CSED/AIGS 526: Data Mining

Your name: Jaeyoon Sim

Student ID: 20222421

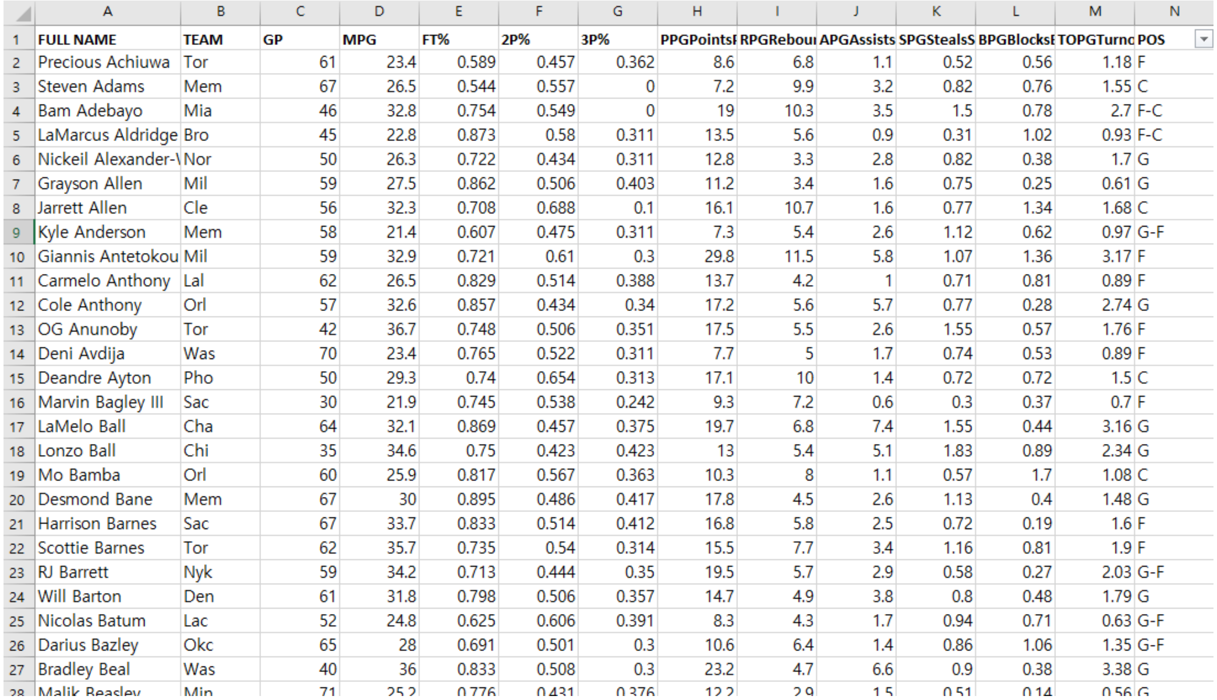
Use late days quota(Yes/No): No

In the case of the file used for question 1, we received the excel file and used it as follows. If we just use this data, the shape is (240, 14). However, the problem uses only 9 features, so the shape should be made (240, 9). Here, in order to use only the specific column desired in the problem, the remaining unnecessary columns were deleted.

```
 file_name = 'nbastat2021.xlsx' # Input file
 nba_data = pd.read_excel(file_name, engine='openpyxl') # Input data with full-columns
 nba_data_9feat = nba_data.drop(['FULL NAME', 'TEAM', 'GP', 'MPG', 'POS'], axis=1) # Input data with pre-processing
```

Listing 1: input file

In this assignment, input and output types were written as comments for all functions. Inputs may be used as a DataFrame for pandas, or as an ndarray for numpy. First of all, for convenience, input data was created only with the DataFrame of Pandas. From now on, we will show how the desired function was implemented in each problem, a description of the code, and the results. For most code descriptions, comments are written in detail. And the data used for this problem is composed as follows.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	FULL NAME	TEAM	GP	MPG	FT%	2P%	3P%	PPGPointsI	RPGRebou	APGAssists	SPGStealsS	BPGBlocksE	TOPGTurnc	POS
2	Precious Achiuwa	Tor	61	23.4	0.589	0.457	0.362	8.6	6.8	1.1	0.52	0.56	1.18	F
3	Steven Adams	Mem	67	26.5	0.544	0.557	0	7.2	9.9	3.2	0.82	0.76	1.55	C
4	Bam Adebayo	Mia	46	32.8	0.754	0.549	0	19	10.3	3.5	1.5	0.78	2.7	F-C
5	LaMarcus Aldridge	Bro	45	22.8	0.873	0.58	0.311	13.5	5.6	0.9	0.31	1.02	0.93	F-C
6	Nickeil Alexander-\	Nor	50	26.3	0.722	0.434	0.311	12.8	3.3	2.8	0.82	0.38	1.7	G
7	Grayson Allen	Mil	59	27.5	0.862	0.506	0.403	11.2	3.4	1.6	0.75	0.25	0.61	G
8	Jarrett Allen	Cle	56	32.3	0.708	0.688	0.1	16.1	10.7	1.6	0.77	1.34	1.68	C
9	Kyle Anderson	Mem	58	21.4	0.607	0.475	0.311	7.3	5.4	2.6	1.12	0.62	0.97	G-F
10	Giannis Antetokou	Mil	59	32.9	0.721	0.61	0.3	29.8	11.5	5.8	1.07	1.36	3.17	F
11	Carmelo Anthony	Lal	62	26.5	0.829	0.514	0.388	13.7	4.2	1	0.71	0.81	0.89	F
12	Cole Anthony	Orl	57	32.6	0.857	0.434	0.34	17.2	5.6	5.7	0.77	0.28	2.74	G
13	OG Anunoby	Tor	42	36.7	0.748	0.506	0.351	17.5	5.5	2.6	1.55	0.57	1.76	F
14	Deni Avdija	Was	70	23.4	0.765	0.522	0.311	7.7	5	1.7	0.74	0.53	0.89	F
15	Deandre Ayton	Pho	50	29.3	0.74	0.654	0.313	17.1	10	1.4	0.72	0.72	1.5	C
16	Marvin Bagley III	Sac	30	21.9	0.745	0.538	0.242	9.3	7.2	0.6	0.3	0.37	0.7	F
17	LaMelo Ball	Cha	64	32.1	0.869	0.457	0.375	19.7	6.8	7.4	1.55	0.44	3.16	G
18	Lonzo Ball	Chi	35	34.6	0.75	0.423	0.423	13	5.4	5.1	1.83	0.89	2.34	G
19	Mo Bamba	Orl	60	25.9	0.817	0.567	0.363	10.3	8	1.1	0.57	1.7	1.08	C
20	Desmond Bane	Mem	67	30	0.895	0.486	0.417	17.8	4.5	2.6	1.13	0.4	1.48	G
21	Harrison Barnes	Sac	67	33.7	0.833	0.514	0.412	16.8	5.8	2.5	0.72	0.19	1.6	F
22	Scottie Barnes	Tor	62	35.7	0.735	0.54	0.314	15.5	7.7	3.4	1.16	0.81	1.9	F
23	RJ Barrett	Nyk	59	34.2	0.713	0.444	0.35	19.5	5.7	2.9	0.58	0.27	2.03	G-F
24	Will Barton	Den	61	31.8	0.798	0.506	0.357	14.7	4.9	3.8	0.8	0.48	1.79	G
25	Nicolas Batum	Lac	52	24.8	0.625	0.606	0.391	8.3	4.3	1.7	0.94	0.71	0.63	G-F
26	Darius Bazley	Okc	65	28	0.691	0.501	0.3	10.6	6.4	1.4	0.86	1.06	1.35	G-F
27	Bradley Beal	Was	40	36	0.833	0.508	0.3	23.2	4.7	6.6	0.9	0.38	3.38	G
28	Malik Beasley	Min	71	25.2	0.776	0.431	0.376	12.2	2.9	1.5	0.51	0.14	0.56	G

Figure 1: input data(nbastat2021.xlsx)

This problem is to write a function that computes Correlation and Covariance for different features. And both of the results are in shape of 9*9 matrix. The code was written based on the following formula.

$$\text{COV}(X,Y) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - E[X])(y_i - E[Y]) \tag{1}$$

$$\text{COR}(X,Y) = \frac{COV(X,Y)}{\sqrt{\text{VAR}(X)\text{VAR}(Y)}} \tag{2}$$

The following is my own code for Covariance and Correlation.

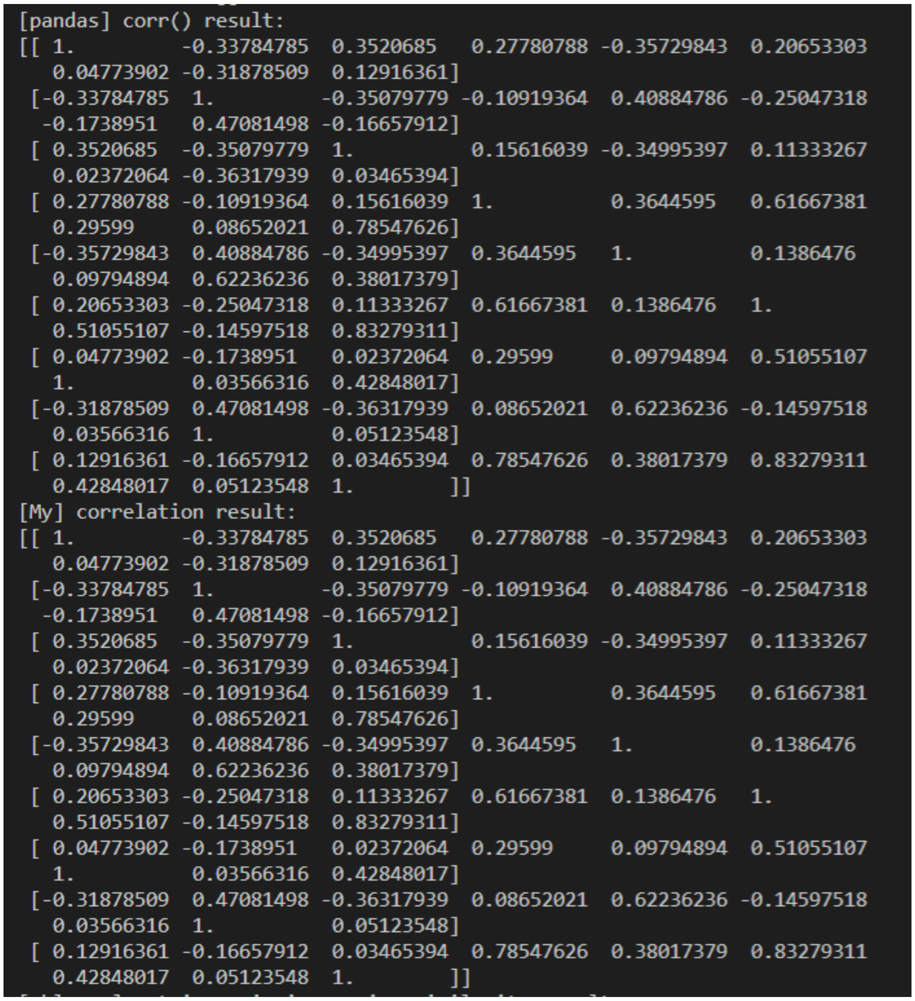
```python
# Problem 1.1
def covariance(X): # Input : X(’pandas.core.frame.DataFrame’) -> Output : output(’numpy.ndarray’)
    X = X.values # Convert dataframe to array
    rows, cols = X.shape # nbastat2021 shape : (240, 9)
    normalization = rows - 1 # Normalization value : n - 1

    output = np.zeros((cols, cols), dtype=np.float64) # Output covariance matrix

    # Construct covariance matrix
    for x in range(cols):
        mean_x = np.mean(X[:,x]) # E[X] : scalar value

        for y in range(cols):
            mean_y = np.mean(X[:,y]) # E[Y] : scalar value
            summation = np.sum((X[:, x] - mean_x) * (X[:, y] - mean_y)) # Summation of (x-E[X])(y-E[Y])
            output[x, y] = summation / normalization # Normalization

    return output

def correlation(X): # Input : X(’pandas.core.frame.DataFrame’) -> Output : output(’numpy.ndarray’)
    cov_X = covariance(X) # Compute covariance matrix

    X = X.values
    rows, cols = cov_X.shape # nbastat2021 covariance matrix shape : (9, 9)

    output = np.zeros((rows, cols), dtype=np.float64) # Output correlation matrix

    # Construct correlation matrix
    for x in range(rows):
        var_x = np.var(X[:,x], ddof=1) # Var(X) : scalar value

        for y in range(cols):
            var_y = np.var(X[:,y], ddof=1) # Var(Y) : scalar value
            normalization = np.sqrt(var_x * var_y) # Normalization value : std(X) * std(Y)
            output[x, y] = cov_X[x, y] / normalization # Normalization

    return output
```

Listing 2: covariance / correlation

In order to check the results of the implementation of covariance and correlation, we compared the results of the already well-made library.

