

Comparative Clustering of Shape and S-Set Datasets:

S-sets:

Synthetic 2-d data with N=5000 vectors and k=15 Gaussian clusters with different degree of cluster overlap

P. Fränti and O. Virmajoki, "Iterative shrinking method for clustering problems", Pattern Recognition, 39 (5), 761-765, May 20

Shape Sets:

Pathbased

N=300, k=3, D=2 Pathbased: txt H. Chang and D.Y. Yeung, Robust path-based spectral clustering. Pattern Recognition, 2008. 41(1): p. 191-2

Spiral

N=312, k=3, D=2 Spiral: txt H. Chang and D.Y. Yeung, Robust path-based spectral clustering. Pattern Recognition, 2008. 41(1): p. 191-2

Jain

N=373, k=2, D=2 Jain: txt A. Jain and M. Law, Data clustering: A user's dilemma. Lecture Notes in Computer Science, 2005. 3776: p. 1-

Flame

N=240, k=2, D=2 Flame: txt L. Fu and E. Medico, FLAME, a novel fuzzy clustering method for the analysis of DNA microarray data. BMC bioinformatics, 2007. 8(1): p. 3.10.03.03.06

Importing modules and datasets

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

S1 = pd.read_csv("s1.txt",sep="\s+",header=None)
S4 = pd.read_csv("s4.txt",sep="\s+",header=None)

#Third row is label
pathbased = pd.read_csv("pathbased.txt",sep="\s+",header=None)
spiral = pd.read_csv("spiral.txt",sep="\s+",header=None)
jain = pd.read_csv("jain.txt",sep="\s+",header=None)
flame = pd.read_csv("flame.txt",sep="\s+",header=None)
spiral
```

	0	1	2
0	31.95	7.95	3
1	31.15	7.30	3
2	30.45	6.65	3
3	29.70	6.00	3
4	28.90	5.55	3
...
307	15.75	13.85	2
308	15.65	14.05	2
309	15.65	14.25	2
310	15.65	14.50	2
311	15.65	14.60	2

```
dfs= {"S1": [S1, 15],
      "S4": [S4, 15],
```

```

    "pathbased":[pathbased,3],
    "spiral":[spiral,3],
    "jain":[jain,2],
    "flame":[flame,2]

}

for df_name in dfs.keys():
    if df_name=="S1" or df_name=="S4":
        pass
    else:
        df=dfs[df_name][0]
        df.rename(columns={2:"true_cluster"},inplace=True)
        df['true_cluster']=df['true_cluster']-1 # Changing the labeling from 1...k to 0...

flame.true_cluster.unique()

array([0, 1], dtype=int64)

```

Visualization

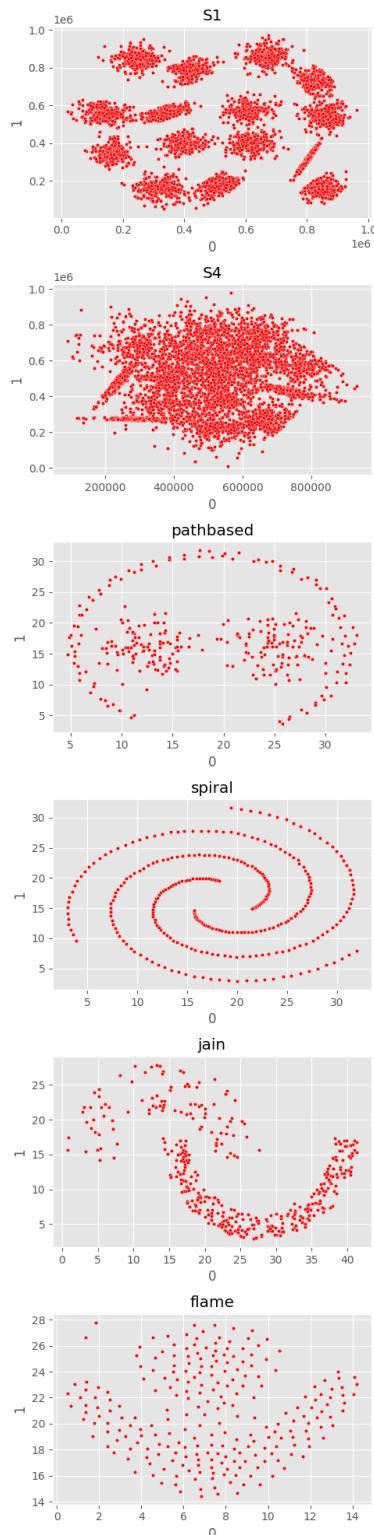
```

plt.style.use('ggplot')

fig,axs=plt.subplots(6,1,figsize=(5,20))
axs=axs.ravel()
i=0
for df in dfs.keys():
    axs[i].set_title(f'{df}')

    sns.scatterplot(x=0,y=1,data=dfs[df][0],ax=axs[i],color="r",s=10)
    i+=1
fig.tight_layout()
#plt.show()

```



```
flame
```

	0	1	true_cluster
0	1.85	27.80	0
1	1.35	26.65	0
2	1.40	23.25	1
3	0.85	23.05	1
4	0.50	22.35	1
...
235	7.50	26.20	0
236	7.50	25.65	0
237	7.05	25.85	0
238	6.90	27.15	0
239	6.15	26.90	0

```
S1.columns
```

```
Index([0, 1], dtype='int64')
```

Kmeans

Kmeans tries to minimize within cluster variability. Within cluster variability is defined as sum of squared distance of each pair of data in a cluster. The Distance is defined as Euclidean Distance. The Optimization of this objective is not trivial. Instead in the K-means algorithm a local optimum is obtained by the following operation.

Initialization:

Samples are randomly assigned to a cluster First Step:

Cluster Centroid is calculated (a vector with k dimensions each feature with the mean of samples in the cluster)

Second Step: The observations that have the minimum distance to a centroid are assigned to that cluster

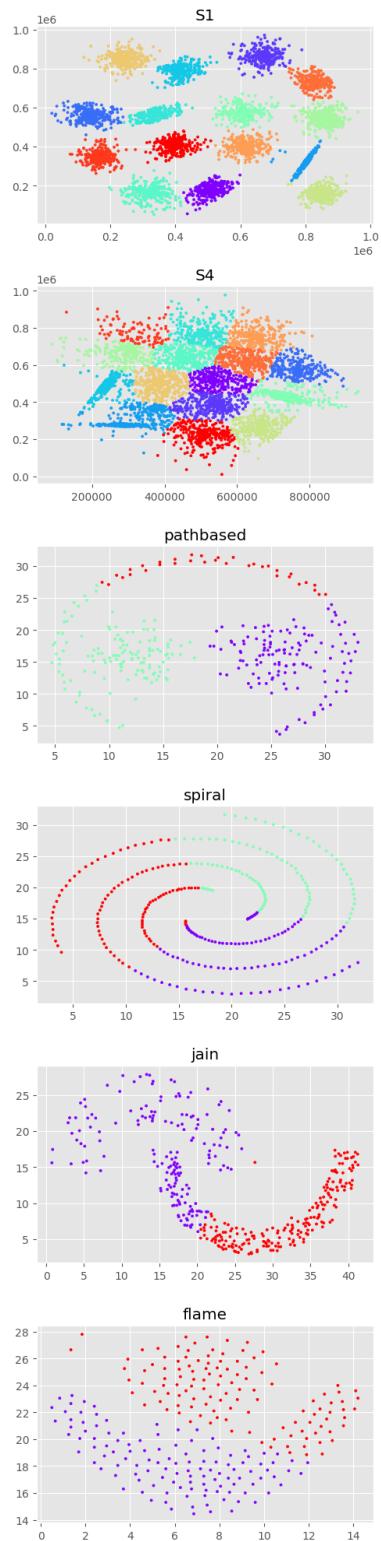
First and Second Steps Are iterated .

```
from sklearn.cluster import KMeans  
  
fig,axs=plt.subplots(6,1,figsize=(5,20))
```

```
axs=axs.ravel()
i=0
for df_name in dfs.keys():
    dataset=dfs[df_name] [0]
    k = dfs[df_name] [1]

    kmeans=KMeans(n_clusters=k,n_init='auto')
    kmeans.fit(dataset.loc[:,0:1])
    dataset["kmeans_cluster"]=kmeans.labels_
    axs[i].set_title(f'{df_name}')

