20127214-
20127612

PARALLELIZE KMEANS CREDIT CARD SEGMENTATION

CSC14116 - APPLIED PARALLEL PROGRAMMING

CONTENT

OUR TEAM

20127214

Nguyễn Trương Minh Khôi

20127612

Đỗ Khánh Sang

01

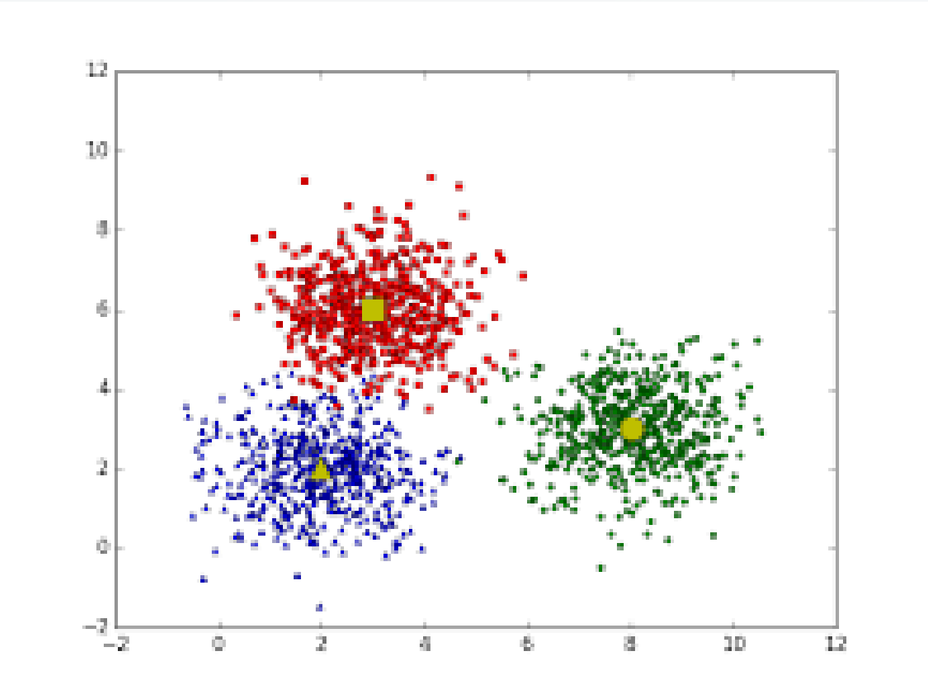
INTRODUCTION

About the project

INTRODUCTION

- Working on KMeans clustering algorithms to improve its performance and efficiency

- Using GPU computing and parallel processing, the model will be able to handle larger datasets and require less training time



INTRODUCTION

Python

Used to implement KMeans as well as paralyze sequent code

Numba

Used to optimized machine code at runtime

INTRODUCTION

Tasks

calculating distance between each data point, assigning data points to proper clusters and updating new centroids

PCA

Use PCA to reduce the dimension of clusters since the dataset has many attributes

Run time

Compare running time and the accuracy of each version of KMeans such as sequent code, scikit-learn code and paralyzed code

02

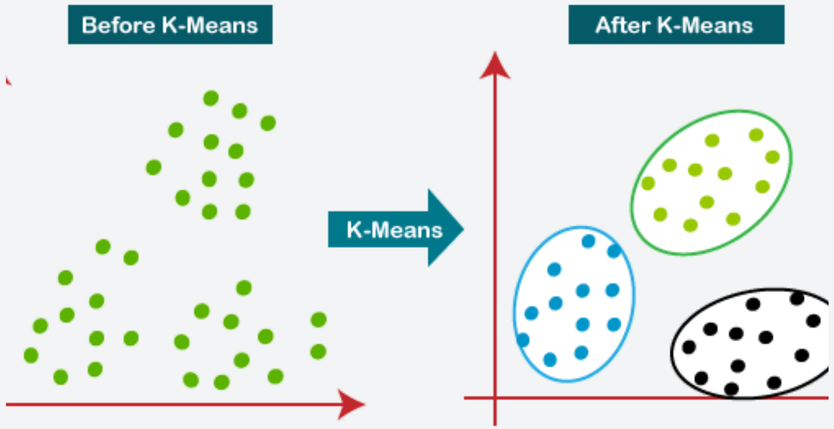
BACKGROUND

About dataset and KMeans

PROBLEM

Credit card

Credit card segmentation is a strategic approach that involves dividing a customer base into distinct groups based on various criteria



KMeans

To KMeans it means grouping customers based on their transaction behaviors to identify patterns and trends

03

COMPLETED WORK

About KMeans and PCA

KMEANS

Complete sequential version of KMeans Algorithm

- Process:
  1. Generate initinal centroid by KMeans++.
  2. Assign points to their cluster.
  3. Update new centroids.
  4. Loops until converge.