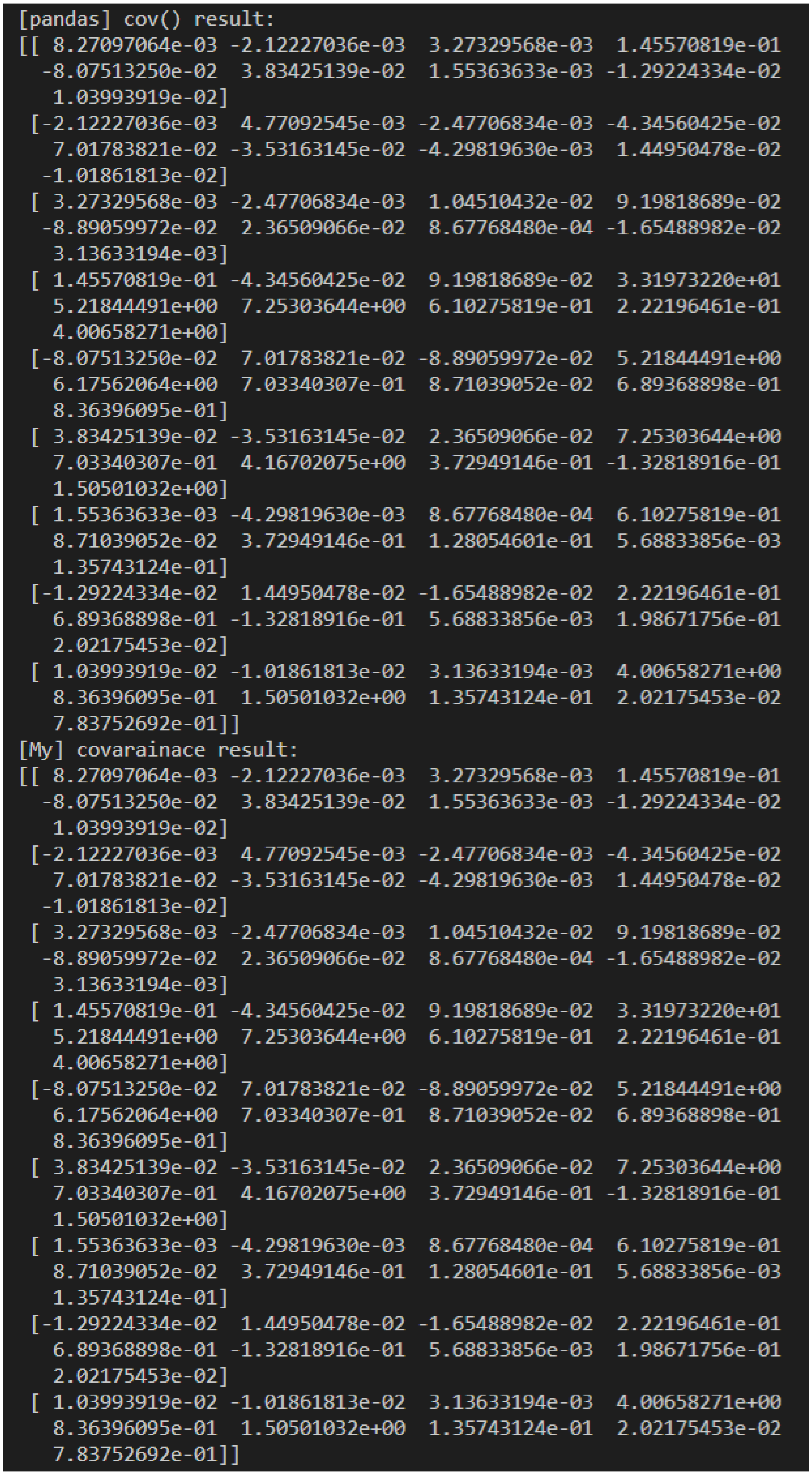
```python
#======================================== Problem 1.1 ========================================
#============= covariance =============
print("[pandas] cov() result:")
print(nba_data_9feat.cov().values)
print("[My] covarainace result:")
print(covariance(nba_data_9feat))
#============= correlation =============
print("[pandas] corr() result:")
print(nba_data_9feat.corr().values)
print("[My] correlation result:")
print(correlation(nba_data_9feat))
```

Listing 3: print the results of covariance / correlation



```
[pandas] corr() result:
[[ 1.         -0.33784785  0.3520685   0.27780788 -0.35729843  0.20653303
   0.04773902 -0.31878509  0.12916361]
 [-0.33784785  1.         -0.35079779 -0.10919364  0.40884786 -0.25047318
  -0.1738951   0.47081498 -0.16657912]
 [ 0.3520685  -0.35079779  1.          0.15616039 -0.34995397  0.11333267
   0.02372064 -0.36317939  0.03465394]
 [ 0.27780788 -0.10919364  0.15616039  1.          0.3644595   0.61667381
   0.29599     0.08652021  0.78547626]
 [-0.35729843  0.40884786 -0.34995397  0.3644595   1.          0.1386476
   0.09794894  0.62236236  0.38017379]
 [ 0.20653303 -0.25047318  0.11333267  0.61667381  0.1386476   1.
   0.51055107 -0.14597518  0.83279311]
 [ 0.04773902 -0.1738951   0.02372064  0.29599     0.09794894  0.51055107
   1.          0.03566316  0.42848017]
 [-0.31878509  0.47081498 -0.36317939  0.08652021  0.62236236 -0.14597518
   0.03566316  1.          0.05123548]
 [ 0.12916361 -0.16657912  0.03465394  0.78547626  0.38017379  0.83279311
   0.42848017  0.05123548  1.        ]]
[My] correlation result:
[[ 1.         -0.33784785  0.3520685   0.27780788 -0.35729843  0.20653303
   0.04773902 -0.31878509  0.12916361]
 [-0.33784785  1.         -0.35079779 -0.10919364  0.40884786 -0.25047318
  -0.1738951   0.47081498 -0.16657912]
 [ 0.3520685  -0.35079779  1.          0.15616039 -0.34995397  0.11333267
   0.02372064 -0.36317939  0.03465394]
 [ 0.27780788 -0.10919364  0.15616039  1.          0.3644595   0.61667381
   0.29599     0.08652021  0.78547626]
 [-0.35729843  0.40884786 -0.34995397  0.3644595   1.          0.1386476
   0.09794894  0.62236236  0.38017379]
 [ 0.20653303 -0.25047318  0.11333267  0.61667381  0.1386476   1.
   0.51055107 -0.14597518  0.83279311]
 [ 0.04773902 -0.1738951   0.02372064  0.29599     0.09794894  0.51055107
   1.          0.03566316  0.42848017]
 [-0.31878509  0.47081498 -0.36317939  0.08652021  0.62236236 -0.14597518
   0.03566316  1.          0.05123548]
 [ 0.12916361 -0.16657912  0.03465394  0.78547626  0.38017379  0.83279311
   0.42848017  0.05123548  1.        ]]
```

Figure 2: correlation result(pandas vs. my)



```
[pandas] cov() result:
[[ 8.27097064e-03 -2.12227036e-03  3.27329568e-03  1.45570819e-01
  -8.07513250e-02  3.83425139e-02  1.55363633e-03 -1.29224334e-02
   1.03993919e-02]
 [-2.12227036e-03  4.77092545e-03 -2.47706834e-03 -4.34560425e-02
   7.01783821e-02 -3.53163145e-02 -4.29819630e-03  1.44950478e-02
  -1.01861813e-02]
 [ 3.27329568e-03 -2.47706834e-03  1.04510432e-02  9.19818689e-02
  -8.89059972e-02  2.36509066e-02  8.67768480e-04 -1.65488982e-02
   3.13633194e-03]
 [ 1.45570819e-01 -4.34560425e-02  9.19818689e-02  3.31973220e+01
   5.21844491e+00  7.25303644e+00  6.10275819e-01  2.22196461e-01
   4.00658271e+00]
 [-8.07513250e-02  7.01783821e-02 -8.89059972e-02  5.21844491e+00
   6.17562064e+00  7.03340307e-01  8.71039052e-02  6.89368898e-01
   8.36396095e-01]
 [ 3.83425139e-02 -3.53163145e-02  2.36509066e-02  7.25303644e+00
   7.03340307e-01  4.16702075e+00  3.72949146e-01 -1.32818916e-01
   1.50501032e+00]
 [ 1.55363633e-03 -4.29819630e-03  8.67768480e-04  6.10275819e-01
   8.71039052e-02  3.72949146e-01  1.28054601e-01  5.68833856e-03
   1.35743124e-01]
 [-1.29224334e-02  1.44950478e-02 -1.65488982e-02  2.22196461e-01
   6.89368898e-01 -1.32818916e-01  5.68833856e-03  1.98671756e-01
   2.02175453e-02]
 [ 1.03993919e-02 -1.01861813e-02  3.13633194e-03  4.00658271e+00
   8.36396095e-01  1.50501032e+00  1.35743124e-01  2.02175453e-02
   7.83752692e-01]]
[My] covarainace result:
[[ 8.27097064e-03 -2.12227036e-03  3.27329568e-03  1.45570819e-01
  -8.07513250e-02  3.83425139e-02  1.55363633e-03 -1.29224334e-02
   1.03993919e-02]
 [-2.12227036e-03  4.77092545e-03 -2.47706834e-03 -4.34560425e-02
   7.01783821e-02 -3.53163145e-02 -4.29819630e-03  1.44950478e-02
  -1.01861813e-02]
 [ 3.27329568e-03 -2.47706834e-03  1.04510432e-02  9.19818689e-02
  -8.89059972e-02  2.36509066e-02  8.67768480e-04 -1.65488982e-02
   3.13633194e-03]
 [ 1.45570819e-01 -4.34560425e-02  9.19818689e-02  3.31973220e+01
   5.21844491e+00  7.25303644e+00  6.10275819e-01  2.22196461e-01
   4.00658271e+00]
 [-8.07513250e-02  7.01783821e-02 -8.89059972e-02  5.21844491e+00
   6.17562064e+00  7.03340307e-01  8.71039052e-02  6.89368898e-01
   8.36396095e-01]
 [ 3.83425139e-02 -3.53163145e-02  2.36509066e-02  7.25303644e+00
   7.03340307e-01  4.16702075e+00  3.72949146e-01 -1.32818916e-01
   1.50501032e+00]
 [ 1.55363633e-03 -4.29819630e-03  8.67768480e-04  6.10275819e-01
   8.71039052e-02  3.72949146e-01  1.28054601e-01  5.68833856e-03
   1.35743124e-01]
 [-1.29224334e-02  1.44950478e-02 -1.65488982e-02  2.22196461e-01
   6.89368898e-01 -1.32818916e-01  5.68833856e-03  1.98671756e-01
   2.02175453e-02]
 [ 1.03993919e-02 -1.01861813e-02  3.13633194e-03  4.00658271e+00
   8.36396095e-01  1.50501032e+00  1.35743124e-01  2.02175453e-02
   7.83752692e-01]]
```

Figure 3: covariance result(pandas vs. my)

This problem is to write a function that computes Cosine similarity, Minkowski distance, Mahalanobis distance. For Minkowski distance, both cases where r=1, 2 and infinity must be implemented. The code was written based on the following formula.

$$\text{Cosine similarity} = \frac{X \cdot Y}{\|X\|\|Y\|} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}} \tag{3}$$

$$\text{Minkowski distance} = \left(\sum_{i=1}^{n} |x_i - y_i|^r\right)^{\frac{1}{r}} \tag{4}$$

$$\text{Mahalanobis distance} = \sqrt{(X - Y)^T \Sigma^{-1} (X - Y)} \tag{5}$$

The following is my own code for Minkowski distance, Cosine similarity, and Mahalanobis distance.

```python
# Problem 1.2
def cosine_similarity(x, y): # Input : x(’pandas.core.series.Series’), y(’pandas.core.series.Series’) -> Output : output(’numpy.float64’)
    x = x.values
    y = y.values

    dot_prod = np.dot(x, y) # X * Y : scalar value
    l2_norm_x = np.sqrt(np.sum(np.square(x))) # ||X|| : scalar value
    l2_norm_y = np.sqrt(np.sum(np.square(y))) # ||Y|| : scalar value
    normalization = l2_norm_x * l2_norm_y # Normalization value : ||X|| * ||Y||
    output = dot_prod / normalization # Normalization

    return output

def minkowski_distance(x, y, r): # Input : x(’pandas.core.series.Series’), y(’pandas.core.series.Series’), r(’int or str’) -> Output : output(’numpy.float64’)
    x = x.values
    y = y.values

    # Minkowski distance with r = infinity
    if r == ’infinity’:
        diff = np.abs(x - y) # Difference : |x - y|
        output = np.max(diff) # Max : max(|x-y|)
    # Minkowski distance with r = 1, 2, ...
    else:
        diff = np.abs(x - y) # Difference : |x - y|
        diff_power_r = np.power(diff, r) # Difference to the power r : |x - y|^r
        summation = np.sum(diff_power_r) # Summation
        reciprocal_r = 1 / r # 1/r
        output = np.power(summation, reciprocal_r) # Summation to the power 1/r

    return output

def mahalanobis_distance(x, y, cov): # Input : x(’pandas.core.series.Series’), y(’pandas.core.series.Series’), cov(’numpy.ndarray’) -> Output : output(’numpy.float64’)
    x = x.values
    y = y.values

    diff = x - y # Difference : x - y
```

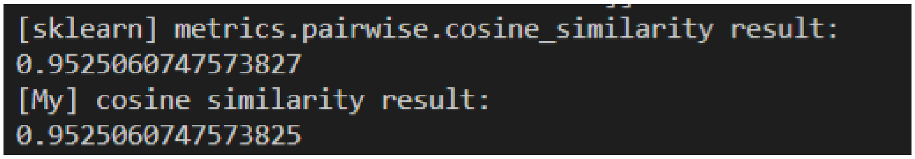
```
diff_transpose = np.transpose(diff)
cov_inverse = np.linalg.inv(cov) # Inverse of covarinace matrix
output = np.sqrt(np.dot(np.dot(diff_transpose,  cov_inverse), diff)) # Multiply difference with inverse of covariance matrix

return output
```

Listing 4: cosine similarity / minkowski distance / mahalanobis distance

In order to check the results of the implementation of cosine similarity, minkowski distance and mahalanobis distance, we compared the results of the already well-made library. Proximities require two samples. So we randomly selected the first sample and the second sample and compared the results.