#sns.scatterplot(x=0,y=1,data=dataset,ax=axs[i],hue='kmeans_cluster',s=10)
#axs[i].scatter(x=dataset.loc[:,0],y=dataset.loc[:,1],c=dataset.loc[:, 'kmeans_cluster'])
i+=1
fig.tight_layout()
#plt.show()
```



Hierarchical Clustering

Two Dissimilarity measures need to be defined here: Metric : is a measure of distance of two observations. could be euclidean,l1,l2, correlation based and etc. Linkage: is about how we find the dissimilarity between two groups of observations. Ward: minimizes the variance of clusters Single: calculates pairwise distances of observations of two clusters and considers the smallest one Complete: calculates pairwise distances of observations of two clusters and considers the largest one Average: calculates pairwise distances of observations of two clusters and averages them Centroid: Dissimilarity between center of one cluster and the center of the other cluster. Center is a mean feature vector of dim p

Documentation of Scikit learn

Metric : str or callable, default=None Metric used to compute the linkage. Can be “euclidean”, “l1”, “l2”, “manhattan”, “cosine”, or “precomputed”. If set to `None` then “euclidean” is used. If linkage is “ward”, only “euclidean” is accepted. If “precomputed”, a distance matrix is needed as input for the fit method.

linkage : {‘ward’, ‘complete’, ‘average’, ‘single’}, default=‘ward’ Which linkage criterion to use. The linkage criterion determines which distance to use between sets of observation. The algorithm will merge the pairs of cluster that minimize this criterion.

- ward minimizes the variance of the clusters being merged.
- average uses the average of the distances of each observation of the two sets.
- complete or maximum linkage uses the maximum distances between all observations of the two sets.
- single uses the minimum of the distances between all observations of the two sets

Average Linkage

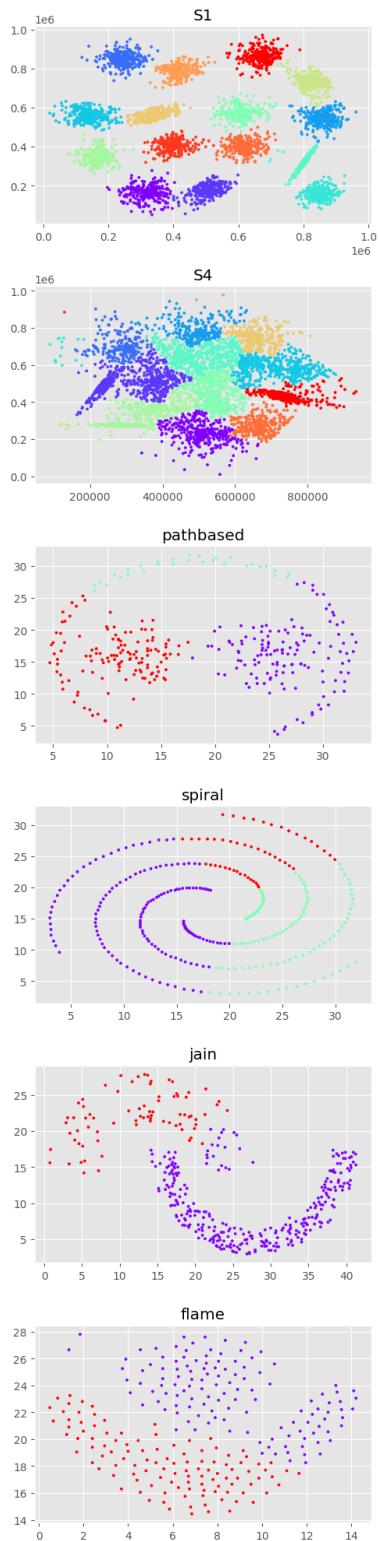
```
from sklearn.cluster import AgglomerativeClustering

fig,axs=plt.subplots(6,1,figsize=(5,20))
axs=axs.ravel()
i=0
for df_name in dfs.keys():
    dataset=dfs[df_name][0]
    k = dfs[df_name][1]

    hierarchical_cluster = AgglomerativeClustering(n_clusters=k, metric='euclidean', linka
    labels = hierarchical_cluster.fit_predict(dataset.loc[:,0:1])
```

```
dataset["avg_hcluster"] = labels
axs[i].set_title(f'{df_name}')
```

```
#sns.scatterplot(x=0,y=1,data=dataset,ax=axs[i],hue='kmeans_cluster',s=10)
#axs[i].scatter(x=dataset.loc[:,0],y=dataset.loc[:,1],c=labels,cmap='rainbow',s=5)
#i+=1
#fig.tight_layout()
#plt.show()
```

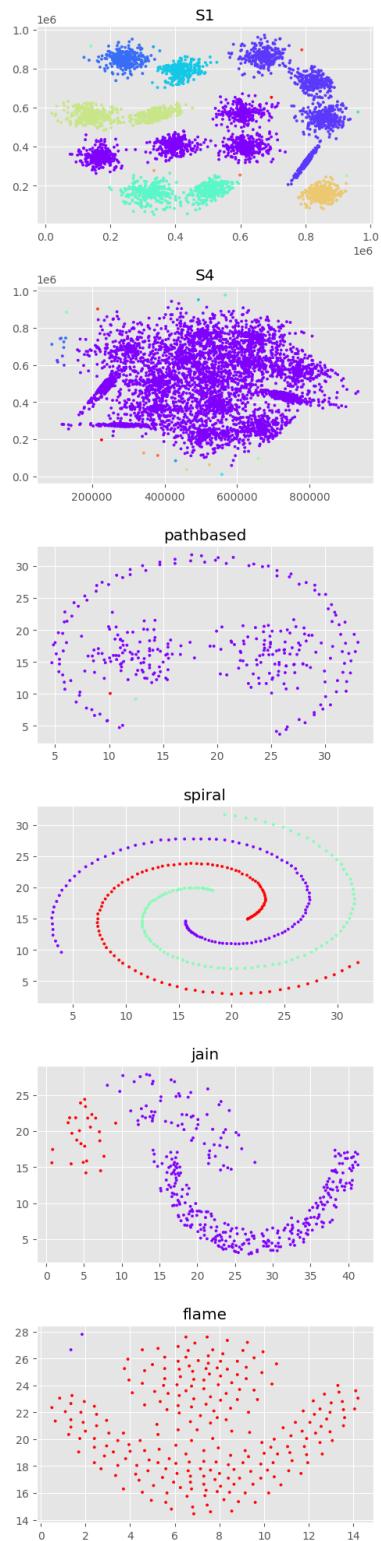


Single Linkage

```
from sklearn.cluster import AgglomerativeClustering

fig,axs=plt.subplots(6,1,figsize=(5,20))
axs=axs.ravel()
i=0
for df_name in dfs.keys():
    dataset=dfs[df_name][0]
    k = dfs[df_name][1]

    hierarchical_cluster = AgglomerativeClustering(n_clusters=k, metric='euclidean', linkage='single')
    labels = hierarchical_cluster.fit_predict(dataset.loc[:,0:k])
    dataset["single_hcluster"] = labels
    axs[i].set_title(f'{df_name}')
```

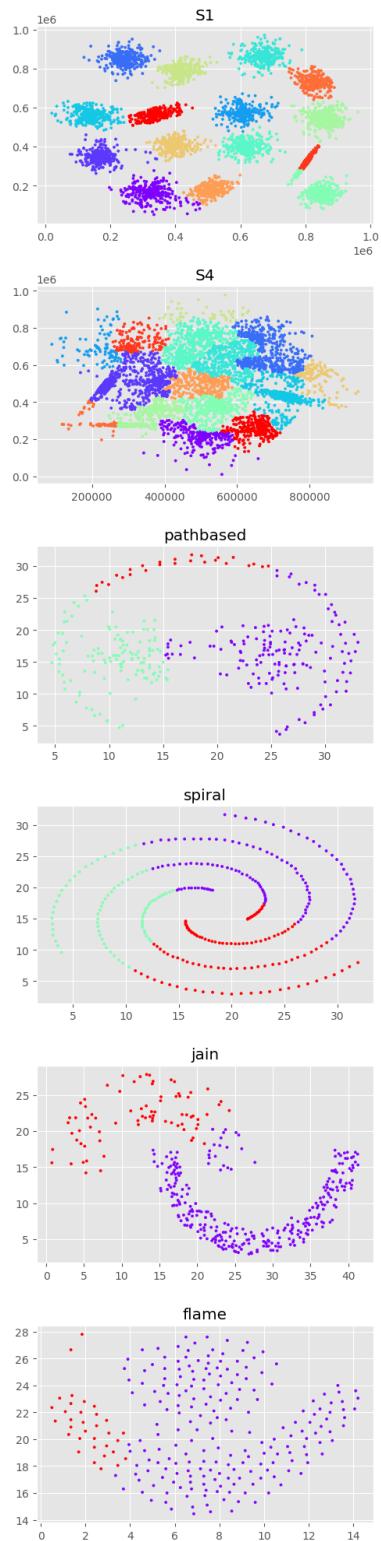


Complete Linkage

```
from sklearn.cluster import AgglomerativeClustering

fig,axs=plt.subplots(6,1,figsize=(5,20))
axs=axs.ravel()
i=0
for df_name in dfs.keys():
    dataset=dfs[df_name][0]
    k = dfs[df_name][1]

    hierarchical_cluster = AgglomerativeClustering(n_clusters=k, metric='euclidean', linkage='complete')
    labels = hierarchical_cluster.fit_predict(dataset.loc[:,0:1])
    dataset["complete_hcluster"] = labels
    axs[i].set_title(f'{df_name}')
```



Spectral Clustering

For this algorithm a similarity graph must be constructed. In this graph the edges represent the similarity between nodes (objects). Similar nodes are connected and the edge weight represents the similarity and if the similarity between two nodes is not above a certain threshold, they don't get connected. In this case the problem boils down to a graph partition problem, where we have to get the connected components of the graph such that edges in each component have high weight and edges between components have low weight. Two metrics should be defined here:

- 1-The measure of similarity between nodes: It's usually defined as $s(x_i, x_j) = \exp(-\|x_i - x_j\|^2/c)$

2- How we construct the graph similarity :- Mutual K-nearest-neighbor graph: If node v_i and v_j are both k-nearest neighbors of each other, they are connected and $w_{ij} = s_{ij}$ - Fully connected graph: connect all points and weight each edge with s_{ij} - epsilon neighborhood graph

affinity : str or callable, default='rbf' How to construct the affinity matrix.
- 'nearest_neighbors': construct the affinity matrix by computing a graph of nearest neighbors.
- 'rbf': construct the affinity matrix using a radial basis function (RBF) kernel.