APPLIED KMEANS++

Sequential code:

Using **kmeans_plusplus** library from **Scikit-learn:**

- normalized data
- number of clusters
- random number generation for centroid initialization

```
kmeans_plusplus(df_Standardize.to_numpy(), n_clusters=3, random_state=0)
```

APPLIED KMEANS++

Centers given by KMeans++:

```
array([[-0.65944319,  0.50448036, -0.51712247, -0.39742169, -0.48008007,
        -0.32039306, -1.16547046, -0.68237582, -0.88602072,  0.11298066,
        -0.03346818, -0.60451057, -1.16155939, -0.49812349, -0.29928689,
        -0.26828014,  0.50817915],
       [ 0.16110378,  0.49321598,  1.361444  , -0.12652727,  3.3104138 ,
        -0.47663128,  1.25415311,  1.17944845,  1.63933249, -0.63490638,
        -0.48356644,  1.96376081,  0.63010485,  1.23065799, -0.19421358,
        -0.58585736,  0.5177819 ],
       [-0.13201319,  0.50226787,  0.71755277,  1.07425348, -0.25839649,
        -0.40421331,  1.29242485,  0.95745806,  1.22272092,  0.10047235,
        -0.18329416,  0.55556617, -0.54567534, -0.1805813 , -0.2337163 ,
         0.01292614,  0.51453195]])
```

KMEANS

Sequential code:

Using pure Python to implement KMeans algorithm

```
Processing time: 79.65466332435608 s
Processing time of Calculating distance and Assigning label: 66.51338958740234 s
Processing time of Updating centers: 13.140365600585938 s
Processing time of Checking convergence: 0.0003476142883300781 s
Number of loops: 34
Centers found by k-means algorithm:
[array([ 1.19168987,  0.34421507, -0.32010396, -0.23287517, -0.32069653,
         1.39396077, -0.64437982, -0.31437716, -0.55257196,  1.54508494,
         1.34146746, -0.37518699,  0.60075134,  0.46880129,  0.47682362,
        -0.41347373, -0.10659876]), array([ 0.29412643,  0.43817707,  1.64282592,  1.40057225,  1.2905566 ,
        -0.2417939 ,  1.13159748,  1.55184502,  0.93868818, -0.34589899,
        -0.24005716,  1.68312071,  0.86419886,  0.82594424,  0.13665682,
         0.49203036,  0.27453538]), array([-0.3715357 , -0.17974495, -0.25326497, -0.22637134, -0.18087682,
        -0.31446993, -0.06374809, -0.23610481, -0.04816849, -0.33259097,
        -0.30111677, -0.24714186, -0.33409903, -0.29179447, -0.1525501 ,
         0.00709218, -0.02845433])]
============================================================
```

KMEANS

Optimized sequential code:

Using numba with decorator @jit to optimize sequential KMeans code

```
Processing time: 0.11242198944091797 s
Processing time of Calculating distance and Assigning label: 0.06739926338195801 s
Processing time of Updating centers: 0.044393777784729004 s
Number of loops: 34
Centers found by k-means algorithm:
[[ 1.19168987  0.34421507 -0.32010396 -0.23287517 -0.32069653  1.39396077
  -0.64437982 -0.31437716 -0.55257196  1.54508494  1.34146746 -0.37518699
   0.60075134  0.46880129  0.47682362 -0.41347373 -0.10659876]
 [ 0.29412643  0.43817707  1.64282592  1.40057225  1.2905566  -0.2417939
   1.13159748  1.55184502  0.93868818 -0.34589899 -0.24005716  1.68312071
   0.86419886  0.82594424  0.13665682  0.49203036  0.27453538]
 [-0.3715357  -0.17974495 -0.25326497 -0.22637134 -0.18087682 -0.31446993
  -0.06374809 -0.23610481 -0.04816849 -0.33259097 -0.30111677 -0.24714186
  -0.33409903 -0.29179447 -0.1525501   0.00709218 -0.02845433]]
==========================================================
```

KMEANS

Paralyzed sequential code running on CPU:

Using numba with decorator @jit and prange to paralyze sequential KMeans code

```
Processing time: 0.225517272949218 75 s
Processing time of Calculating distance and Assigning label: 0.13729453086853027 s
Processing time of Updating centers: 0.087625503540039 06 s
Number of loops: 41
Centers found by k-means algorithm:
[[ 1.19168987  0.34421507 -0.32010396 -0.23287517 -0.32069653  1.39396077
  -0.64437982 -0.31437716 -0.55257196  1.54508494  1.34146746 -0.37518699
   0.60075134  0.46880129  0.47682362 -0.41347373 -0.10659876]
 [ 0.29412643  0.43817707  1.64282592  1.40057225  1.2905566  -0.2417939
   1.13159748  1.55184502  0.93868818 -0.34589899 -0.24005716  1.68312071
   0.86419886  0.82594424  0.13665682  0.49203036  0.27453538]
 [-0.3715357  -0.17974495 -0.25326497 -0.22637134 -0.18087682 -0.31446993
  -0.06374809 -0.23610481 -0.04816849 -0.33259097 -0.30111677 -0.24714186
  -0.33409903 -0.29179447 -0.1525501   0.00709218 -0.02845433]]
==========================================================
```

KMEANS

Paralyzed sequential code running on GPU:

Thread and Block Configuration:

- Threads per block: because GPUs are optimized for this configuration.
- Blocks per grid: ensures that all data points are covered.

```
threads_per_block = 256
blocks_per_grid = (data_points.shape[0] + threads_per_block - 1) // threads_per_block
```

KMEANS

Distance measurement and assign label:

Computes the Euclidean distance between each data point and each cluster center:

- Threads: each thread is responsible for calculating the distance between a single data point and all cluster centers.
- Blocks: handles a subset of the data points.

$$\|x_p - \mu_i\| = \sqrt{\sum_{j=1}^{n} (x_{pj} - \mu_{ij})^2}$$

KMEANS

Update centroids:

Calculate the new centroids as the mean of all data points assigned to each cluster:

- Threads: each thread contributes to updating the sum of the coordinates of data points for a particular cluster.
- Blocks: handles a subset of the data points.

$$\mu_i = \frac{1}{|C_i|} \sum_{x_p \in C_i} x_p$$

KMEANS

Paralyzed sequential code running on GPU:

Using CUDA and manage threads and blocks to paralyze sequential KMeans code

```
Processing time: 0.2731308937072754 s
Processing time of Calculating distance and Assigning label: 0.11008739471435547 s
Processing time of Updating centers: 0.154240131378173 83 s
Number of loops: 34
Centers found by k-means algorithm:
[[ 1.19168987  0.34421507 -0.32010396 -0.23287517 -0.32069653  1.39396077
  -0.64437982 -0.31437716 -0.55257196  1.54508494  1.34146746 -0.37518699
   0.60075134  0.46880129  0.47682362 -0.41347373 -0.10659876]
 [ 0.29412643  0.43817707  1.64282592  1.40057225  1.2905566  -0.2417939
   1.13159748  1.55184502  0.93868818 -0.34589899 -0.24005716  1.68312071
   0.86419886  0.82594424  0.13665682  0.49203036  0.27453538]
 [-0.3715357  -0.17974495 -0.25326497 -0.22637134 -0.18087682 -0.31446993
  -0.06374809 -0.23610481 -0.04816849 -0.33259097 -0.30111677 -0.24714186
  -0.33409903 -0.29179447 -0.1525501   0.00709218 -0.02845433]]
==========================================================
```

KMEANS

Utilize shared memory while running on GPU/device

```
Processing time: 0.10505962371826172 s
Processing time of Calculating distance and Assigning label: 0.03352832794189453 s
Processing time of Updating centers: 0.06657910346984863 s
Number of loops: 34
Centers found by k-means algorithm:
[[ 1.19168987  0.34421507 -0.32010396 -0.23287517 -0.32069653  1.39396077
  -0.64437982 -0.31437716 -0.55257196  1.54508494  1.34146746 -0.37518699
   0.60075134  0.46880129  0.47682362 -0.41347373 -0.10659876]
 [ 0.29412643  0.43817707  1.64282592  1.40057225  1.2905566  -0.2417939
   1.13159748  1.55184502  0.93868818 -0.34589899 -0.24005716  1.68312071
   0.86419886  0.82594424  0.13665682  0.49203036  0.27453538]
 [-0.3715357  -0.17974495 -0.25326497 -0.22637134 -0.18087682 -0.31446993
  -0.06374809 -0.23610481 -0.04816849 -0.33259097 -0.30111677 -0.24714186
  -0.33409903 -0.29179447 -0.1525501   0.00709218 -0.02845433]]
============================================================
```

KMEANS

Evalute:

- Using KMeans library from Scikit-learn to get final center of each cluster (fixed initial centers).
- Compare the result of this version with the result of KMeans library. If there is no difference then the result is correct.

```
Kmean_sklearn = KMeans(n_clusters=K, init=init_centers)
Kmean_sklearn.fit(df_Standardize)
correct centers = Kmean sklearn.cluster centers
```

KMEANS

Evalute

Final centers given by KMeans library:

```
array([[ 0.20020117, -0.12558956, -0.38743108, -0.2607737 , -0.42295466,
         0.3069329 , -0.86359168, -0.39765792, -0.76194567,  0.4295371 ,
         0.31367256, -0.49396806, -0.05976826, -0.06420056,  0.03561236,
        -0.39206314, -0.08852164],
       [ 0.84891124,  0.45113311,  2.44251754,  2.13367299,  1.82799982,
         0.15685439,  1.13440672,  1.66808695,  1.01929016, -0.05602164,
         0.07703106,  2.3340537 ,  1.25822596,  1.56020803,  0.47670528,
         0.33644917,  0.28187606],
       [-0.38383022,  0.06465949,  0.0113216 , -0.08005086,  0.16248614,
        -0.38351536,  0.79693382,  0.16182432,  0.69983821, -0.48737714,
        -0.37702777,  0.15411254, -0.1560702 , -0.20500673, -0.12659237,
         0.39377209,  0.05203958]])
```

COMPARISON

```
np.mean(np.abs(correct_centers - centers))
```

KMEANS

Conclusion:

	Whole Kmeans algorithm	Calculating distance and Assigning label	Updating centers	Loop	Accuracy
Sequential 1: Pure Python	79.654663	66.513390	13.140366	34.0	0.001758
Sequential 2: optimized KMeans code	0.112422	0.067399	0.044394	34.0	0.001758
Parallel code version 1: paralyzed KMeans code on CPU/host	0.225517	0.137295	0.087626	41.0	0.001758
Parallel code version 2: Paralyzed KMeans code on GPU/device	0.273131	0.110087	0.154240	34.0	0.001758
Parallel code version 3: Utilize shared memory while running on GPU/device	0.105060	0.033528	0.066579	34.0	0.001758

PCA

Finish version of sequential PCA code

- Steps:

1. Calculate the covariance matrix.
2. Find eigenvalues and eigenvectors value.
3. Chose major features.
4. Reduce data dimension.