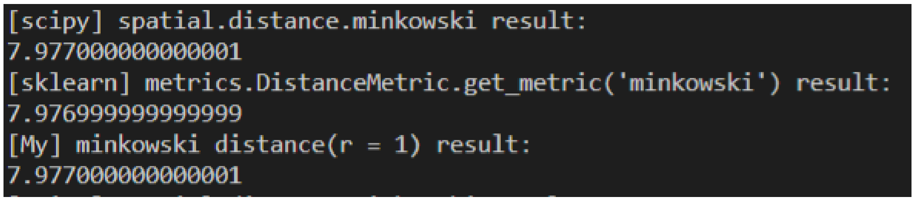
```
#======================================= Problem 1.2 =======================================
x = nba_data_9feat.iloc[0] # One sample from dataset (9, )
y = nba_data_9feat.iloc[1] # One sample from dataset (9, )
#============= cosine similarity =============
print("[sklearn] metrics.pairwise.cosine_similarity result:")
print(metrics.pairwise.cosine_similarity(nba_data_9feat, nba_data_9feat)[0, 1])
print("[My] cosine similarity result:")
print(cosine_similarity(x, y))
#============= minkowski distance(r = 1) =============
print("[scipy] spatial.distance.minkowski result:")
print(spatial.distance.minkowski(x, y, 1))
print("[sklearn] metrics.DistanceMetric.get_metric('minkowski') result:")
print(metrics.DistanceMetric.get_metric('minkowski', p=1).pairwise(nba_data_9feat, nba_data_9feat)[0, 1])
print("[My] minkowski distance(r = 1) result:")
print(minkowski_distance(x, y, 1))
#============= minkowski distance(r = 2) =============
print("[scipy] spatial.distance.minkowski result:")
print(spatial.distance.minkowski(x, y, 2))
print("[sklearn] metrics.DistanceMetric.get_metric('minkowski') result:")
print(metrics.DistanceMetric.get_metric('minkowski', p=2).pairwise(nba_data_9feat, nba_data_9feat)[0, 1])
print("[My] minkowski distance(r = 2) result:")
print(minkowski_distance(x, y, 2))
#============= minkowski distance(r = infinity) =============
print("[scipy] spatial.distance.chebyshev result:")
print(spatial.distance.chebyshev(x, y))
print("[sklearn] metrics.DistanceMetric.get_metric('chebyshev') result:")
print(metrics.DistanceMetric.get_metric('chebyshev').pairwise(nba_data_9feat, nba_data_9feat)[0, 1])
print("[My] minkowski distance(r = 'infinity') result:")
print(minkowski_distance(x, y, 'infinity'))
#============= mahalanobis distance =============
cov_matrix = covariance(nba_data_9feat)
print("[scipy] spatial.distance.mahalanobis result:")
print(spatial.distance.mahalanobis(x, y, np.linalg.inv(cov_matrix)))
print("[sklearn] metrics.DistanceMetric.get_metric('mahalanobis') result:")
print(metrics.DistanceMetric.get_metric('mahalanobis', V=cov_matrix).pairwise(nba_data_9feat, nba_data_9feat)[0, 1])
print("[My] mahalanobis distance result:")
print(mahalanobis_distance(x, y, cov_matrix))
```

Listing 5: print the results of cosine similarity / minkowski distance / mahalanobis distance



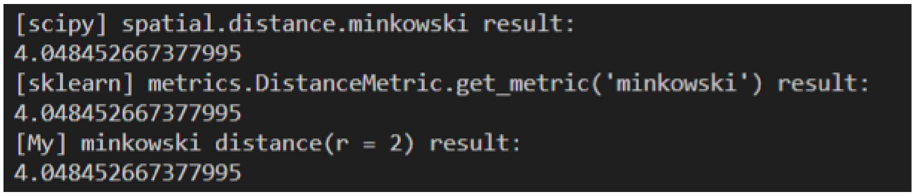
```
[sklearn] metrics.pairwise.cosine_similarity result:
0.9525060747573827
[My] cosine similarity result:
0.9525060747573825
```

Figure 4: cosine similarity result(sklearn vs. my)



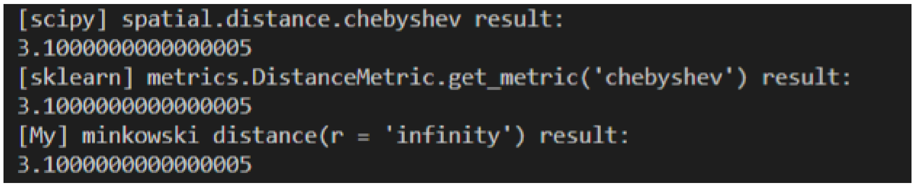
```
[scipy] spatial.distance.minkowski result:
7.977000000000001
[sklearn] metrics.DistanceMetric.get_metric('minkowski') result:
7.976999999999999
[My] minkowski distance(r = 1) result:
7.977000000000001
```

Figure 5: minkowski distance(r=1) result(scipy vs. sklearn vs. my)



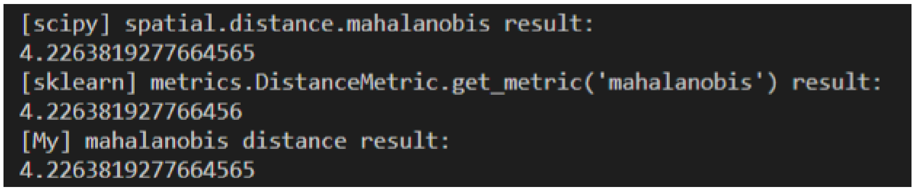
```
[scipy] spatial.distance.minkowski result:
4.048452667377995
[sklearn] metrics.DistanceMetric.get_metric('minkowski') result:
4.048452667377995
[My] minkowski distance(r = 2) result:
4.048452667377995
```

Figure 6: minkowski distance(r=2) result(scipy vs. sklearn vs. my)



```
[scipy] spatial.distance.chebyshev result:
3.1000000000000005
[sklearn] metrics.DistanceMetric.get_metric('chebyshev') result:
3.1000000000000005
[My] minkowski distance(r = 'infinity') result:
3.1000000000000005
```

Figure 7: minkowski distance(r=max) result(scipy vs. sklearn vs. my)



```
[scipy] spatial.distance.mahalanobis result:
4.2263819277664565
[sklearn] metrics.DistanceMetric.get_metric('mahalanobis') result:
4.226381927766456
[My] mahalanobis distance result:
4.2263819277664565
```

Figure 8: mahalanobis distance result(scipy vs. sklearn vs. my)

This problem is to write a function that computes Mutual Information(MI) between X and Y. And we additionally have to compute total entropy of ”Position” and ”Team”. The code was written based on the following formula.

$$\text{Entropy } H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{6}$$

$$\text{Joint Entropy } H(X,Y) = -\sum_{i}\sum_{j} p_{ij} \log_2 p_{ij} \tag{7}$$

$$\text{Mutual Information } = H(X) + H(Y) - H(X,Y) \tag{8}$$

The following is my own code for Entropy, Joint Entropy, and Mutual Information.

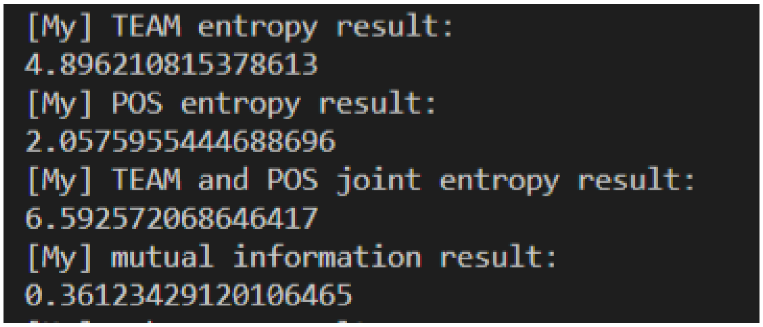
```
# Problem 1.3
def entropy(x): # Input : x(’numpy.ndarray’) -> Output : output(’numpy.float64’)
    counter = {} # Counter of position for each team
    plogp = {} # Results of plogp

    # Count the number of poisition for each team
    for info in x:
        if info in counter:
            counter[info] += 1
        else:
            counter[info] = 1

    # Calculate the result of plogp
    for info, count in counter.items():
        prob = count / len(x)
        plogp[info] = prob * np.log2(prob)

    output = (-1) * sum(plogp.values()) # Entropy

    return output

def joint_entropy(x, y): # Input : x(’numpy.ndarray’), y(’numpy.ndarray’) -> Output : output(’numpy.float64’)
    counter = {}
    plogp = {}

    # Find the number of teams
    counter_x = {}
    for info in x:
        if info in counter:
            counter_x[info] += 1
        else:
            counter_x[info] = 1

    # Initialization of nested dictionary like {’team name’ : {’position’ : # of positions}}
    for team in counter_x.keys():
        counter[team] = {}
        plogp[team] = {}

    xy = np.transpose(np.vstack((x, y)))
    rows, cols = xy.shape

    # Count the number of poisition for each team
    for i in range(rows):
        team = xy[i, 0]
        pos = xy[i, 1]
        if pos in counter[team]:
            counter[team][pos] += 1
        else:
            counter[team][pos] = 1

    # Calculate the result of plogp and their summation
    summation = 0
    for team, position in counter.items():
        for pos, count in position.items():
            prob = count / rows
            plogp[team][pos] = prob * np.log2(prob)
            summation += plogp[team][pos]

    output = (-1) * summation # Joint entropy

    return output

def mutual_information(x, y): # Input : x(’numpy.ndarray’), y(’numpy.ndarray’) -> Output : output(’numpy.float64’)
    entropy_x = entropy(x) # Entropy of x
    entropy_y = entropy(y) # Entropy of y
    entropy_xy = joint_entropy(x, y) # Joint entropy with two vectors

    output = entropy_x + entropy_y - entropy_xy # Mutual information(MI)

    return output
```

Listing 6: entropy / joint entropy / mutual information

In order to check the results of the implementation of entropy, joint entropy, and mutual information, the ”TEAM” and ”POS” columns were used. Two specific columns were extracted from input data and used after changing them to ndarray format.

```
    #======================================= Problem 1.3 =======================================
    team_info = nba_data[["TEAM"]].values.flatten() # Column of "TEAM" converted to numpy.ndarray
    pos_info = nba_data[["POS"]].values.flatten() # Column of "POS" converted to numpy.ndarray
    #============= entropy =============
    print("[My] TEAM entropy result:")
    print(entropy(team_info))
    print("[My] POS entropy result:")
    print(entropy(pos_info))
    print("[My] TEAM and POS joint entropy result:")
    print(joint_entropy(team_info, pos_info))
    #============= mutual information =============
    print("[My] mutual information result:")
    print(mutual_information(team_info, pos_info))
```

Listing 7: print the results of entropy / joint entropy / mutual information

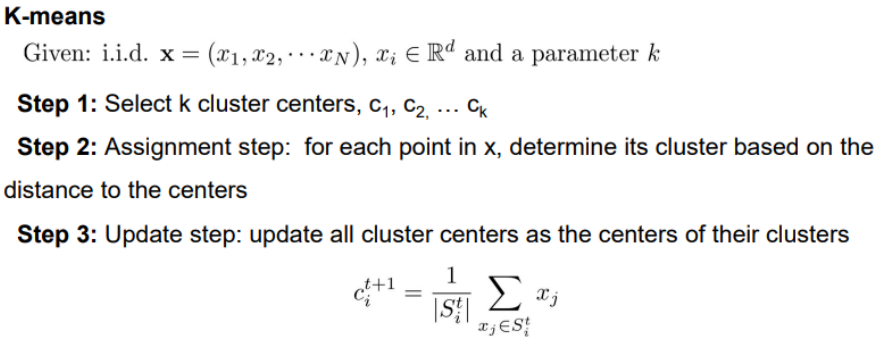


```
[My] TEAM entropy result:
4.896210815378613
[My] POS entropy result:
2.0575955444688696
[My] TEAM and POS joint entropy result:
6.592572068646417
[My] mutual information result:
0.36123429120106465
```

Figure 9: entropy, mutual information result

For the ”TEAM” data, entropy was about 4.89. And for ”POS” data, entropy was about 2.05. Their joint entropy was about 6.59. The Mutual Information (MI) value obtained using these results was about 0.36.

This problem is to implement a function that operates the k-means clustering algorithm. In particular, the function ”mykmeans(X,k,c)” should include cluster data X, cluster count k, and initial center c as input. An important part here is the termination condition in the process of updating the cluster centers. If the L2-norm between the previous cluster centers and the cluster centers to be updated is less than or equal to 0.001, or if the update process has been performed 100 times, the algorithm must be terminated.