```
from sklearn.cluster import SpectralClustering
spectral_cluster = SpectralClustering(n_clusters=k, affinity="nearest_neighbors")

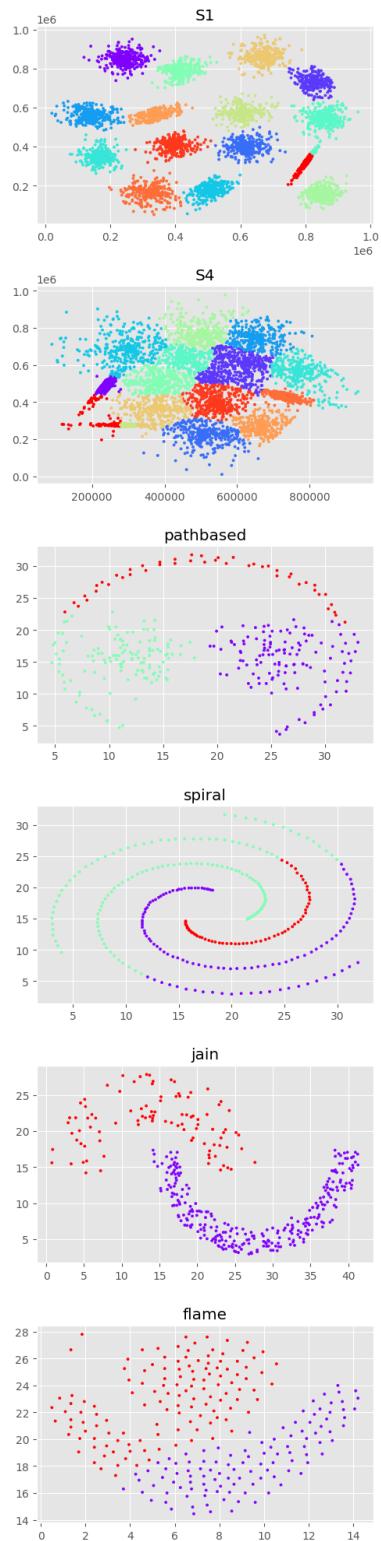
from sklearn.cluster import SpectralClustering

fig, axs=plt.subplots(6,1,figsize=(5,20))
axs=axs.ravel()
i=0
for df_name in dfs.keys():
    dataset=dfs[df_name][0]
    k = dfs[df_name][1]

    spectral_cluster = SpectralClustering(n_clusters=k, affinity="nearest_neighbors")
    spectral_cluster.fit(dataset.loc[:,0:1])
    dataset["spectral_cluster"]= spectral_cluster.labels_
    axs[i].set_title(f'{df_name}')

#sns.scatterplot(x=0,y=1,data=dataset,ax=axs[i],hue='kmeans_cluster',s=10)
axs[i].scatter(x=dataset.loc[:,0],y=dataset.loc[:,1],c=dataset["spectral_cluster"],cmap
i+=1
fig.tight_layout()
plt.show()
```

```
D:\Python3.11\Lib\site-packages\sklearn\manifold\_spectral_embedding.py:273: UserWarning: Gra  
warnings.warn(
```



```
for df_name in dfs.keys():

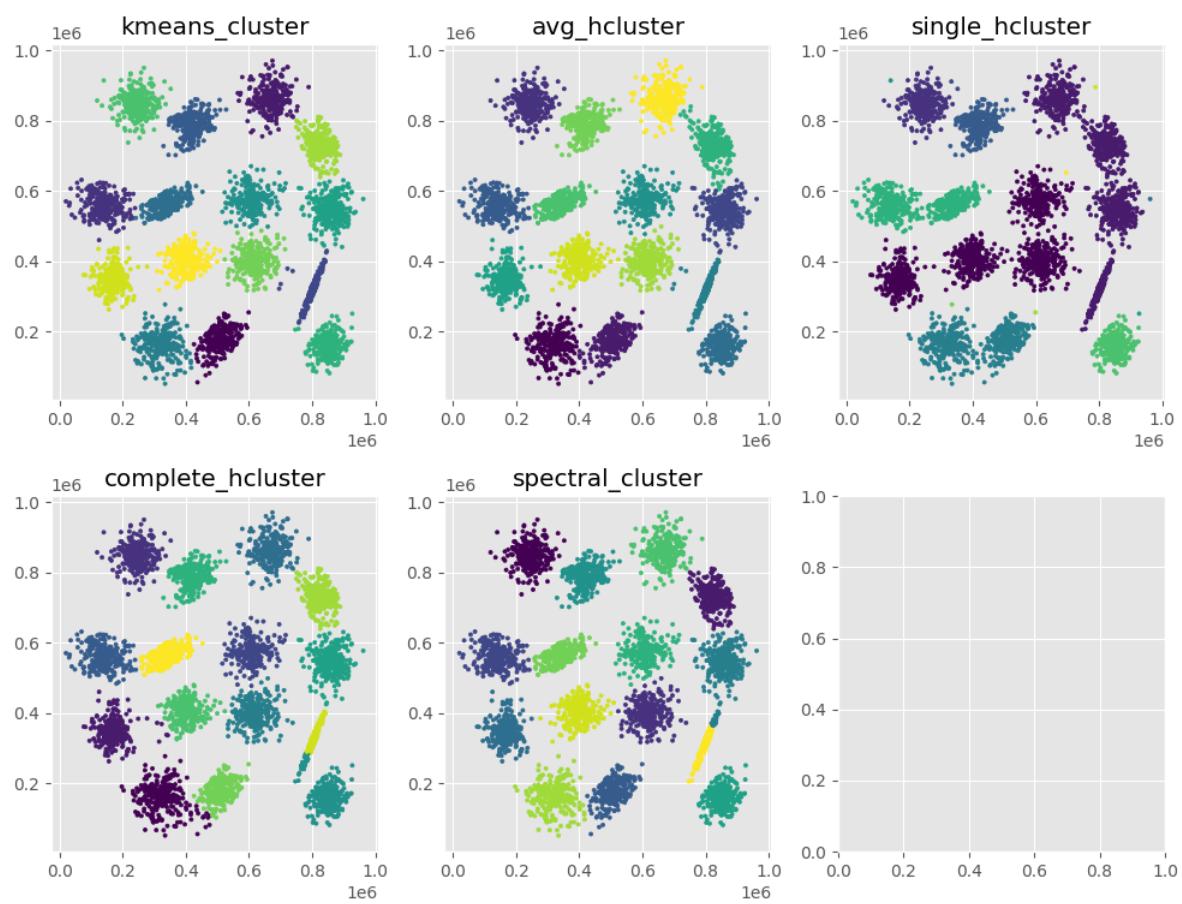
    fig,axs=plt.subplots(2,3,figsize=(10,8))
    axs=axs.ravel()
    dataset=dfs[df_name][0]
    k = dfs[df_name][1]
    fig.suptitle(f'{df_name}')
    for i in range(6):
        try:

            axs[i].set_title(f'{dataset.iloc[:,i+2].name}')

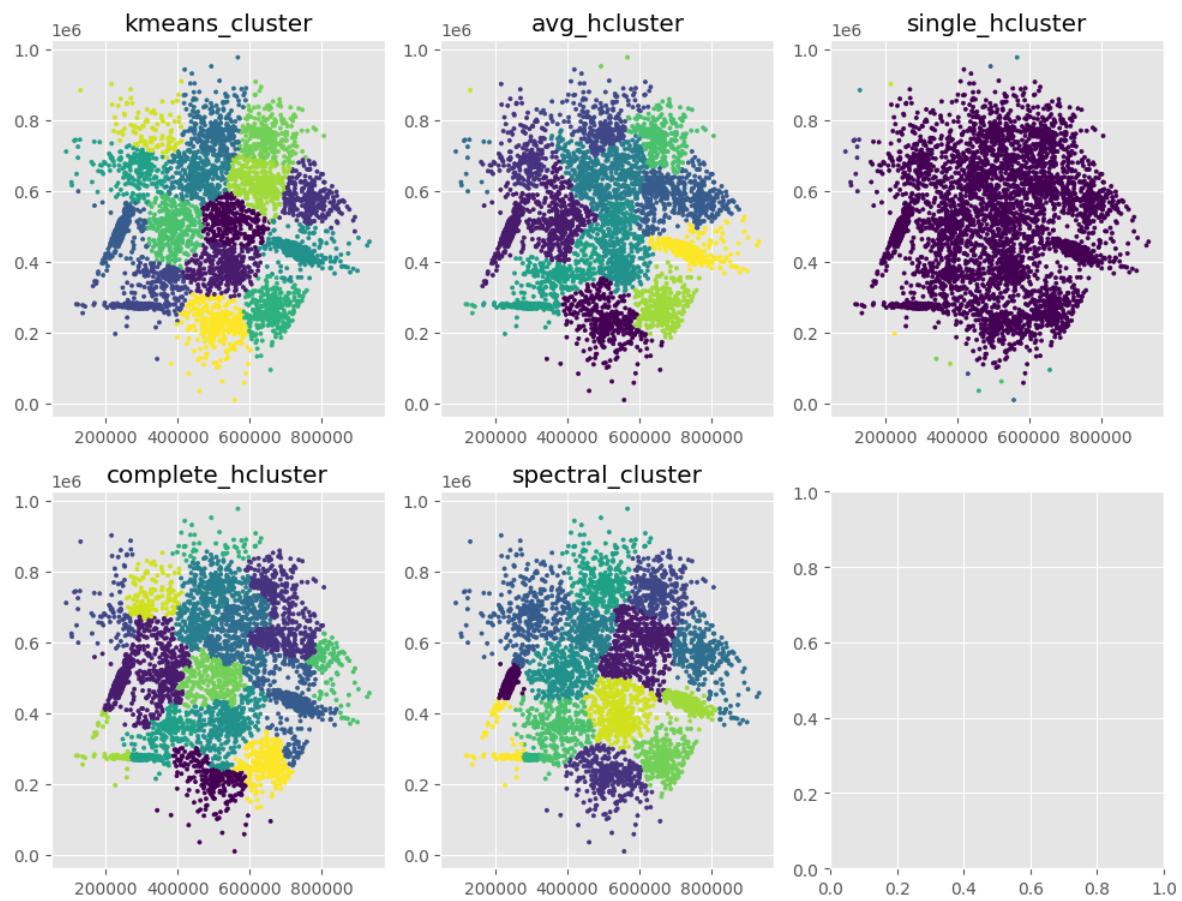
            #sns.scatterplot(x=0,y=1,data=dataset,ax=axs[i],hue='kmeans_cluster',s=10)
            #print(f"x:{x}\ny:{y}")
            axs[i].scatter(x=dataset.loc[:,0],y=dataset.loc[:,1],c=dataset.iloc[:,i+2],cmap='viridis')
        except:
            pass
    fig.tight_layout()