PCA

1. Calculate the covariance matrix.

$$cov(X,Y) = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{(n-1)}$$

PCA

2. Find eigenvalues and eigenvectors value.

```
eigenvalues, eigenvectors = np.linalg.eig(np.array(cov_matrix))
```

eigenvalues is a 17-dimensional vector
eigenvectors is a matrix 17x17

PCA

3. Choose major features

- Sorting eigenvalues by descending order.
- Rearrange the eigenvectors base on the order of the columns in the eigenvector matrix, corresponding to the descending order of the eigenvalues.
- Select the top n eigenvectors in eigenvectors corresponding to the number of dimensions you want to reduce.

PCA

4. Reducing data dimension

- After selecting the top n eigenvectors in eigenvectors corresponding to the number of dimensions you want to reduce, proceed to multiply the two matrices: the original matrix x the matrix of the top n eigenvectors.

PCA

Sequential code:

Using pure Python to implement PCA algorithm

```
Processing time: 5.60746693611145 s
Timing calculating covariance matrix: 5.238395929336548
Timing getting eigenvalues and eigenvectors: 0.00200581550598144 53
Timing sorting eigenvalues and eigenvectors: 4.0292739868164 06e-05
Timing Selecting top n_components eigenvectors: 9.5367431640625e-07
Timing Reducing dimension of data: 0.36568474769592285
```

PCA

Optimized sequential code:

Using numba with decorator @jit to optimize sequential PCA code

```
Processing time: 0.01402091979804688 s
Timing calculating covariance matrix: 0.0116529464721679 69
Timing getting eigenvalues and eigenvectors: 0.0001988410949707 0312
Timing sorting eigenvalues and eigenvectors: 3.6478042602539 06e-05
Timing Selecting top n_components eigenvectors: 1.430511474609375e-06
Timing Reducing dimension of data: 0.0018751621246337 89
```

PCA

Paralyzed PCA code running on CPU/host:

Using numba with decorator @jit to optimize sequential PCA code

```
Processing time: 0.0090517997741699 22 s
Timing calculating covariance matrix: 0.0077404975891113 28
Timing getting eigenvalues and eigenvectors: 0.0001688003540039 0625
Timing sorting eigenvalues and eigenvectors: 2.956390380859375e-05
Timing Selecting top n_components eigenvectors: 1.430511474609375e-06
Timing Reducing dimension of data: 0.0009584426879882812
```

Paralyzed sequential code running on GPU : Computing the Covariance Matrix

01

Parallelized Step:
The computation of the covariance matrix is parallelized. Each element of the covariance matrix is computed independently, making it an excellent candidate for parallelization.

02

Block and Grid Size:

- We choose a 2D block size of (16, 16) threads as ourr covariance matrix size is 17x17.
- The grid size is (2x2) calculated to cover the entire covariance matrix, which is n_features x n_features.
- Each thread computes one element of the covariance matrix.
- The if i < n_features and j < n_features check ensures we don't compute unnecessary elements if the matrix size isn't exactly divisible by the block size.

03

What each thread does:

- Determines its position (i, j) in the covariance matrix using cuda.grid(2).
- Computes the covariance between feature i and feature j
- Divides the sum by (n_samples - 1) to get the covariance.
- Stores the result in cov_matrix[i, j].

PCA

Paralyzed sequential code running on GPU : Computing the Covariance Matrix

01

Parallelized Step:
The projection of the data onto the principal components is parallelized. Each element of the projected data matrix is computed independently.

02

Block and Grid Size:

- Again, we use a 2D block size of (16, 16) threads for the same reasons as before.
- The grid size is calculated to cover the entire projected data matrix, which is n_samples x n_components.
- Each thread computes one element of the projected data matrix.
- The if i < n_samples and j < n_components check ensures we don't compute unnecessary elements.

03

What each thread does:

- Determines its position (i, j) in the projected data matrix using cuda.grid(2).
- Computes the dot product between the i-th data point and the j-th eigenvector:
  - Iterates over all features.
  - For each feature, it multiplies the value of the data point with the corresponding eigenvector component.
  - Sums these products.
- Stores the result in X_pca[i, j].

PCA

Paralyzed sequential code running on GPU

```
Processing time: 0.02676129341125488 3 s
Timing calculating covariance matrix: 0.023976087570190 43
Timing getting eigenvalues and eigenvectors: 0.0001187324523925 9812
Timing sorting eigenvalues and eigenvectors: 2.00271606445312 5e-05
Timing Selecting top n_components eigenvectors: 7.1525573730468 75e-07
Timing Reducing dimension of data: 0.0025126934051513 67
```

COMPARISON

Pearson correlation coefficient for principal components

```
Correlations between our implementation and sklearn principal components:
  PC1: 1.0000
  PC2: 1.0000
```