K-means

Given: i.i.d. $\mathbf{x} = (x_1, x_2, \cdots x_N)$, $x_i \in \mathbb{R}^d$ and a parameter k

Step 1: Select k cluster centers, c_1, c_2, … c_k

Step 2: Assignment step: for each point in x, determine its cluster based on the distance to the centers

Step 3: Update step: update all cluster centers as the centers of their clusters

$$c_i^{t+1} = \frac{1}{|S_i^t|} \sum_{x_j \in S_i^t} x_j$$

Step 4: Repeat 2 and 3 until converges

The following is my own code for mykmeans. Here, the Assignment step was divided into step 2 and step 3, and the code for Update step was written as step 4.

```python
# Problem 1.4, 1.5
def mykmeans(X, k, c = None): # Input : X('pandas.core.frame.DataFrame'), k('int'), c('numpy.ndarray') -> Output : clusters('numpy.ndarray')
    np.random.seed(0)
    X_df = X.copy()
    X = X.values
    rows, cols = X.shape

    # Step 1 : Initialize cluster centers
    if c is None:
        i = np.random.choice(rows, k, False)
        cluster_centers = X[i,:] # (k, 9)
    else:
        cluster_centers = c

    proximity = 0 # 0: minkowski distance(r = 2) / 1: cosine similarity / 2: mahalanobis distance
    cov = covariance(X_df) # Covaraince matrix for mahalanobis distance
    epoch = 0
    while True:
        # Step 2 : Calculate the distance between each points and cluster centers
        all_distances = np.zeros((rows, k)) # (240, k)
        cluster_centers_df = pd.DataFrame(cluster_centers) # Convert array to dataframe to use my proximities
        for i in range(rows):
            for j in range(k):
                if proximity == 0: # Option 0 : minkowski distance(r = 2)
                    d = minkowski_distance(X_df.iloc[i], cluster_centers_df.iloc[j], 2)
                elif proximity == 1: # Option 1 : cosine similarity -> cosine distance
                    d = 1 - cosine_similarity(X_df.iloc[i], cluster_centers_df.iloc[j])
                elif proximity == 2: # Option 2 : mahalanobis distance
```

```
                    d = mahalanobis_distance(X_df.iloc[i], cluster_centers_df.iloc[j], cov)
                all_distances[i, j] = d

        # Step 3 : Assign each points to the closest cluster center
        clusters = np.zeros((rows, 1)) # Cluster labels : (240, 1)
        min_distances = np.zeros((rows, 1), dtype=np.float64) # Minimum distance : (240, 1)
        for i in range(rows):
            min_index = np.argmin(all_distances[i,:])
            clusters[i, 0] = min_index
            min_distances[i, 0] = all_distances[i, min_index]

        # Step 4 : Update cluster centers
        counter = {}
        for label in clusters:
            label = int(label.item())
            if label in counter:
                counter[label] += 1
            else:
                counter[label] = 1

        # Calculate updated cluster centers
        distance_summation = np.zeros((k, cols), dtype=np.float64) # Summation of x : (k, # of features)
        updated_cluster_centers = np.zeros((k, cols), dtype=np.float64) # Updated cluster centers : (k, # of features)
        for i, label in enumerate(clusters):
            l = int(label.item()) # label(cluster) value : 0, 1, 2, 3, 4
            distance_summation[l] = [a + b for a, b in zip(X[i, :], distance_summation[l])]

        updated_cluster_centers = distance_summation.copy()
        for label, count in counter.items():
            updated_cluster_centers[label] /= count

        # Calculate distances between old cluster centers and updated cluster centers
        center_distances = np.zeros((k, 1), dtype=np.float64) # Distance between old cluster center and updated cluster center : (k, 1)
        updated_cluster_centers_df = pd.DataFrame(updated_cluster_centers) # Convert array to dataframe to use my proximities
        for i in range(k):
            center_distance = minkowski_distance(updated_cluster_centers_df.iloc[i], cluster_centers_df.iloc[i], 2)
            center_distances[i] = center_distance

        # Update cluster centers
        cluster_centers = updated_cluster_centers.copy()

        # Terminate condition
        if max(center_distances).item() <= 0.001:
            break

        epoch += 1
        if epoch == 100:
            break

    # print(updated_cluster_centers)
```

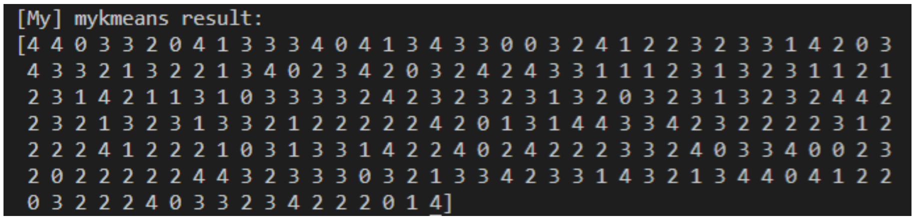
```
clusters = np.ravel(clusters, order='C')
clusters = clusters.astype(np.int32)

return clusters
```

Listing 8: mykmeans

In order to check the results of mykmeans, we had to set the number of clusters "k". In addition, the proximity was configured to be set through a hyper-parameter called "proximity" in the mymeans function for use among Minkowski distances(r=2), Cosine similarity, and Mahalanobis distances. Within the mykmeans function, one of the 3 proximities can be used through values of 0, 1, and 2.

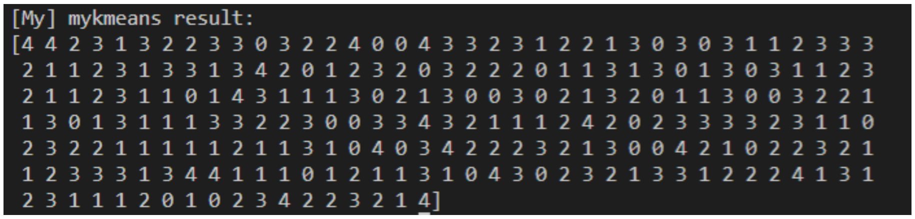
```
#==================================== Problem 1.4, 1.5 ====================================
k = 3 # The number of clusters
mykmeans_result = mykmeans(nba_data_9feat, k) # The result of mykmeans
#============= mykmeans =============
print("[My] mykmeans result:")
print(mykmeans_result)
pos_distribution_per_cluster(pos_info, mykmeans_result) # Distribution of position in each clustsers
compute_accuracy(pos_info, mykmeans_result) # Compute maximum accuracy of my kmeans result
```

Listing 9: print the results of mykmeans



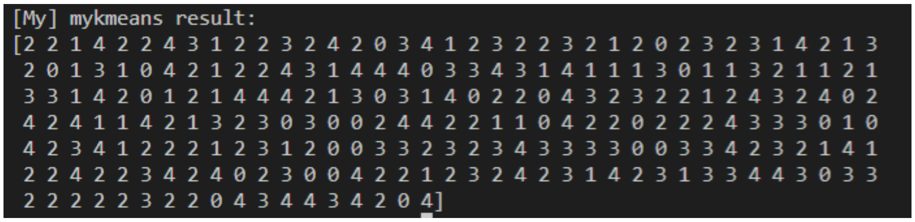
```
[My] mykmeans result:
[4 4 0 3 3 2 0 4 1 3 3 3 4 0 4 1 3 4 3 3 0 0 3 2 4 1 2 2 3 2 3 3 1 4 2 0 3
 4 3 3 2 1 3 2 2 1 3 4 0 2 3 4 2 0 3 2 4 2 4 3 3 1 1 1 2 3 1 3 2 3 1 1 2 1
 2 3 1 4 2 1 1 3 1 0 3 3 3 3 2 4 2 3 2 3 2 3 1 3 2 0 3 2 3 1 3 2 3 2 4 4 2
 2 3 2 1 3 2 3 1 3 3 2 1 2 2 2 2 2 4 2 0 1 3 1 4 4 3 3 4 2 3 2 2 2 2 3 1 2
 2 2 2 4 1 2 2 2 1 0 3 1 3 3 1 4 2 2 4 0 2 4 2 2 2 3 3 2 4 0 3 3 4 0 0 2 3
 2 0 2 2 2 2 2 4 4 3 2 3 3 3 0 3 2 1 3 3 4 2 3 3 1 4 3 2 1 3 4 4 0 4 1 2 2
 0 3 2 2 2 4 0 3 3 2 3 4 2 2 2 0 1 4]
```

Figure 10: kmeans with minkowski(r=2) distance k=5 result



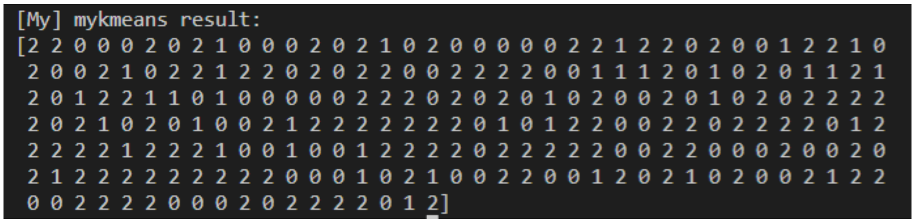
```
[My] mykmeans result:
[4 4 2 3 1 3 2 2 3 3 0 3 2 2 4 0 0 4 3 3 2 3 1 2 2 1 3 0 3 0 3 1 1 2 3 3 3
 2 1 1 2 3 1 3 3 1 3 4 2 0 1 2 3 2 0 3 2 2 2 0 1 1 3 1 3 0 1 3 0 3 1 1 2 3
 2 1 1 2 3 1 1 0 1 4 3 1 1 1 3 0 2 1 3 0 0 3 0 2 1 3 2 0 1 1 3 0 0 3 2 2 1
 1 3 0 1 3 1 1 1 3 3 2 2 3 0 0 3 3 4 3 2 1 1 1 2 4 2 0 2 3 3 3 3 2 3 1 1 0
 2 3 2 2 1 1 1 1 1 2 1 1 3 1 0 4 0 3 4 2 2 2 3 2 1 3 0 0 4 2 1 0 2 2 3 2 1
 1 2 3 3 3 1 3 4 4 1 1 1 0 1 2 1 1 3 1 0 4 3 0 2 3 2 1 3 3 1 2 2 2 4 1 3 1
 2 3 1 1 1 2 0 1 0 2 3 4 2 2 3 2 1 4]
```

Figure 11: kmeans with cosine similarity k=5 result



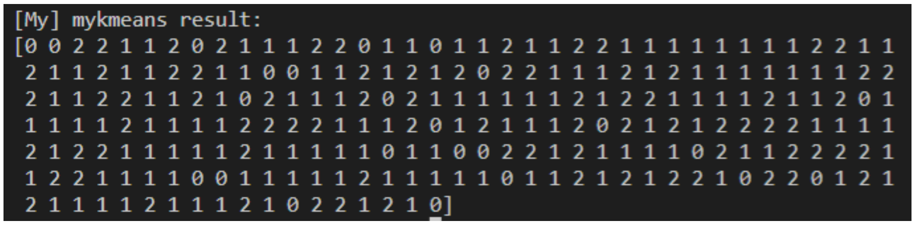
```
[My] mykmeans result:
[2 2 1 4 2 2 4 3 1 2 2 3 2 4 2 0 3 4 1 2 3 2 2 3 2 1 2 0 2 3 2 3 1 4 2 1 3
 2 0 1 3 1 0 4 2 1 2 2 4 3 1 4 4 4 0 3 3 4 3 1 4 1 1 1 3 0 1 1 3 2 1 1 2 1
 3 3 1 4 2 0 1 2 1 4 4 4 2 1 3 0 3 1 4 0 2 2 0 4 3 2 3 2 2 1 2 4 3 2 4 0 2
 4 2 4 1 1 4 2 1 3 2 3 0 3 0 0 2 4 4 2 2 1 1 0 4 2 2 0 2 2 2 4 3 3 3 0 1 0
 4 2 3 4 1 2 2 2 1 2 3 1 2 0 0 3 3 2 3 2 3 4 3 3 3 3 0 0 3 3 4 2 3 2 1 4 1
 2 2 4 2 2 3 4 2 4 0 2 3 0 0 4 2 2 1 2 3 2 4 2 3 1 4 2 3 1 3 3 4 4 3 0 3 3
 2 2 2 2 2 3 2 2 0 4 3 4 4 3 4 2 0 4]
```