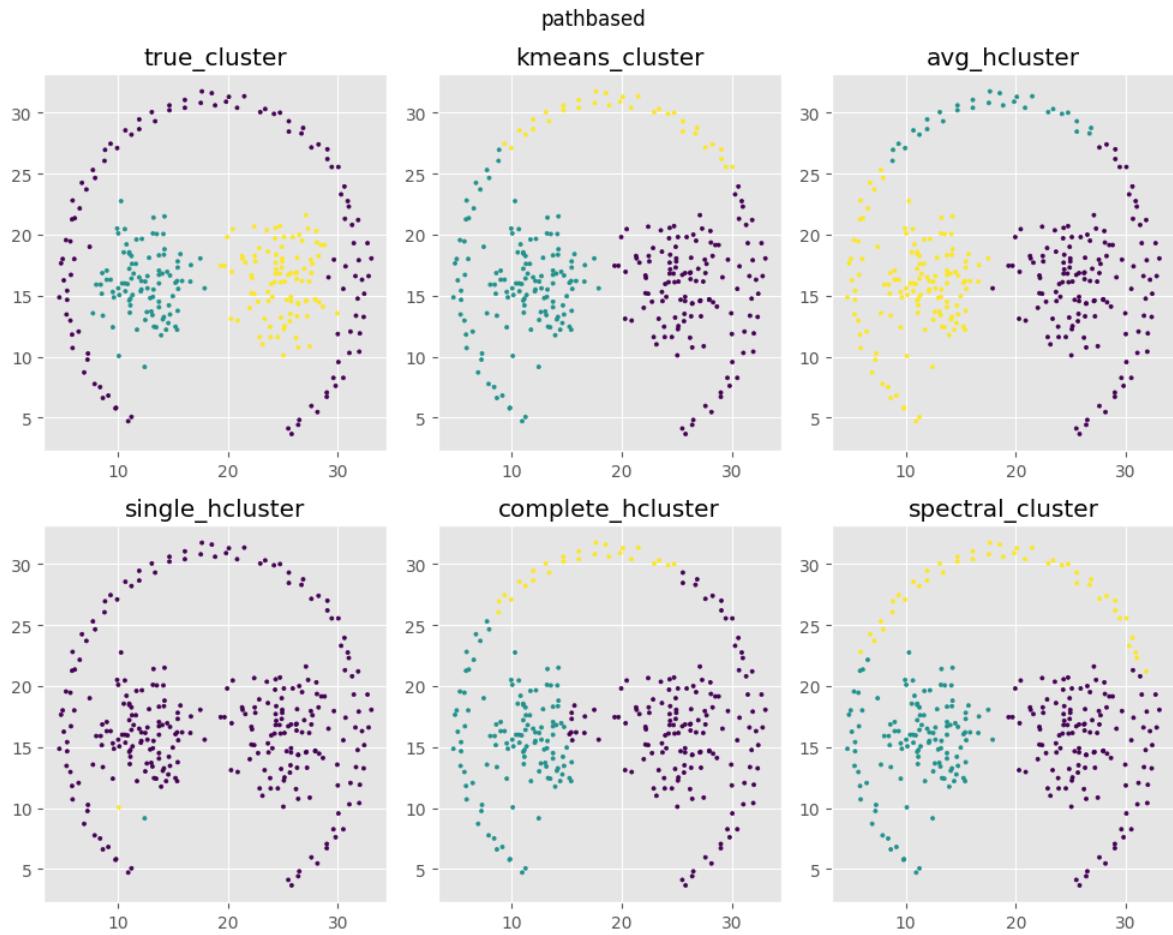
#plt.show()
```

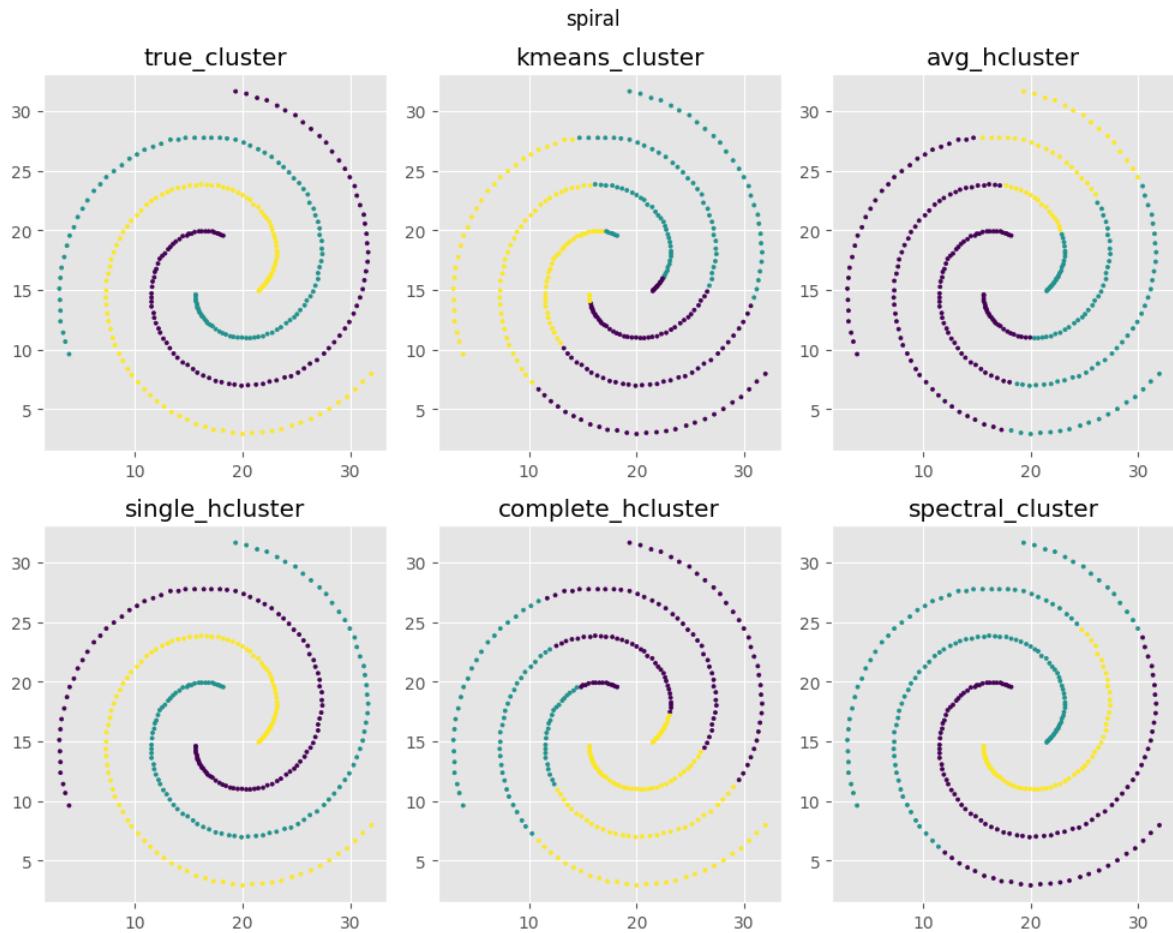
S1

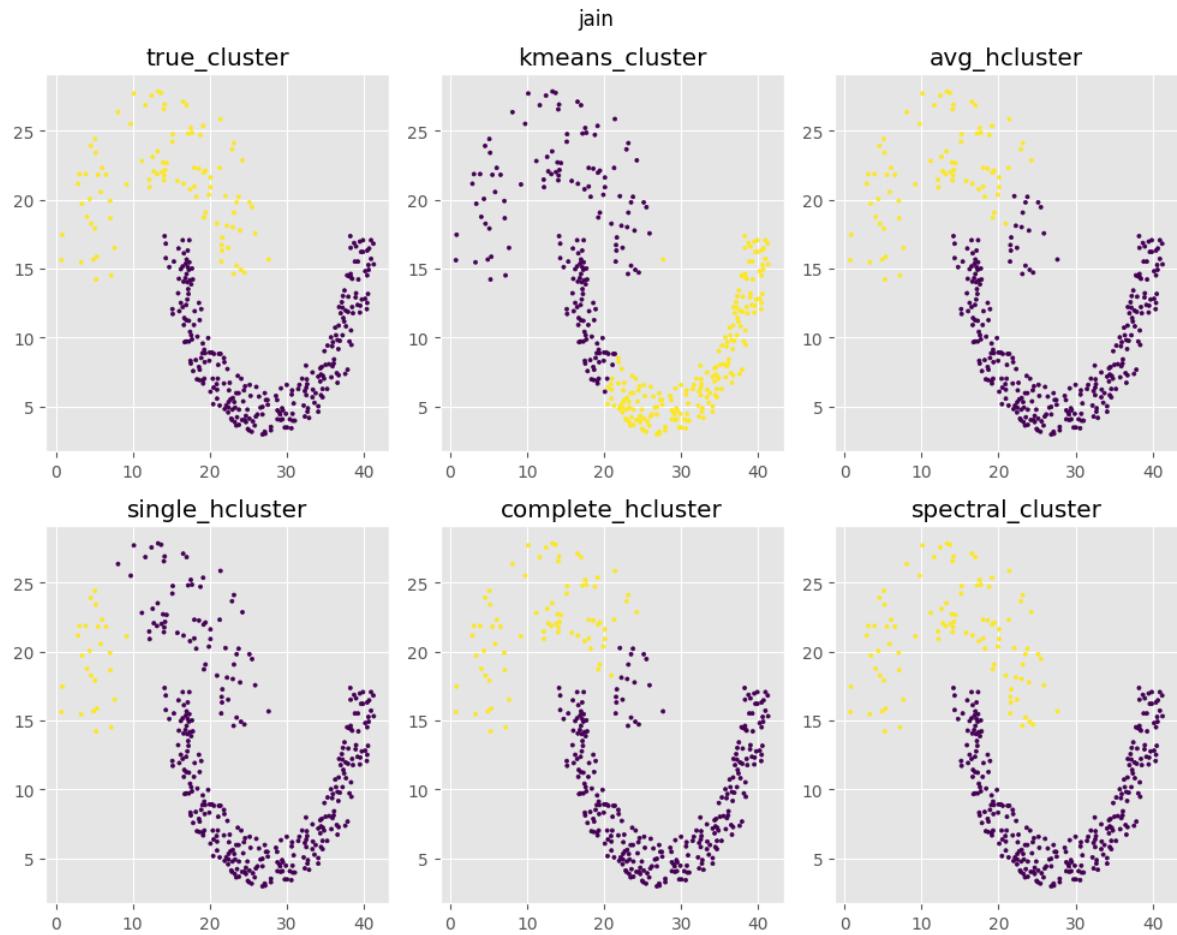


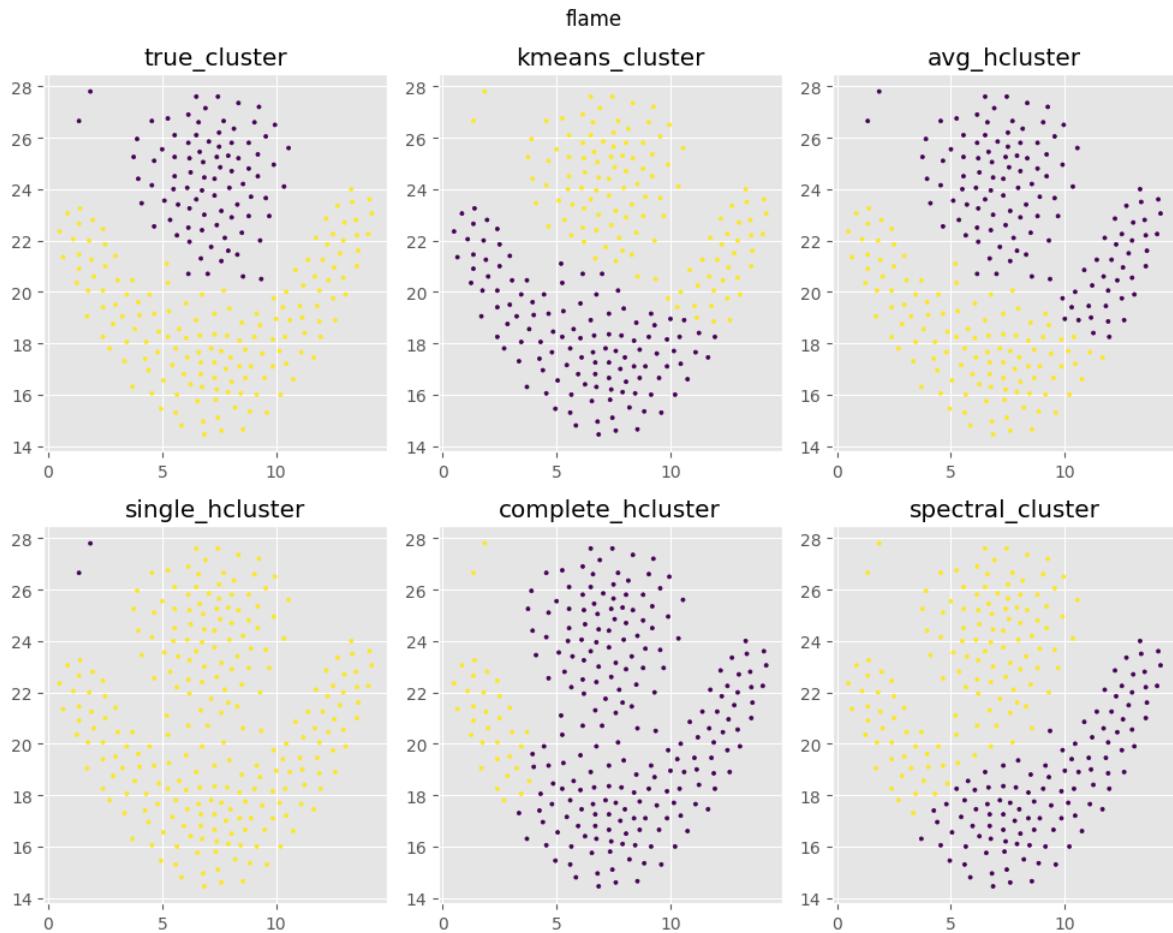
S4











Purity index :