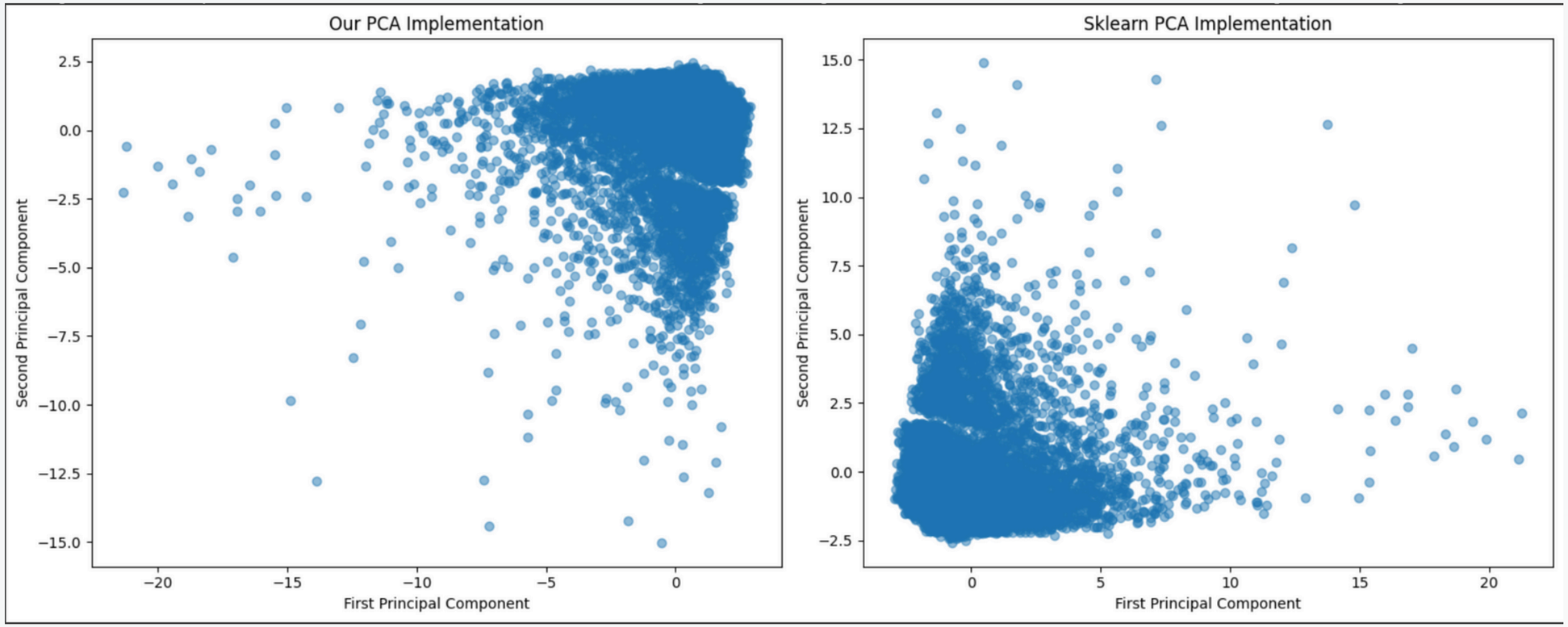
Correlation Coefficient for the dimensions of the two reduced datasets

```
Correlations between our implementation and sklearn reduced data dimensions:
  Dimension 1: -1.0000
  Dimension 2: -1.0000
```