Figure 12: kmeans with mahalanobis distance k=5 result



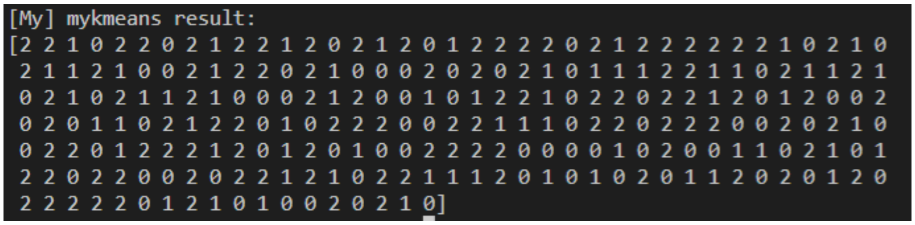
```
[My] mykmeans result:
[2 2 0 0 0 2 0 2 1 0 0 0 2 0 2 1 0 2 0 0 0 0 0 2 2 1 2 2 0 2 0 0 1 2 2 1 0
 2 0 0 2 1 0 2 2 1 2 2 0 2 0 2 2 0 0 2 2 2 2 0 0 1 1 1 2 0 1 0 2 0 1 1 2 1
 2 0 1 2 2 1 1 0 1 0 0 0 0 0 2 2 2 0 2 0 2 0 1 0 2 0 0 2 0 1 0 2 0 2 2 2 2
 2 0 2 1 0 2 0 1 0 0 2 1 2 2 2 2 2 2 2 0 1 0 1 2 2 0 0 2 2 0 2 2 2 2 0 1 2
 2 2 2 2 1 2 2 2 1 0 0 1 0 0 1 2 2 2 2 0 2 2 2 2 2 0 0 2 2 0 0 0 2 0 0 2 0
 2 1 2 2 2 2 2 2 2 2 2 0 0 0 1 0 2 1 0 0 2 2 0 0 1 2 0 2 1 0 2 0 0 2 1 2 2
 0 0 2 2 2 2 0 0 0 2 0 2 2 2 2 0 1 2]
```

Figure 13: kmeans with minkowski(r=2) distance k=3 result



```
[My] mykmeans result:
[0 0 2 2 1 1 2 0 2 1 1 1 2 2 0 1 1 0 1 1 2 1 1 2 2 1 1 1 1 1 1 1 1 2 2 1 1
 2 1 1 2 1 1 2 2 1 1 0 0 1 1 2 1 2 1 2 0 2 2 1 1 1 2 1 2 1 1 1 1 1 1 1 2 2
 2 1 1 2 2 1 1 2 1 0 2 1 1 1 2 0 2 1 1 1 1 1 1 2 1 2 2 1 1 1 1 2 1 1 2 0 1
 1 1 1 1 2 1 1 1 1 2 2 2 2 1 1 1 2 0 1 2 1 1 1 2 0 2 1 2 1 2 2 2 2 1 1 1 1
 2 1 2 2 1 1 1 1 1 2 1 1 1 1 0 1 1 0 0 2 2 1 2 1 1 1 1 0 2 1 1 2 2 2 2 2 1
 1 2 2 1 1 1 1 0 0 1 1 1 1 1 2 1 1 1 1 1 0 1 1 2 1 2 1 2 2 1 0 2 2 0 1 2 1
 2 1 1 1 1 2 1 1 1 2 1 0 2 2 1 2 1 0]
```

Figure 14: kmeans with cosine similarity k=3 result



```
[My] mykmeans result:
[2 2 1 0 2 2 0 2 1 2 2 1 2 0 2 1 2 0 1 2 2 2 2 0 2 1 2 2 2 2 2 2 1 0 2 1 0
 2 1 1 2 1 0 0 2 1 2 2 0 2 1 0 0 0 2 0 2 0 2 1 0 1 1 1 2 2 1 1 0 2 1 1 2 1
 0 2 1 0 2 1 1 2 1 0 0 0 2 1 2 0 0 1 0 1 2 2 1 0 2 2 0 2 2 1 2 0 1 2 0 0 2
 0 2 0 1 1 0 2 1 2 2 0 1 0 2 2 2 0 0 2 2 1 1 1 0 2 2 0 2 2 2 0 0 2 0 2 1 0
 0 2 2 0 1 2 2 2 1 2 0 1 2 0 1 0 0 2 2 2 2 0 0 0 0 1 0 2 0 0 1 1 0 2 1 0 1
 2 2 0 2 2 0 0 2 0 2 2 1 2 1 0 2 2 1 1 1 2 0 1 0 1 0 2 0 1 1 2 0 2 0 1 2 0
 2 2 2 2 2 0 1 2 1 0 1 0 0 2 0 2 1 0]
```

Figure 15: kmeans with mahalanobis distance k=3 result

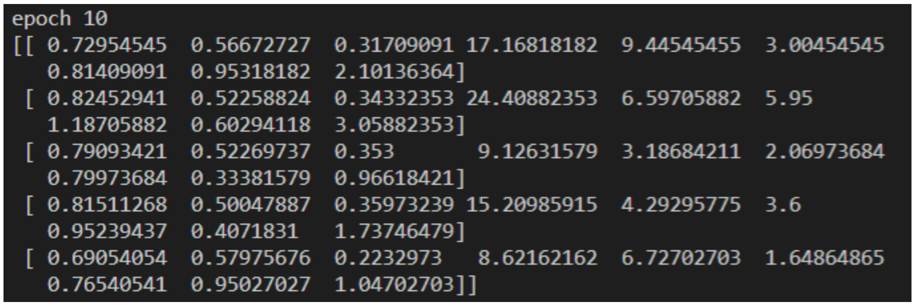
The results of this problem should be analyzed using the mykmeans function and NBA data implemented in the previous problem. In particular, the proximity used in the process of finding cluster centers should be used in both Minkowski distance (r=2), Cosine similarity, and Mahalanobis distance we implemented before.

- How many iterations did it take?

First, for the case of 5 clusters, Minkowski distance was 10, Cosine similarity was 12, and Mahalanobis distance was 10 epochs. In addition, for the case of 3 clusters, Minkowski distance was 7, Cosine similarity was 6, and Mahalanobis distance was 17 epochs. In the case of Minkowski distance and cosine similarity, fewer epochs were shown in the case of fewer clusters, but in the case of Mahalanobis distance, the results were reversed.

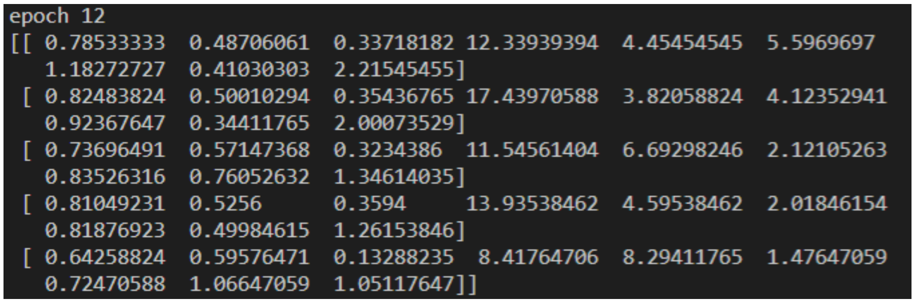
- Are the centers found by different proximities similar or different?

The following is the epoch according to each proximity and the values of the cluster centers in the last updated and terminated state.



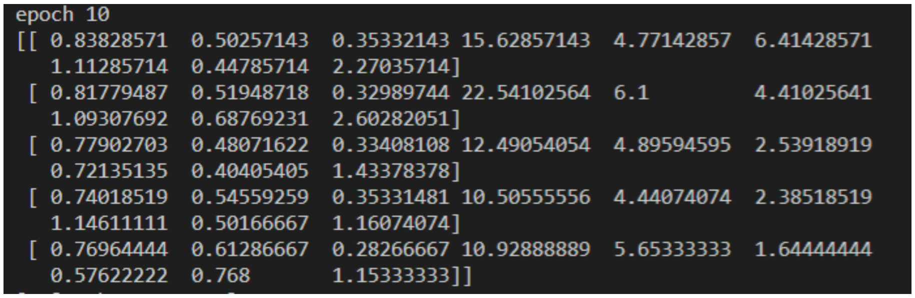
```
epoch 10
[[ 0.72954545  0.56672727  0.31709091 17.16818182  9.44545455  3.00454545
   0.81409091  0.95318182  2.10136364]
 [ 0.82452941  0.52258824  0.34332353 24.40882353  6.59705882  5.95
   1.18705882  0.60294118  3.05882353]
 [ 0.79093421  0.52269737  0.353       9.12631579  3.18684211  2.06973684
   0.79973684  0.33381579  0.96618421]
 [ 0.81511268  0.50047887  0.35973239 15.20985915  4.29295775  3.6
   0.95239437  0.4071831   1.73746479]
 [ 0.69054054  0.57975676  0.2232973   8.62162162  6.72702703  1.64864865
   0.76540541  0.95027027  1.04702703]]
```

Figure 16: epochs and updated centers with minkowski distance(r=2) with 5 clusters



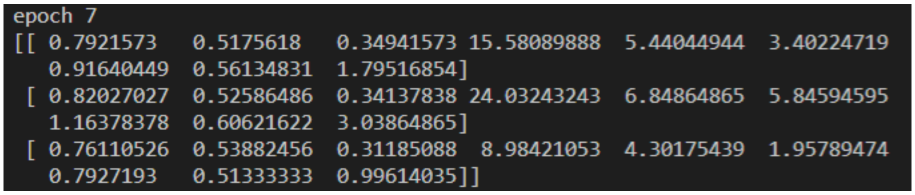
```
epoch 12
[[ 0.78533333  0.48706061  0.33718182 12.33939394  4.45454545  5.5969697
   1.18272727  0.41030303  2.21545455]
 [ 0.82483824  0.50010294  0.35436765 17.43970588  3.82058824  4.12352941
   0.92367647  0.34411765  2.00073529]
 [ 0.73696491  0.57147368  0.3234386  11.54561404  6.69298246  2.12105263
   0.83526316  0.76052632  1.34614035]
 [ 0.81049231  0.5256      0.3594     13.93538462  4.59538462  2.01846154
   0.81876923  0.49984615  1.26153846]
 [ 0.64258824  0.59576471  0.13288235  8.41764706  8.29411765  1.47647059
   0.72470588  1.06647059  1.05117647]]
```

Figure 17: epochs and updated centers with cosine similarity with 5 clusters



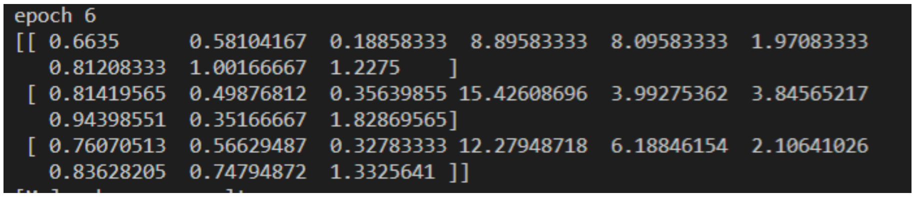
```
epoch 10
[[ 0.83828571  0.50257143  0.35332143 15.62857143  4.77142857  6.41428571
   1.11285714  0.44785714  2.27035714]
 [ 0.81779487  0.51948718  0.32989744 22.54102564  6.1         4.41025641
   1.09307692  0.68769231  2.60282051]
 [ 0.77902703  0.48071622  0.33408108 12.49054054  4.89594595  2.53918919
   0.72135135  0.40405405  1.43378378]
 [ 0.74018519  0.54559259  0.35331481 10.50555556  4.44074074  2.38518519
   1.14611111  0.50166667  1.16074074]
 [ 0.76964444  0.61286667  0.28266667 10.92888889  5.65333333  1.64444444
   0.57622222  0.768       1.15333333]]
```

Figure 18: epochs and updated centers with mahalanobis distance with 5 clusters



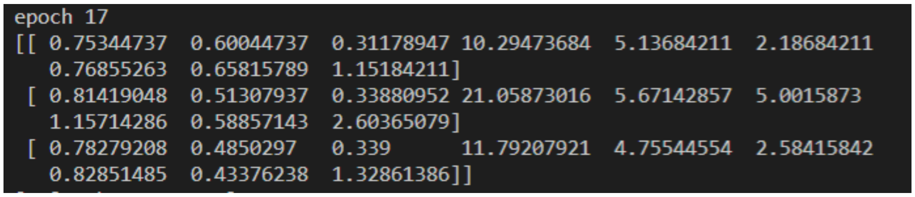
```
epoch 7
[[ 0.7921573   0.5175618   0.34941573 15.58089888  5.44044944  3.40224719
   0.91640449  0.56134831  1.79516854]
 [ 0.82027027  0.52586486  0.34137838 24.03243243  6.84864865  5.84594595
   1.16378378  0.60621622  3.03864865]
 [ 0.76110526  0.53882456  0.31185088  8.98421053  4.30175439  1.95789474
   0.7927193   0.51333333  0.99614035]]
```

Figure 19: epochs and updated centers with minkowski distance(r=2) with 3 clusters



```
epoch 6
[[ 0.6635      0.58104167  0.18858333  8.89583333  8.09583333  1.97083333
   0.81208333  1.00166667  1.2275    ]
 [ 0.81419565  0.49876812  0.35639855 15.42608696  3.99275362  3.84565217
   0.94398551  0.35166667  1.82869565]
 [ 0.76070513  0.56629487  0.32783333 12.27948718  6.18846154  2.10641026
   0.83628205  0.74794872  1.3325641 ]]
```

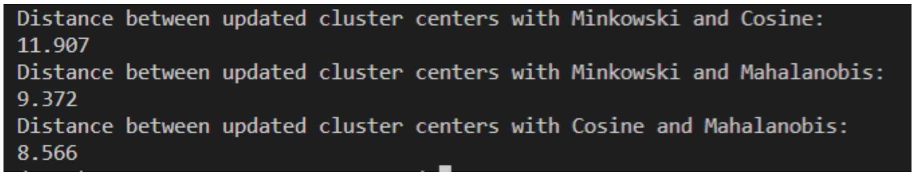
Figure 20: epochs and updated centers with cosine similarity with 3 clusters



```
epoch 17
[[ 0.75344737  0.60044737  0.31178947 10.29473684  5.13684211  2.18684211
   0.76855263  0.65815789  1.15184211]
 [ 0.81419048  0.51307937  0.33880952 21.05873016  5.67142857  5.0015873
   1.15714286  0.58857143  2.60365079]
 [ 0.78279208  0.4850297   0.339      11.79207921  4.75544554  2.58415842
   0.82851485  0.43376238  1.32861386]]
```