```

from sklearn.metrics import confusion_matrix,classification_report
for df_name in dfs.keys():
    if df_name=="S1" or df_name=="S4":
        continue
    dataset=dfs[df_name][0]
    k = dfs[df_name][1]
    N= len(dataset)
    true= dataset.true_cluster
    #print()
    plt.suptitle(f"{df_name}")
    fig,axs=plt.subplots(2,3,figsize=(10,8))

```

- **Purity**: Purity is a measure of the extent to which clusters contain a single class.^[37]

Its calculation can be thought of as follows: For each cluster, count the number of data points from the most common class in said cluster. Now take the sum over all clusters and divide by the total number of data points. Formally, given some set of clusters M and some set of classes D , both partitioning N data points, purity can be defined as:

$$\frac{1}{N} \sum_{m \in M} \max_{d \in D} |m \cap d|$$

Figure 1: image.png

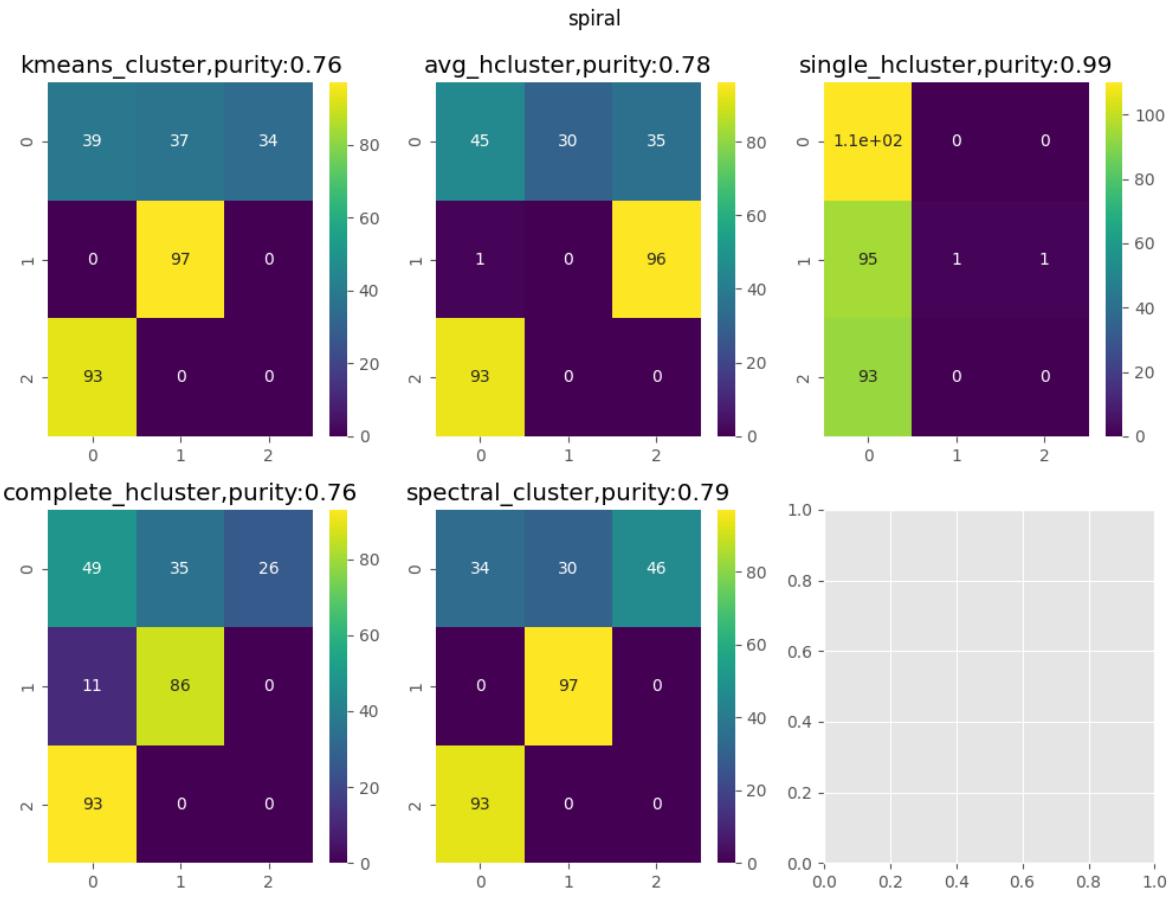
```

