][j]
            if weighted_matrix[i][j] < min_value:
                min_value = weighted_matrix [i][j]
        if criteria_preferences[j] == 1:
            positive_ideal.append(max_value)
            negative_ideal.append(min_value)
        else:
            positive_ideal.append(min_value)
            negative_ideal.append(max_value)

    print('\nPositive ideal point for each column: ')
    for i in positive_ideal:
        print(f'{i:.4f}',end=' ')
    print()
    print('\nNegative ideal point for each column: ')
    for i in negative_ideal:
        print(f'{i:.4f}',end=' ')
    print()

    return positive_ideal, negative_ideal
```

Step 4 : Positive and Negative Ideal Solution

```
Positive ideal point for each column:
0.0815 0.1215 0.0544 0.0387 0.1357

Negative ideal point for each column:
0.1222 0.0790 0.1361 0.1546 0.0678
```

Step 5 : Separation Measures

$$Positive\ Separation : S_i^{+} : \sqrt{\Sigma_{j=1}^{n}(V_{ij} - A_j^{+})^2}$$

$$Negative\ Separation : S_i^{-} : \sqrt{\Sigma_{j=1}^{n}(V_{ij} - A_j^{-})^2}$$

Step 5 : Separation Measures

```
def separation_from_ideal_point(weighted_matrix,positive_ideal,negative_ideal):
    weighted_rows = len(weighted_matrix)
    positive_separation = []
    negative_separation = []

    for i in range(weighted_rows):
        pos_sep = 0
        neg_sep = 0
        for j in range(len(positive_ideal)):
            pos_sep += (weighted_matrix[i][j] - positive_ideal[j]) ** 2
            neg_sep += (weighted_matrix[i][j] - negative_ideal[j]) ** 2
        positive_separation.append(pos_sep ** 0.5)
        negative_separation.append(neg_sep ** 0.5)

    print('\nPositive separation: ')
    for i in (positive_separation):
        print(f'{i:.4f}')
    print('\nNegative separation: ')
    for i in (negative_separation):
        print(f'{i:.4f}')
    return positive_separation,negative_separation
```

Step 5 : Separation Measures

```
Positive separation:
0.0785
0.0425
0.1624
0.0883

Negative separation:
0.0922
0.1624
0.0425
0.0925
```

Step 6 : Relative Closeness to Ideal Solution

$$C^* = \frac{S^-}{S^+ + S^-}$$

Step 6 : Relative Closeness to Ideal Solution

```
def similarities_to_PIS(positive_separation,negative_separation):
    num_rows = len(positive_separation)
    relative_similarity = []

    for i in range(num_rows):
        pos_sep = positive_separation[i]
        neg_sep = negative_separation[i]
        similarity = neg_sep/(pos_sep + neg_sep)
        relative_similarity.append(similarity)

    print('\nOrder: ')
    for i in (relative_similarity):
        print(f'{i:.4f}')
    return relative_similarity
```

Step 6 : Relative Closeness to Ideal Solution

```
Order:
0.5402
0.7924
0.2076
0.5115
```

Step 6 : Relative Closeness to Ideal Solution

	RENT COST ($)	FOOT TRAFFIC	NUMBER OF COMPETITION	PROXIMITY TO SCHOOL (KM)	PARKING SPOT
LOCATION B	2000	1300	2	0.5	20
LOCATION A	2500	1500	3	1.2	15
LOCATION D	2200	1700	4	1	12
LOCATION C	3000	2000	5	2	10

Conclusion

- By leveraging fundamental Python programming skills, we can perform simple MCDA calculations to solve real-world problems with multiple criteria.
- This systematic approach helps us evaluate and rank alternatives, ensuring well-informed decisions, such as selecting the best location for a new ice cream store.