Figure 21: epochs and updated centers with mahalanobis distance with 3 clusters

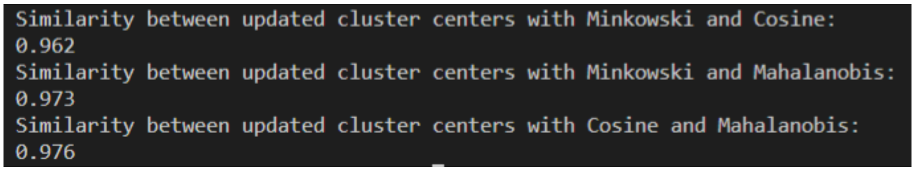
The distance between the final cluster centers obtained using the three methods was calculated.



```
Distance between updated cluster centers with Minkowski and Cosine:
11.907
Distance between updated cluster centers with Minkowski and Mahalanobis:
9.372
Distance between updated cluster centers with Cosine and Mahalanobis:
8.566
```

Figure 22: Distances between updated cluster centers with each proximity

The results showed that the distance between Minkowski and Cosine was 11.907, the distance between Minkowski and Mahalanobis was 9.372, and the distance between Cosine and Mahalanobis was 8.556. It was difficult to determine whether it was clearly similar or not based on the value obtained using Minkowski distance. Comparing the values of cluster centers, they looked quite similar. So this time, we compared the results using cosine similarity.



```
Similarity between updated cluster centers with Minkowski and Cosine:
0.962
Similarity between updated cluster centers with Minkowski and Mahalanobis:
0.973
Similarity between updated cluster centers with Cosine and Mahalanobis:
0.976
```

Figure 23: Similarities between updated cluster centers with each proximity

It can be said that the characteristics of Cosine similarity are similar as it is closer to 1. As a result, we can say that all updated cluster centers are similar because the cosine similarity between cluster centers obtained using three proximities is approximated to 1.

- What is the distribution of positions in each cluster?

Cluster labels were assigned to each sample through mykmeans. However, since there is a label in NBA data, it is possible to compare the cluster label assigned to each sample with the actual position label. So we created the following function to see the number of and the proportion of samples for each cluster that actually have a position.

```python
def pos_distribution_per_cluster(pos, clusters): # Input : pos(’numpy.ndarray’), clusters(’numpy.ndarray)
    num_samples = len(pos) # 240
    num_clusters = max(clusters) + 1 # 3 or 5

    counters = np.zeros((num_clusters, 5), dtype=np.float64)
    p = np.zeros((num_clusters, 5), dtype=np.float64) # (3 or 5, 5)

    # Count the number of positions per each cluster
    for i in range(num_samples):
        if pos[i] == ’G’:
            counters[clusters[i], 0] += 1
        elif pos[i] == ’G-F’:
            counters[clusters[i], 1] += 1
        elif pos[i] == ’F’:
            counters[clusters[i], 2] += 1
        elif pos[i] == ’F-C’:
            counters[clusters[i], 3] += 1
        elif pos[i] == ’C’:
            counters[clusters[i], 4] += 1

    counters_per_clusters = counters.sum(axis=1)

    # Position probability distribution in each cluster
    for i in range(num_clusters):
        for j in range(5):
            p[i, j] = (counters[i, j] / counters_per_clusters[i]) * 100

    # Visualization
    fig = plt.figure(figsize=(20, 12), facecolor=’white’)
    #fig.suptitle("Distribution of positions in each cluster", fontweight ="bold", fontsize = 20)
    fig.suptitle("Probability distribution of positions in each cluster", fontweight ="bold", fontsize = 20)
    for i in range(num_clusters):
        ax=fig.add_subplot(2, 3, i+1)
        p_per_cluster = p[i]
        labels = [’G’, ’G-F’, ’F’, ’F-C’, ’C’]
        #plt.bar(labels, counters[i], color=[’b’, ’orange’, ’g’, ’r’, ’purple’]) #
        wedgeprops={’width’:0.7, ’edgecolor’:’w’, ’linewidth’:2}
        plt.pie(p_per_cluster, labels=labels, autopct=’%.1f%%’, startangle=90, counterclock=False, wedgeprops=wedgeprops)
        ax.set_title(f"cluster {i}", fontsize = 15)
    plt.show()
```

Listing 10: distribution of positions in each clusters

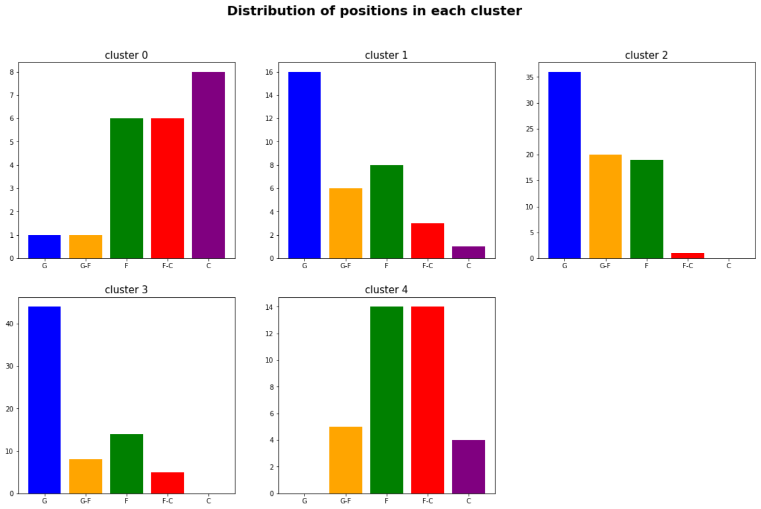


Figure 24: distribution of positions in each cluster(k=5) with minkowski distance(r=2)

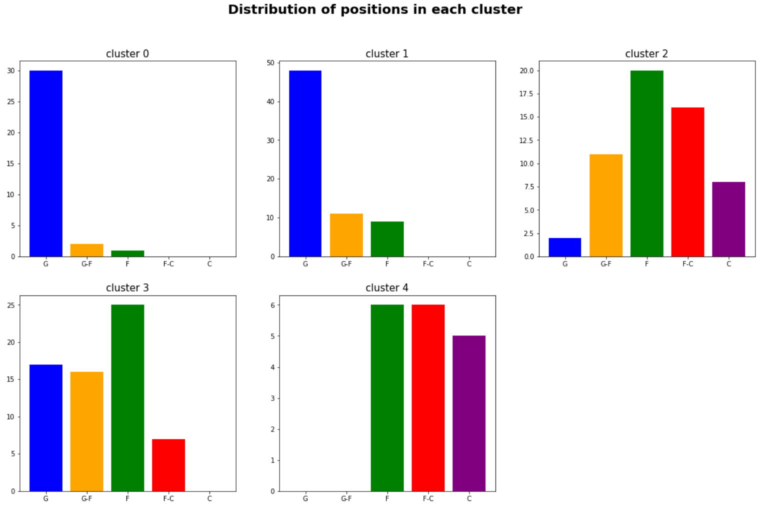


Figure 25: distribution of positions in each cluster(k=5) with cosine similarity

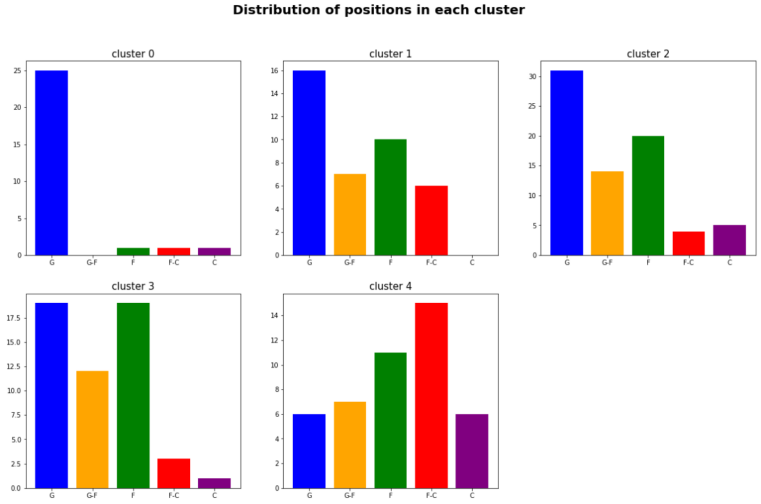


Figure 26: distribution of positions in each cluster(k=5) with mahalanobis distance

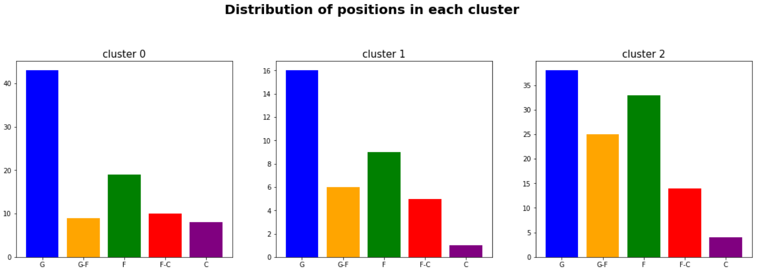


Figure 27: distribution of positions in each cluster(k=3) with minkowski distance(r=2)

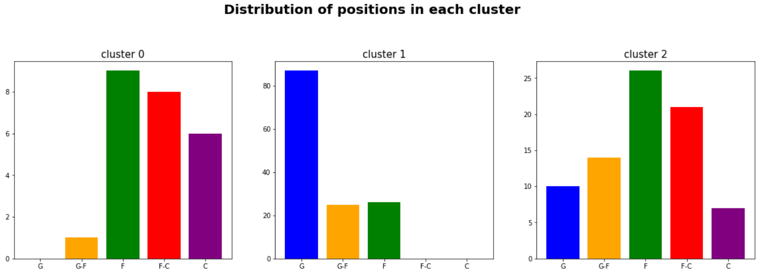


Figure 28: distribution of positions in each cluster(k=3) with cosine similarity

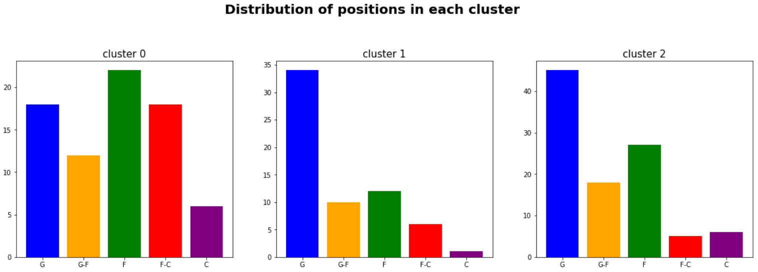


Figure 29: distribution of positions in each cluster(k=3) with mahalanobis distance

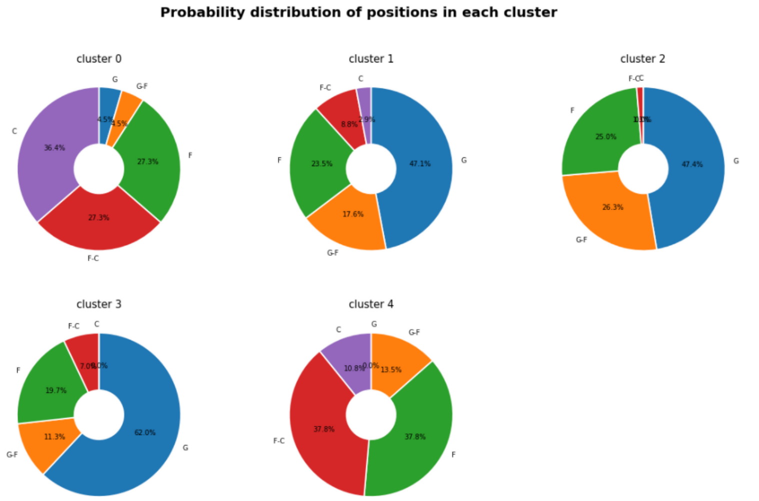


Figure 30: probability distribution of positions in each cluster(k=5) with minkowski distance(r=2)

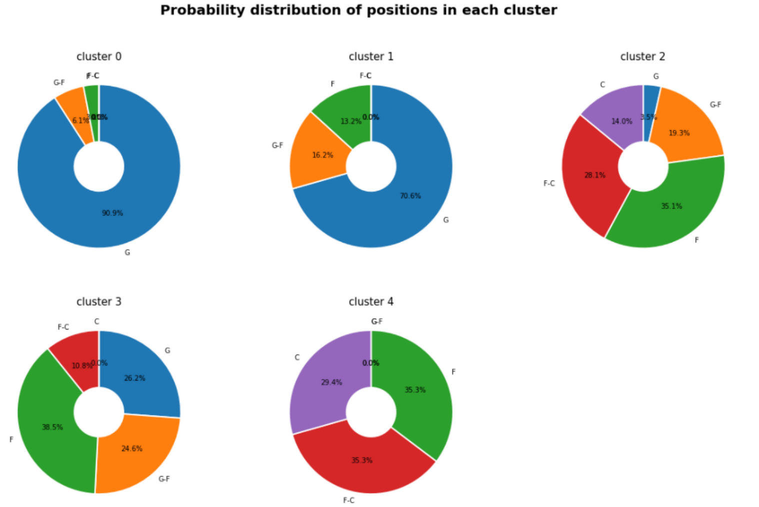


Figure 31: probability distribution of positions in each cluster(k=5) with cosine similarity