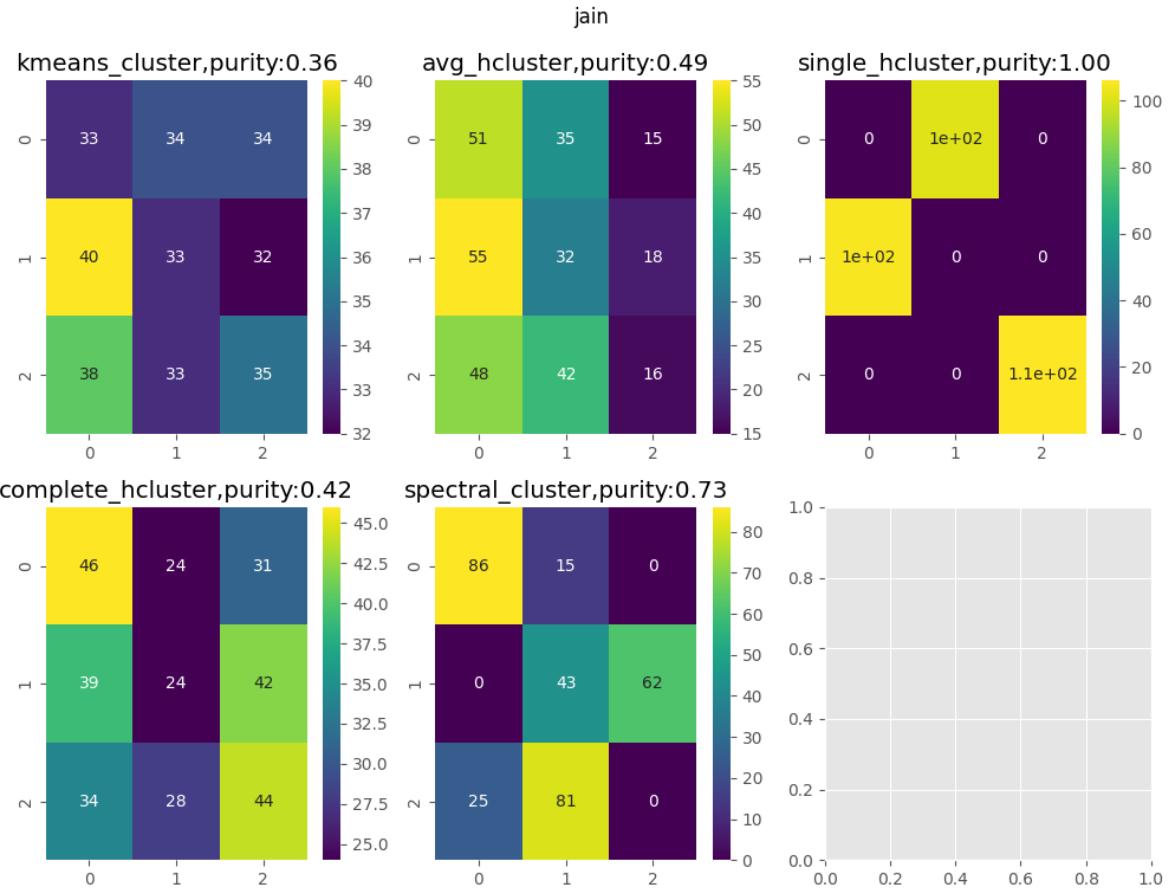
axs=axs.ravel()
for i in range(5):
    predicted=dataset.iloc[:,i+3]
    confusion_mat=confusion_matrix(true, predicted)
    #print(confusion_mat)
    purity= np.sum(np.amax(confusion_mat, axis=1))/len(dataset)
    #print(f"Purity Index of {dataset.iloc[:,i+3].name}:{purity}")
    axs[i].set_title(f"{dataset.iloc[:,i+3].name},purity:{purity:.2f}")

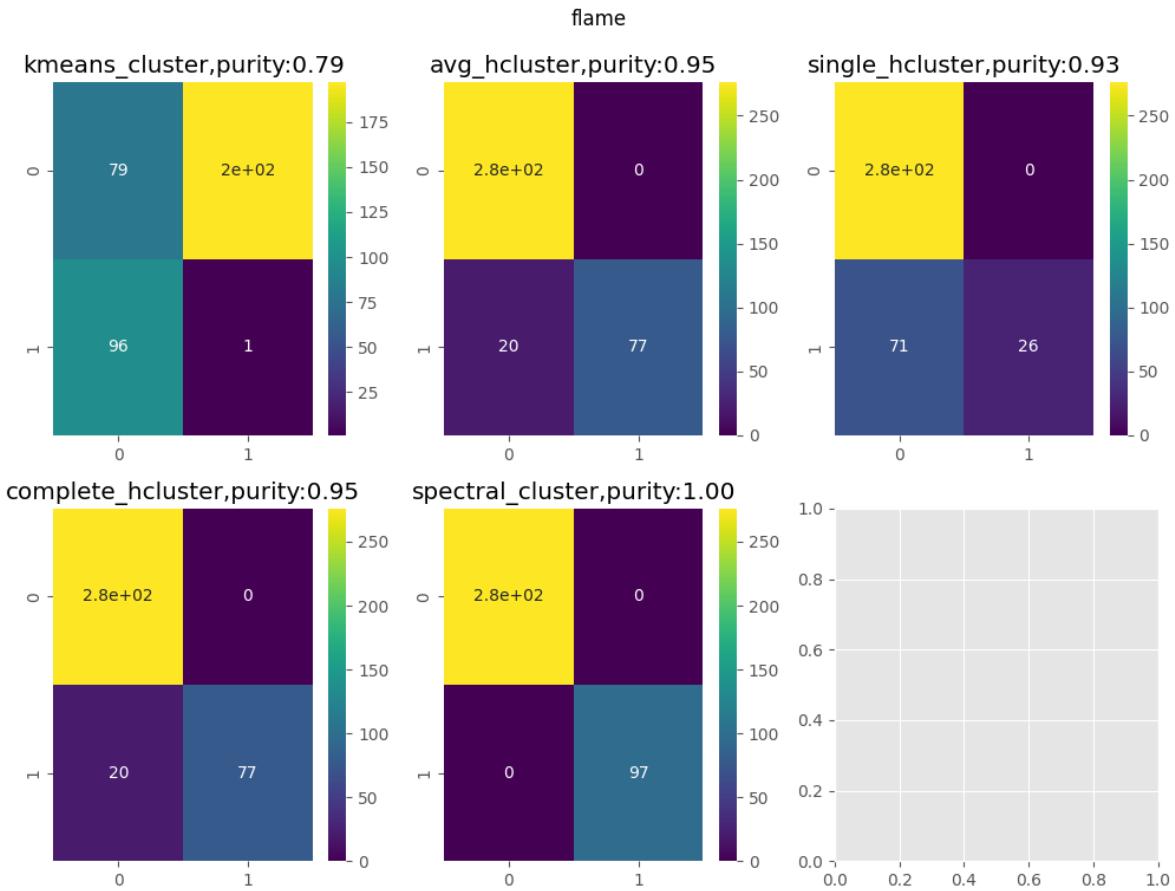
sns.heatmap(confusion_mat,ax=axs[i],annot=True,cmap="viridis")
fig.tight_layout(rect=[0, 0.03, 1, 0.95])

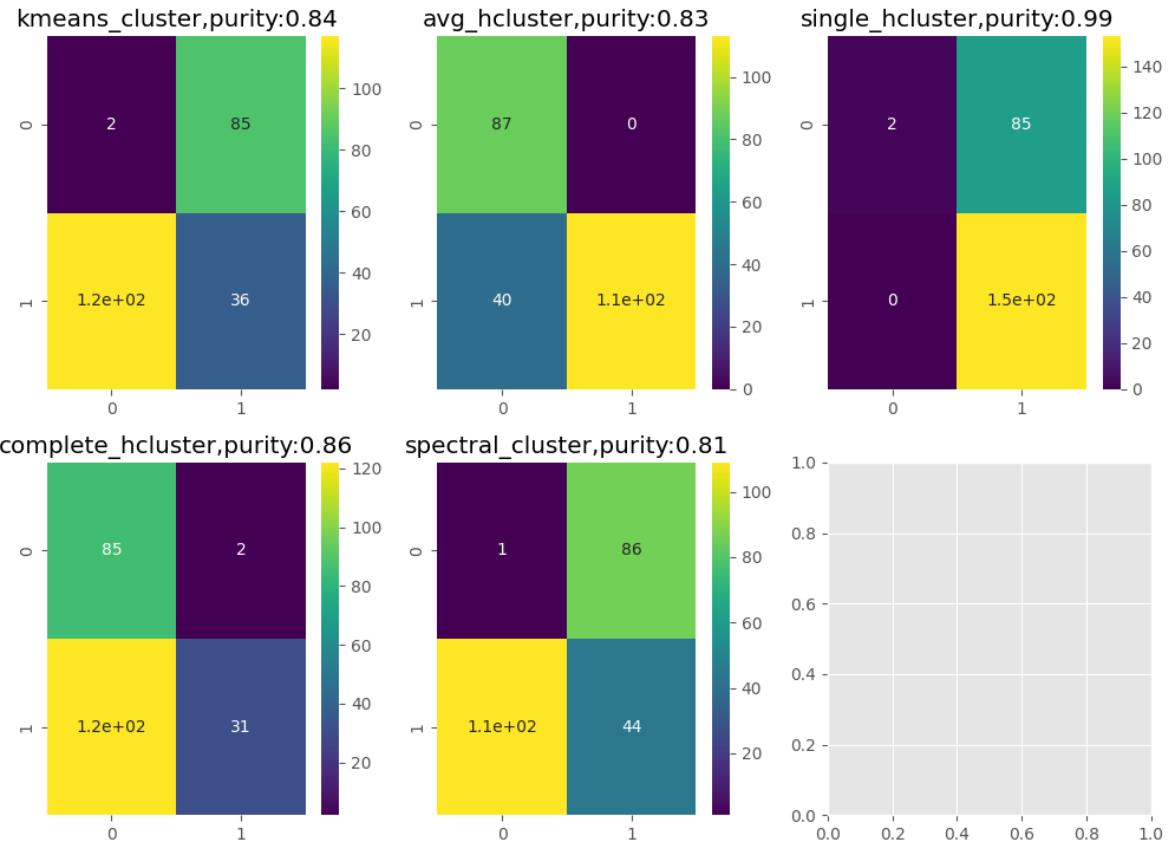
```

<Figure size 640x480 with 0 Axes>









Rand index :

Rand index [edit]

Definition [edit]

Given a set of n elements $S = \{o_1, \dots, o_n\}$ and two partitions of S to compare, $X = \{X_1, \dots, X_r\}$, a partition of S into r subsets, and $Y = \{Y_1, \dots, Y_s\}$, a partition of S into s subsets, define the following:

- a , the number of pairs of elements in S that are in the same subset in X and in the same subset in Y
- b , the number of pairs of elements in S that are in different subsets in X and in different subsets in Y
- c , the number of pairs of elements in S that are in the same subset in X and in different subsets in Y
- d , the number of pairs of elements in S that are in different subsets in X and in the same subset in Y

The Rand index, R , is:[1][2]

$$R = \frac{a + b}{a + b + c + d} = \frac{a + b}{\binom{n}{2}}$$

Intuitively, $a + b$ can be considered as the number of agreements between X and Y and $c + d$ as the number of disagreements between X and Y .

Since the denominator is the total number of pairs, the Rand index represents the frequency of occurrence of agreements over the total pairs, or the probability that X and Y will agree on a randomly chosen pair.

$\binom{n}{2}$ is calculated as $n(n - 1)/2$.

Similarly, one can also view the Rand index as a measure of the percentage of correct decisions made by the algorithm. It can be computed using the following formula:

Similarly, one can also view the Rand index as a measure of the percentage of correct decisions made by the algorithm. It can be computed using the following formula:

$$RI = \frac{TP + TN}{TP + FP + FN + TN}$$

where TP is the number of true positives, TN is the number of true negatives, FP is the number of false positives, and FN is the number of false negatives.