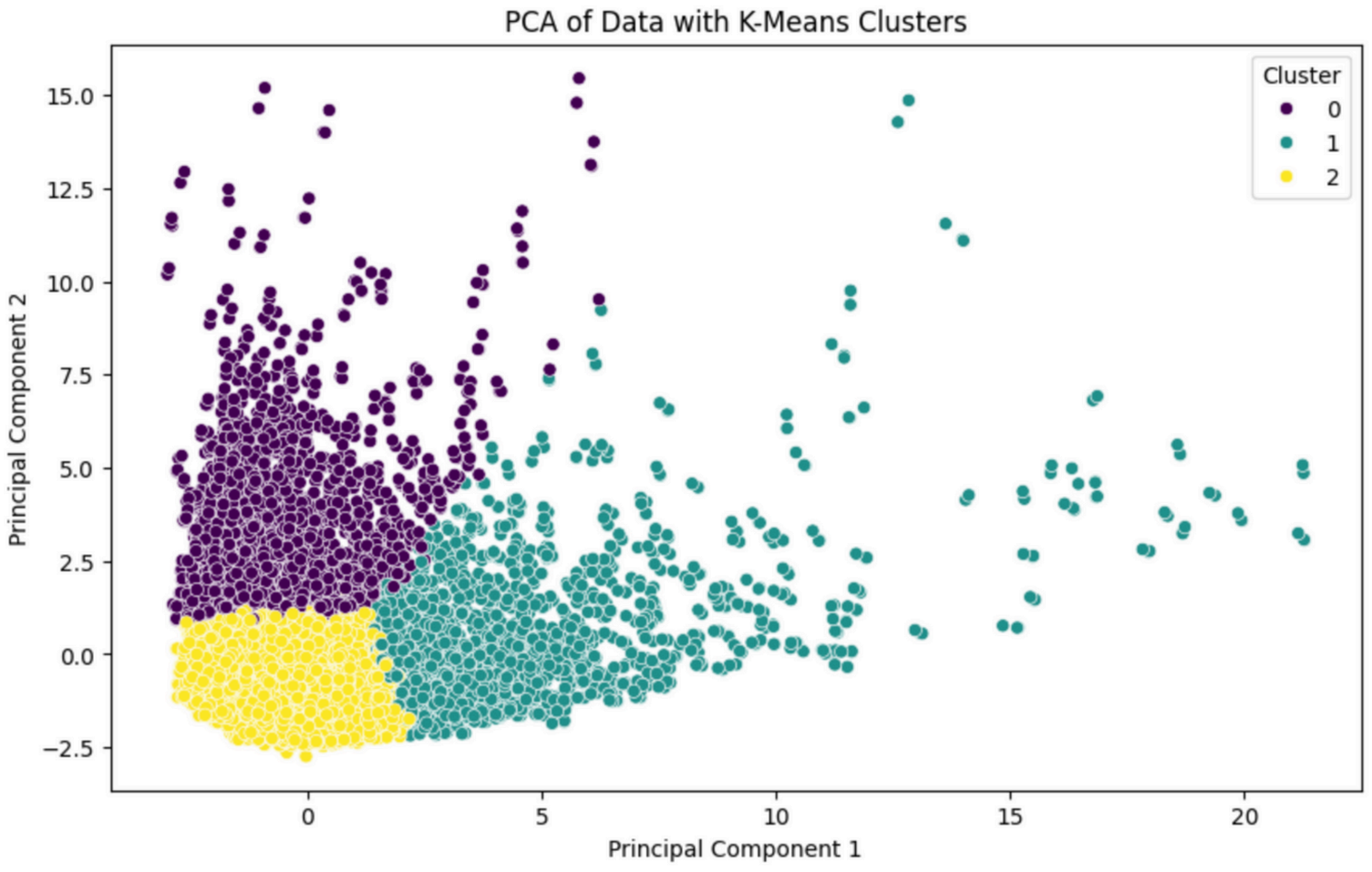
CONCLUSION

	Whole PCA algorithm	Calculate Covariance matrix	Reduce dimension of data
Sequential code version 1: Pure PCA code using Python	5.607467	5.238396	0.365685
Sequential code version 2: Optimized sequential PCA code	0.014021	0.011653	0.001875
Parallel code version 1: Paralyzed PCA code running on CPU/host	0.009052	0.007740	0.000958
Parallel code version 2: Paralyzed PCA code running on GPU/device	0.026761	0.023976	0.002513

COMPARISON


Our PCA Implementation
Second Principal Component
First Principal Component
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
−15.0
−20
−15
−10
−5
0
Sklearn PCA Implementation
Second Principal Component
First Principal Component
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
0
5
10
15
20