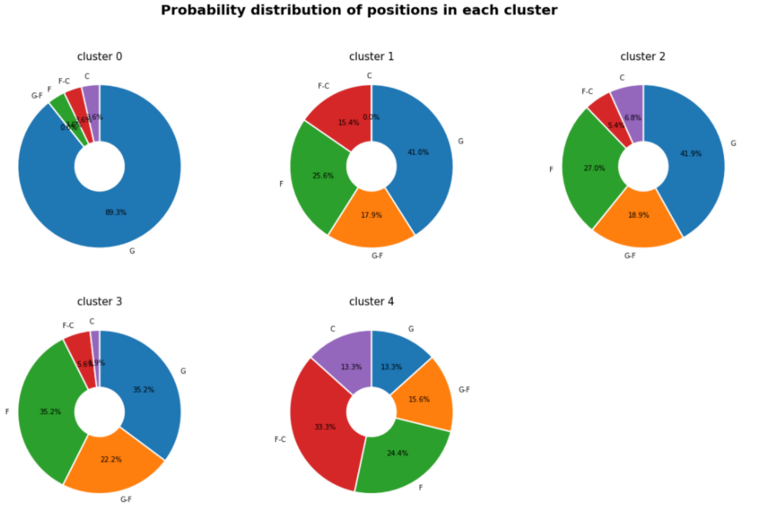


Figure 32: probability distribution of positions in each cluster(k=5) with mahalanobis distance

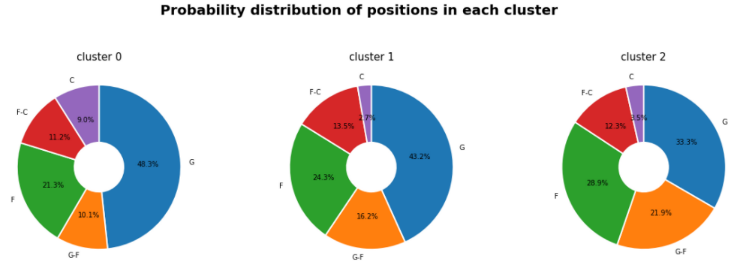


Figure 33: probability distribution of positions in each cluster(k=3) with minkowski distance(r=2)

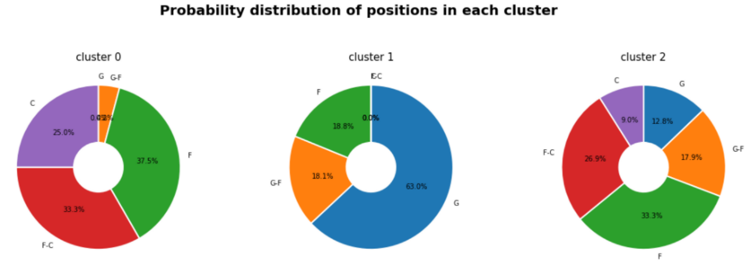


Figure 34: probability distribution of positions in each cluster(k=3) with cosine similarity

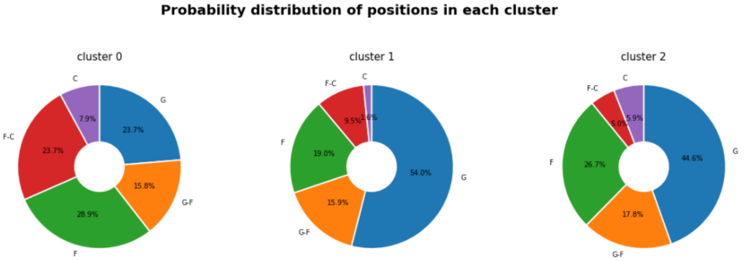


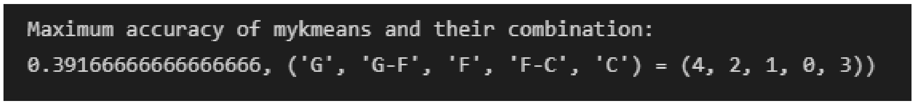
Figure 35: probability distribution of positions in each cluster(k=3) with mahalanobis distance

- Given that there are 5 different positions, what is the accuracy of the clustering on the position when k = 5?

In the case of kmeans clustering, it is difficult to obtain accuracy because it is unsupervised learning. In fact, since there is an actual label in this data, it is necessary to randomly assign a position label to the cluster in order to obtain the accuracy. Therefore, the dominant position for each cluster can be arbitrarily assumed and then the accuracy can be obtained. For more convenience, a total of 120 combinations can be obtained, so we would like to obtain the actual position accruacy for all combinations of 5 clusters and then adopt the highest result as accuracy.

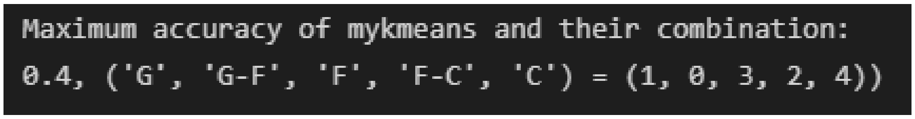
```python
def compute_accuracy(pos, clusters): # Input : pos('numpy.ndarray'), clusters('numpy.ndarray)
    num_samples = len(pos) # 240
    num_clusters = max(clusters) + 1 # 3 or 5

    counters = np.zeros((num_clusters, 5), dtype=np.float64)
    p = np.zeros((num_clusters, 5), dtype=np.float64) # (3 or 5, 5)

    # Count the number of positions per each cluster
    for i in range(num_samples):
        if pos[i] == 'G':
            counters[clusters[i], 0] += 1
        elif pos[i] == 'G-F':
            counters[clusters[i], 1] += 1
        elif pos[i] == 'F':
            counters[clusters[i], 2] += 1
        elif pos[i] == 'F-C':
            counters[clusters[i], 3] += 1
        elif pos[i] == 'C':
            counters[clusters[i], 4] += 1

    accuracy = 0
    combination = 0

    # Compute the accuracy
    for i, tup in enumerate(permutations([0, 1, 2, 3, 4], 5)):
        now = 0
        for j in range(num_clusters):
            now += counters[j][tup[j]]

        # Find the maximum accuracy
        if accuracy < (now / 240):
            accuracy = now / 240
            combination = tup

    print("Maximum accuracy of mykmeans and their combination:")
    print(f"{accuracy}, ('G', 'G-F', 'F', 'F-C', 'C') = {combination})")
```

Listing 11: accuracy of the clustering on the position



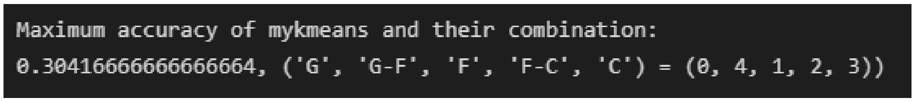
```
Maximum accuracy of mykmeans and their combination:
0.39166666666666666, ('G', 'G-F', 'F', 'F-C', 'C') = (4, 2, 1, 0, 3))
```

Figure 36: Accuracy of Minkowski(r=2) distance with 5 clusters



```
Maximum accuracy of mykmeans and their combination:
0.4, ('G', 'G-F', 'F', 'F-C', 'C') = (1, 0, 3, 2, 4))
```

Figure 37: Accuracy of Cosine similarity with 5 clusters



```
Maximum accuracy of mykmeans and their combination:
0.30416666666666664, ('G', 'G-F', 'F', 'F-C', 'C') = (0, 4, 1, 2, 3))
```

Figure 38: Accuracy of Mahalanobis distance with 5 clusters

The highest measured accuracy in k-means clustering obtained using three distance measures was 39.16%, 40%, and 30.41%. When obtaining the highest accuracy according to each proximity, it can be seen that the result is between 30% and 40% as described above.

- Can the positions be clustered with k-means? Explain.

I think position can be clustered with k-means clustering, but I don't think the accuracy is high. As we can see from the results so far, it is difficult to say that the accuracy is high. So, it seems better to classify positions using a better classification method. Clusters can be divided through k-means clustering, but it is not possible to connect which cluster is the corresponding position. Therefore, even if the highest accuracy is obtained through all permutations, only about 30-40% is measured. The problem with K-means clustering is that the center is initially randomly determined, so the results may vary a lot depending on this process. However, in the case of k-means clustering, it is easy and has a characteristic that the operation is fast. And convergence is guaranteed. Even so, it cannot be guaranteed that a global optimal has been found. Still, the result can be close to global optimal, and if the number of clusters is set well, I think clustering can be done sufficiently.

This problem requires the implementation of ”mykde(X,h)” function that performs kernel density estimation(KDE) on data X with bandwidth h. This function should return the estimated density p(x) and its domain x where we estimated the p(x) for X in 1-D and 2-D. The code was written based on the following formula.

$$\text{Kernel function } k(u) = \begin{cases} 1 & |u_i| \leq \frac{1}{2}, \quad i = 1, \dots, d \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

$$K = \sum_{i=1}^{N} k(\frac{x - x_i}{h}) \tag{10}$$

$$\text{Estimated density } p(x) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{h^d} k(\frac{x - x_i}{h}) \tag{11}$$

The following is my own code for kernel functions and mykde. The kernel function was constructed using Parzen window to count the number of samples in a specific range based on domain x. Using the mykde function returns the domain ”x” and estimated density ”p(x)” as output. Here, in the case of domain x, 100 bins per dimension are set to be used. In particular, in the process of plotting in 1D and 2D, we tried to set the domain more flexibly according to bandwidth h. In addition, according to the dimension of input data, both 1D KDE and 2D KDE can be performed with one mykde function.

```
# Problem 2.1, 2.2
def kernel_function_1d(x): # Input : x(’numpy.float64’) -> Output : ’int’
    # Parzen window : a kernel function
    if np.abs(x) <= 0.5:
        return 1
    else:
        return 0

def kernel_function_2d(x, y): # Input : x(’numpy.float64’), y(’numpy.float64’) -> Output : ’int’
    # Parzen window : a kernel function
    if np.abs(x) <= 0.5 and np.abs(y) <= 0.5:
        return 1
    else:
        return 0

def mykde(X, h): # Input : X(’numpy.ndarray’), h(’float or int’) -> Output : p(’numpy.ndarray’), x(’numpy.ndarray’)
    d = X.ndim

    # 1D kernel density estimation(KDE)
    if d == 1:
        N = X.shape[0] # The number of data : 1000
        nbins = 100 # The number of bins

        X_min = min(X) # Min value of X
        X_max = max(X) # Max value of X

        # Update min / max value with h value
        X_min_new = X_min - (0.5 * h)
        X_max_new = X_max + (0.5 * h)

        # Select x
        x = np.random.uniform(X_min_new, X_max_new, nbins) # (100, )
        x = np.sort(x)

        kernels = np.zeros((nbins, N), dtype=np.float64) # (100, 1000)
        K = np.zeros((nbins,), dtype=np.float64) # (100, )
        p = np.zeros((nbins,), dtype=np.float64) # (100, )
        rows, cols = kernels.shape

        # Total # of data points in the cube(side = h) centered at x using Parzen window
        for i in range(rows):
            x_center = x[i]
            for j in range(cols):
                kernel_function_input = (x_center - X[j]) / h
                kernel_function_output = kernel_function_1d(kernel_function_input) # 0 or 1
                kernels[i, j] = kernel_function_output

        K = kernels.sum(axis = 1)

        # Calculate estimated density p(x)
        for i in range(nbins):
            p[i] = K[i] / (N * (h ** d))

        return [p, x]

    # 2D kernel density estimation(KDE)
    elif d == 2:
        N = X.shape[0] # # The number of data : 500
        nbins = 10 # The number of bins

        X_1 = X[:, 0] # (500, 1)
        X_2 = X[:, 1] # (500, 1)

        X_1_min = min(X_1) # Min value of X1
        X_1_max = max(X_1) # Max value of X1
        X_2_min = min(X_2) # Min value of X2
        X_2_max = max(X_2) # Max value of X2

        # Update min / max value with h value
        X_1_min_new = X_1_min - (0.5 * h)
        X_1_max_new = X_1_max + (0.5 * h)
        X_2_min_new = X_2_min - (0.5 * h)
        X_2_max_new = X_2_max + (0.5 * h)

        # Select x
        x = np.mgrid[X_1_min_new : X_1_max_new : 10j, X_2_min_new : X_2_max_new : 10j] # (2, 10, 10)

        kernels = np.zeros((N, nbins, nbins), dtype=np.float64) # (500, 10, 10)
        K = np.zeros((nbins, nbins), dtype=np.float64) # (10, 10)
        p = np.zeros((nbins, nbins), dtype=np.float64) # (10, 10)

        # Total # of data points in the cube(side = h) centered at x using Parzen window
        for i in range(nbins):
            for j in range(nbins):
                x1_center = x[0, j, i]
                x2_center = x[1, j, i]
                for k in range(N):
                    kernel_function_input_1 = (x1_center - X_1[k]) / h
                    kernel_function_input_2 = (x2_center - X_2[k]) / h
                    kernel_function_output = kernel_function_2d(
  kernel_function_input_1, kernel_function_input_2) # 0 or 1
                    kernels[k, i, j] = kernel_function_output

        K = kernels.sum(axis = 0)

        # Calculate estimated density p(x)
        for i in range(nbins):
            for j in range(nbins):
                p[i, j] = K[i, j] / (N * (h ** d))

        return [p, x]
```