```
from sklearn.metrics.cluster import rand_score
methods=S1.columns[2:]
for df_name in dfs.keys():
    dataset=dfs[df_name][0]
    k = dfs[df_name][1]
    N= len(dataset)
    fig,axs=plt.subplots(1,1,figsize=(10,8))
    randind_mat=np.zeros((5,5))
    for i in range(5):
        for j in range(5):
            randind_mat[i,j]= rand_score(dataset.loc[:,methods[i]],dataset.loc[:,methods[j]])
```

(right) algorithms. The calculated Adjusted Rand index for these two clusterings is $ARI \approx 0.94$

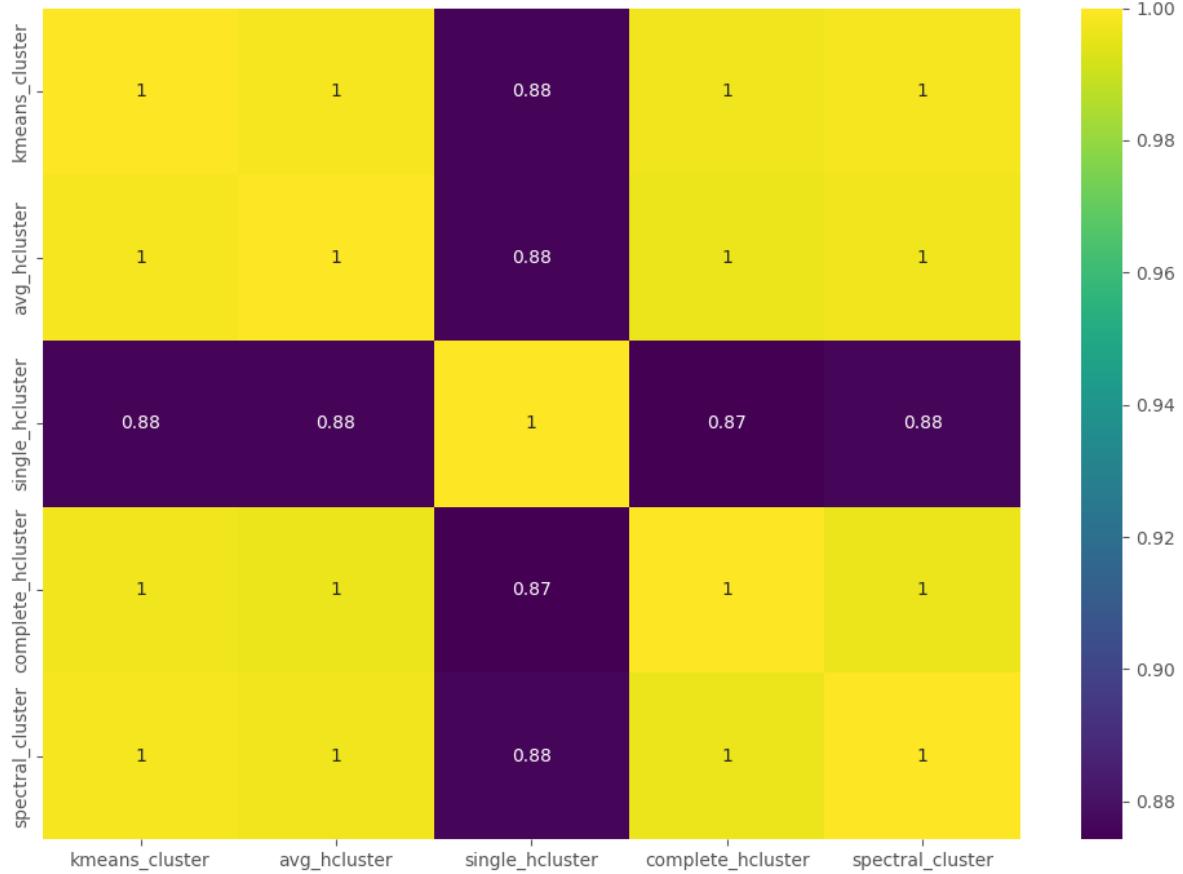
```

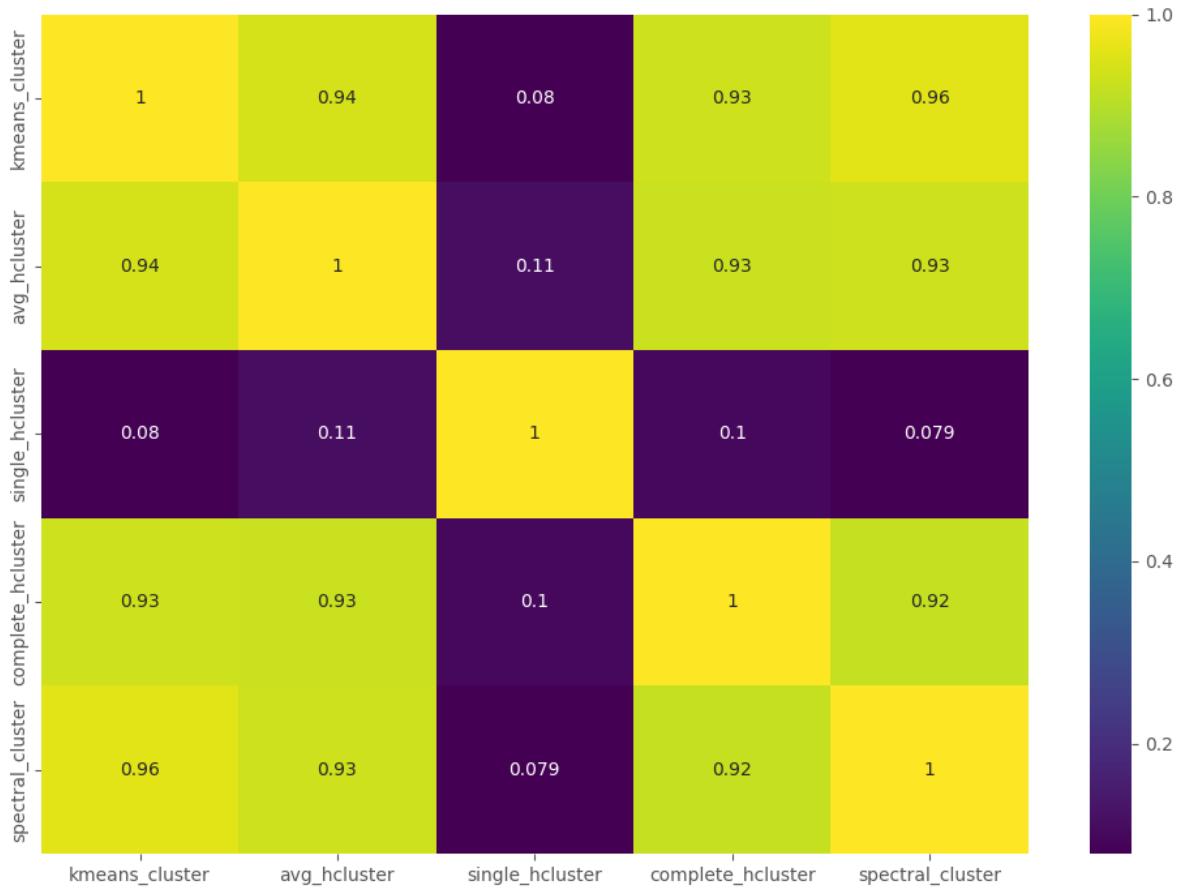
sns.heatmap(randind_mat, annot=True, cmap="viridis", xticklabels=methods, yticklabels=methods)
fig.suptitle(f'{df_name}')