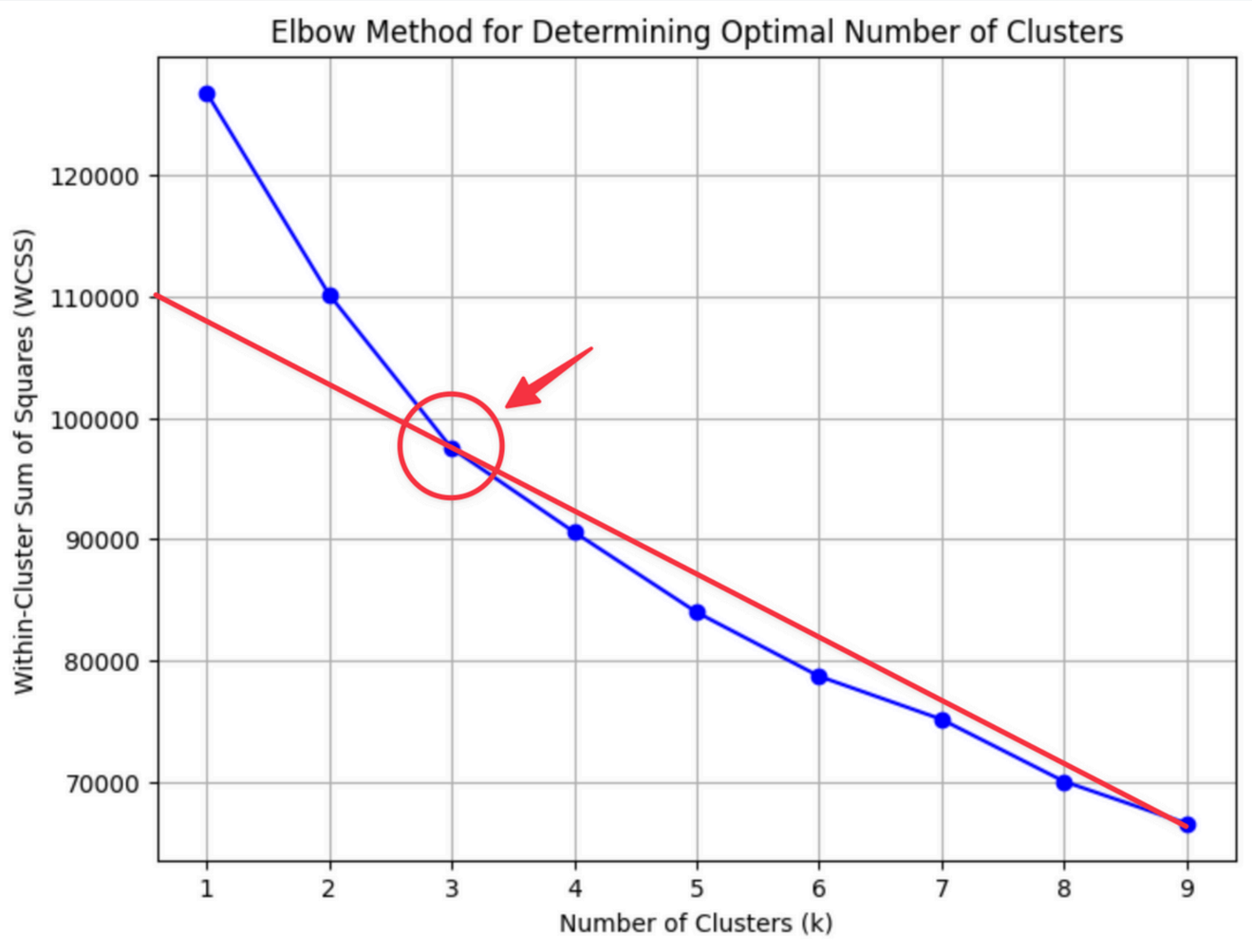
PCA RESULT


PCA of Data with K-Means Clusters
Cluster
0
1
2
Principal Component 2
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
0
5
10
15
20
Principal Component 1