Listing 12: kernel functions in 1-D and 2-D / mykde

And in order to visualize the results of mykde function in one-dimensional, a visualization function was implemented as follows. Visualized results include the histogram of X along with the figures of estimated densities p(x).

```
# Visualize KDE in 1 dimension
def visualization_KDE_1D(X, h): # Input : X('numpy.ndarray'), h('list') -> Output
   : visualization result
    case_num = len(h)

    fig = plt.figure(figsize=(20, 10))
    fig.suptitle("1D Kernel Density Estimation", fontweight ="bold", fontsize =
   20)

    for i in range(case_num):
        [p, x] = mykde(X, h[i]) # My kde function
        ax = fig.add_subplot(2, 3, i+1)
        plt.hist(X, bins=100, density=True, color = 'paleturquoise', label="Data")
        plt.plot(x, p, color='g', label="KDE")
        ax.legend()
        ax.set_title(f"h = {h[i]}", fontsize = 15)
        ax.set_xlabel("x")
        ax.set_ylabel("p(x)")
    plt.show()
```

Listing 13: visualization mykde with 1-D data

The data described in the problem were generated as follows and the results were confirmed using the mykde function in the visualization function.

```
    #======================================= Problem 2.1
   =======================================
    X1 = np.random.normal(5, 1, 1000) # case 1 data : mu=5, cov=1
    X2 = np.random.normal(1, 0.3, 1000)
    X3 = np.concatenate((X1, X2)) # case 2 data : mu=5, cov=1 + mu=1, cov=0.3
    h1 = [0.1, 0.5, 1, 5, 10, 20] # bandwidth h

    visualization_KDE_1D(X1, h1) # case 1
    visualization_KDE_1D(X3, h1) # case 2
```

Listing 14: print the visualization mykde with 1-D data

Case 1 : N $= 1000$, $\mu = 5$, $\sigma = 1$

Case 2 : N $= 1000$, $\mu = 5$, $\sigma = 1 + \mu = 1$, $\sigma = 0.3$

h $= \{0.1, 0.5, 1, 5, 10, 20\}$

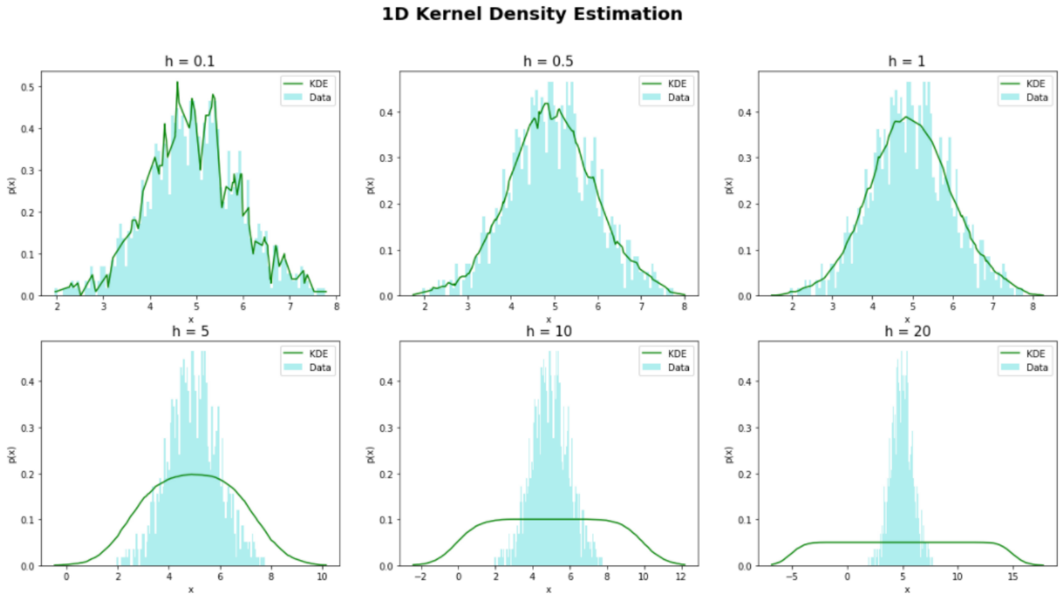


Figure 39: 1D kernel density estimation visualization (case 1)

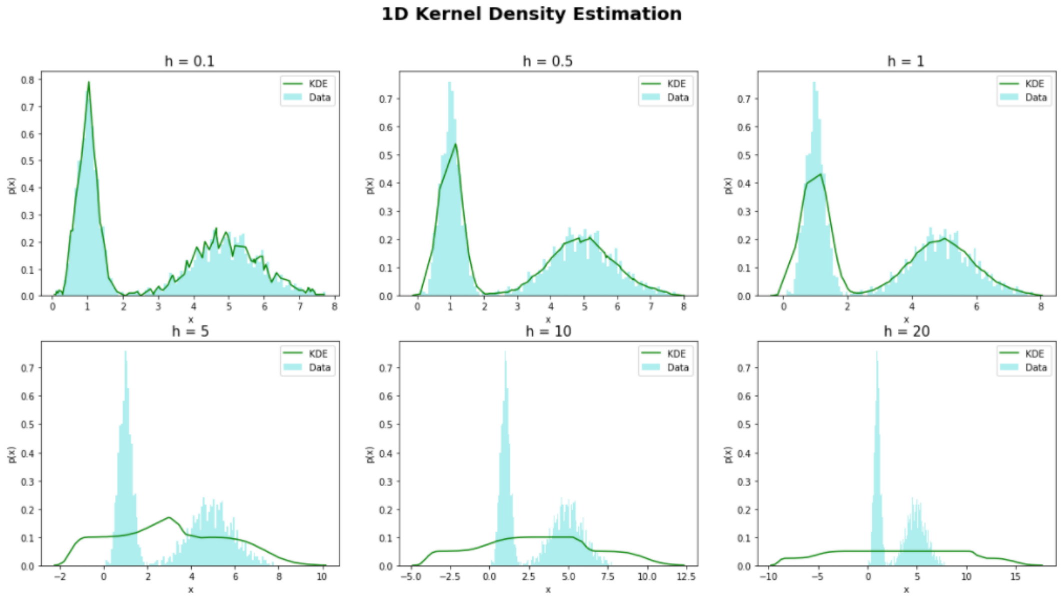


Figure 40: 1D kernel density estimation visualization (case 2)

This problem is to apply the mykde function implemented in the previous problem to two-dimensional data. In particular, the difference in bandwidth should be compared for a total of four cases, and in order to visualize the results of mykde function in two dimensions, a visualization function using a contour plot was implemented as follows.

```python
# Visualize KDE in 2 dimension using contour plot
def visualization_KDE_2D(X, h): # Input : X('numpy.ndarray'), h('list') -> Output : visualization result
    case_num = len(h)

    X_1_min = min(X[:, 0])
    X_1_max = max(X[:, 0])
    X_2_min = min(X[:, 1])
    X_2_max = max(X[:, 1])

    fig = plt.figure(figsize=(20, 12))
    fig.suptitle("2D Kernel Density Estimation", fontweight ="bold", fontsize = 20)

    for i in range(case_num):
        X_1_min_new = X_1_min - (0.5 * h[i])
        X_1_max_new = X_1_max + (0.5 * h[i])
        X_2_min_new = X_2_min - (0.5 * h[i])
        X_2_max_new = X_2_max + (0.5 * h[i])
        [p, x] = mykde(X, h[i]) # My kde function
        ax = fig.add_subplot(2, 2, i+1)
        ax.set_xlim(X_1_min_new, X_1_max_new)
        ax.set_ylim(X_2_min_new, X_2_max_new)
        ax.imshow(np.rot90(p), cmap='coolwarm', extent=[X_1_min_new, X_1_max_new, X_2_min_new, X_2_max_new])
        cset = ax.contour(x[0], x[1], p, colors='k')
        cfset = ax.contourf(x[0], x[1], p, cmap='coolwarm')
        ax.clabel(cfset, fontsize=8, colors='black')
        fig.colorbar(cfset, aspect=5, ax=ax)
        ax.set_xlabel('x1')
        ax.set_ylabel('x2')
        ax.set_title(f"h = {h[i]}", fontsize = 15)
```

Listing 15: visualization mykde with 2-D data

The data described in the problem were generated as follows and the results were confirmed using the mykde function in the visualization function.

```python
    #======================================= Problem 2.2 =======================================
    Y1_mean = [1, 0]
    Y1_cov = [[0.9, 0.4], [0.4, 0.9]]
    Y1 = np.random.multivariate_normal(Y1_mean, Y1_cov, 500) # case 1 data
    Y2_mean = [0, 2.5]
    Y2_cov = [[0.7, 0.4], [0.4, 0.3]]
    Y2 = np.random.multivariate_normal(Y2_mean, Y2_cov, 500) # case 2 data
    h2 = [0.1, 1, 5, 10] # bandwidth h

    visualization_KDE_2D(Y1, h2) # case 1
    visualization_KDE_2D(Y2, h2) # case 2
```

Listing 16: print the visualization mykde with 2-D data

Case 1 : N = 500, μ = [1, 0], σ = [[0.9, 0.4], [0.4, 0.9]]

Case 2 : N = 500, μ = [0, 2.5], σ = [[0.7, 0.4], [0.4, 0.3]]

h = {0.1, 1, 5, 10}

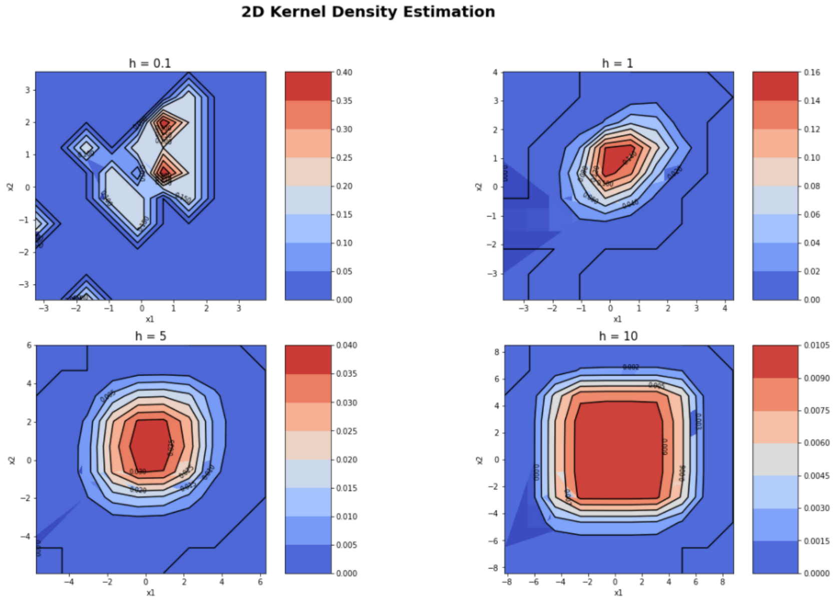


Figure 41: 2D kernel density estimation visualization (case 1)

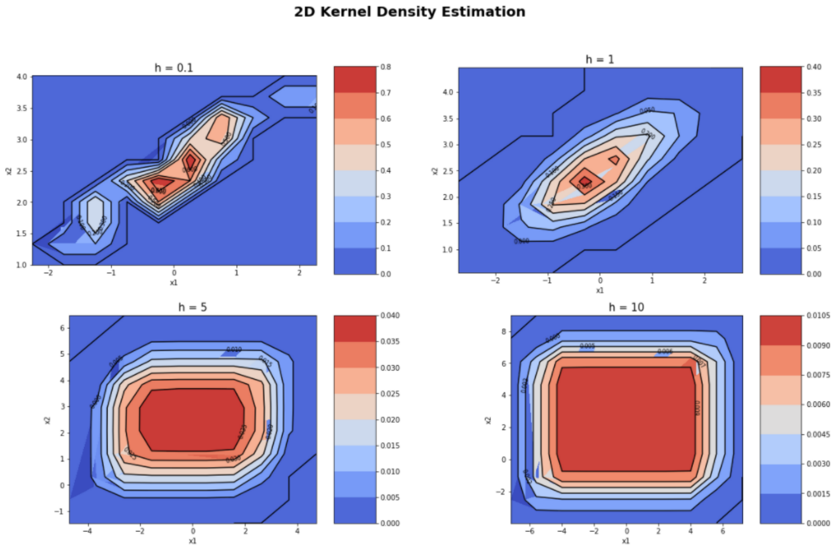


Figure 42: 2D kernel density estimation visualization (case 2)

Since the shape of the graph changes depending on the bandwidth ”h”, it is necessary to set the appropriate bandwidth. If the bandwidth becomes too small, a rapid change in data occurs. Conversely, if the bandwidth becomes too large, it becomes a smooth graph, making it difficult to reflect changes in data. Looking at the above 1D and 2D results, the smaller the h value, the sharper and more data-reflecting graph is obtained, and the larger the h value, the smoother the result was obtained.