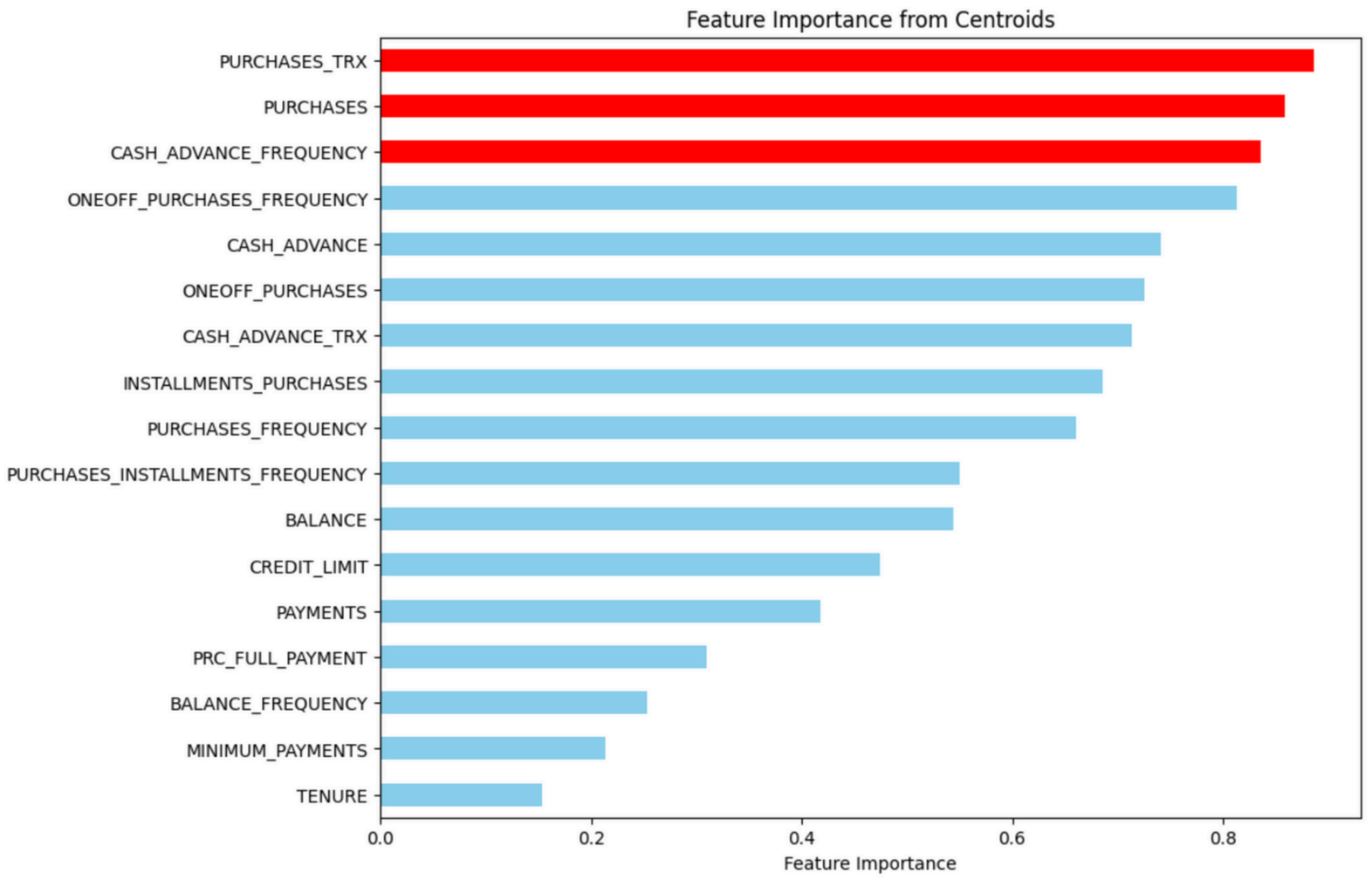
04

MEANING OF CLUSTERS

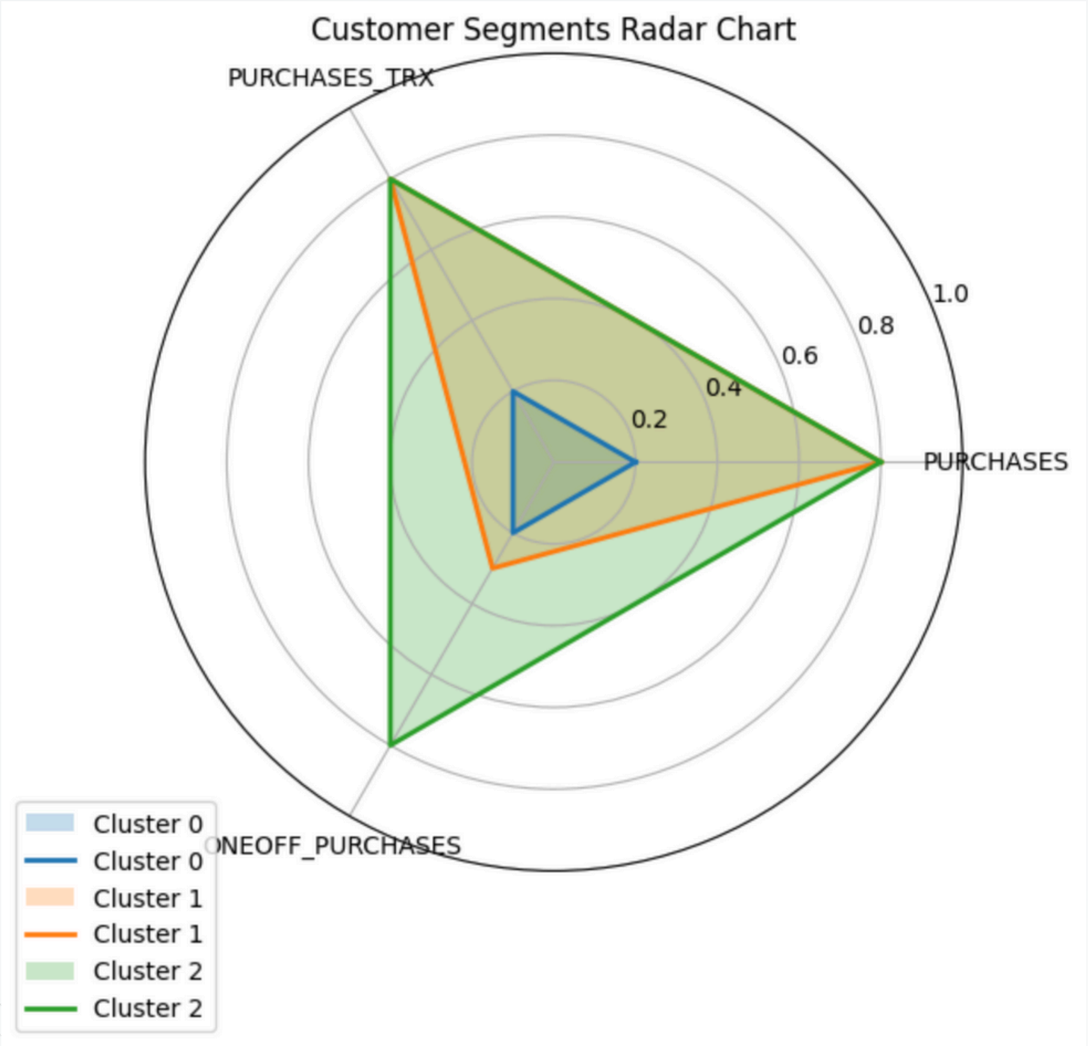
WHY 3 CLUSTER ARE BETTER



MEANING OF EACH CLUSTERS



MEANING OF EACH CLUSTERS



05
INSIGHT

INSIGHTS

Cluster 0

- Customers with low overall purchase amounts, low transaction frequency, and low one-off purchases.
- This cluster represents low-engagement customers. They might be cautious spenders or less active in using their credit cards.

Cluster 1

- Customers with high purchase amounts and transaction frequency, but low one-off purchase amounts.
- These customers are regular spenders who prefer smaller, frequent transactions. They might be using the card for daily expenses rather than large, occasional purchases.

Cluster 2

- Customers with high purchase amounts, high transaction frequency, and high one-off purchases.
- This is the premium segment, with customers who make frequent transactions and are also willing to spend large amounts in a single go. They likely have higher credit limits and are more comfortable with larger financial commitments.

05

ACHIEVEMENT

ACHIEVEMENT

Kmeans algorithm

The result of clustering on test dataset is remarkable.
Processing time is acceptable with small dataset

PCA algorithm

Success in reducing data dimension while avoid losing data information
Processing time is significant

07

SUMMARY

Completed Work and Work Plan

SUMMARY

Completed Work

01 Using pure Python to implement KMeans and PCA algorithm

02 Using numba with decorator @jit to optimize sequential KMeans and PCA code

03 Using numba with decorator @jit to paralyze KMeans and PCA code

04 Understand meaning of each cluster

05 Extract insights after doing project

06
- Complete parallelize minor K-mean and PCA code.
- Complete writing report

07 Submit final code

THANK'S FOR LISTENING

20127214-
20127612