fig.tight_layout(rect=[0, 0.03, 1, 0.95])

```

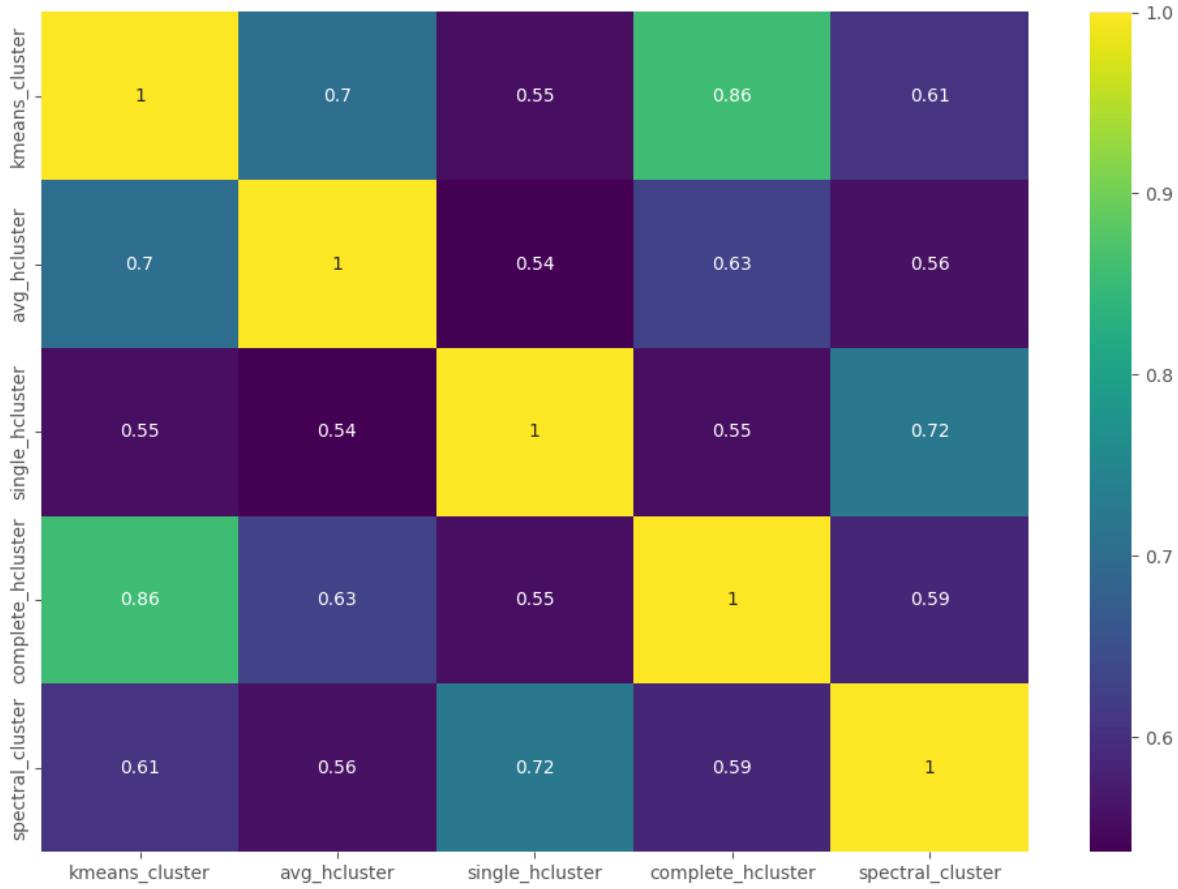
S1



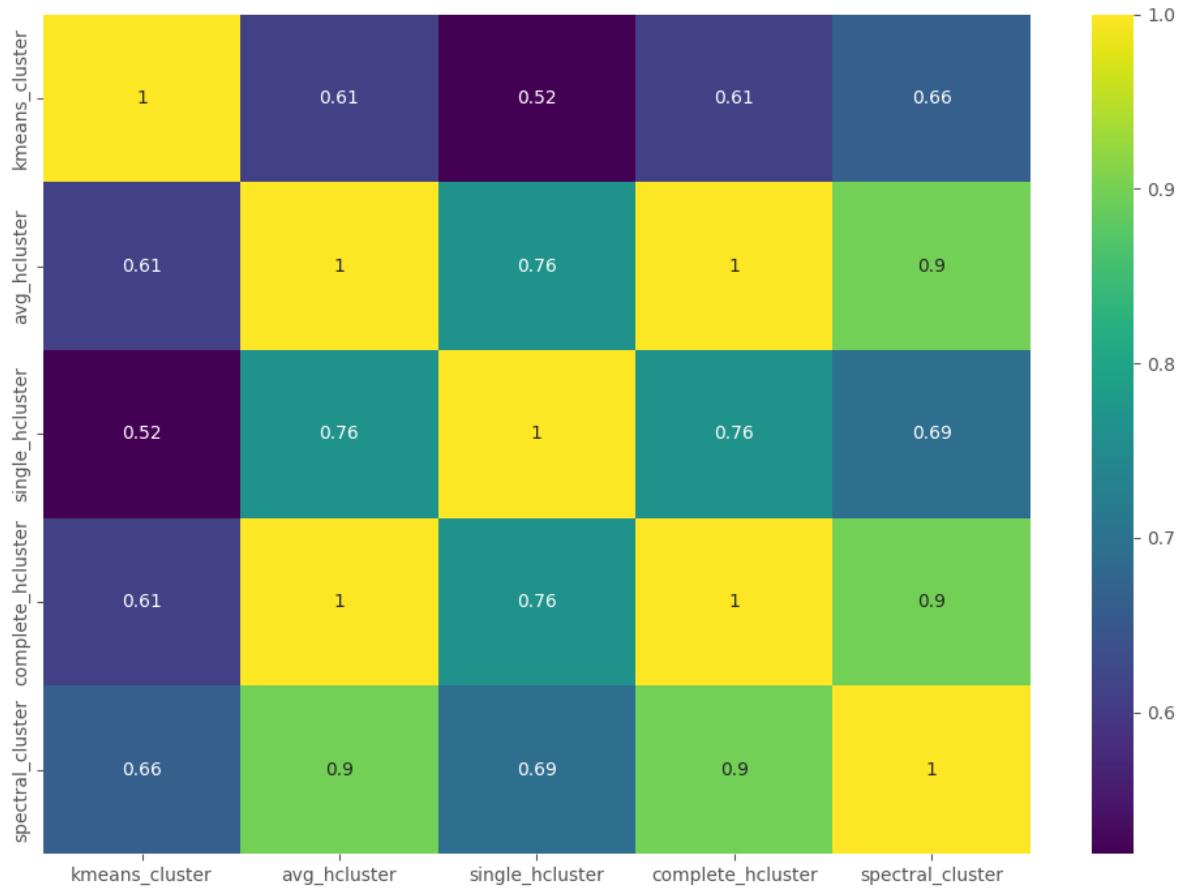




spiral



jain





References

Websites:

- 1- https://www.w3schools.com/python/python_ml_k-means.asp
- 2- https://www.w3schools.com/python/python_ml_hierarchical_clustering.asp
- 3- <https://scikit-learn.org/stable/modules/generated/sklearn.cluster.SpectralClustering.html>
- 4- https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html
- 5- <https://stats.stackexchange.com/questions/95731/how-to-calculate-purity>

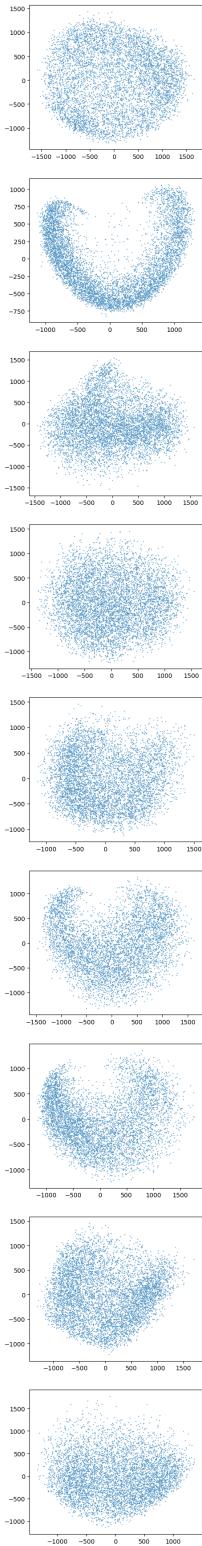
References

```
from keras.datasets import mnist
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

WARNING:tensorflow:From D:\Python3.11\Lib\site-packages\keras\src\losses.py:2976: The name t

(X_train, y_train), (X_test, y_test) = mnist.load_data()

from sklearn.decomposition import PCA
fig,axs=plt.subplots(9,1,figsize=(5,45))
axs.ravel()
for i in range(9):
    set=sets[i]
    #print(set.shape)
    function= lambda x: x.flatten()
    flat= np.array(list(map(function,set)))
    #print(flat.shape)
    pca = PCA()
    transformed_data=pca.fit_transform(flat)
    sns.scatterplot(x=transformed_data[:,0],y=transformed_data[:,1],ax=axs[i],s=2)
```



```

for set in sets:

    function= lambda x: x.flatten()

    flat= np.array(list(map(function,set)))
    pca = PCA(n_components=2)
    transformed_data=pca.fit_transform(flat)
    x = transformed_data[:,0] #PC1
    y = transformed_data[:,1] #PC2


    # Assuming `points` is a 2D numpy array where each row is a point

    # Create a 5x5 grid
    x_grid = np.linspace(x.min(), x.max(), 6)
    y_grid = np.linspace(y.min(), y.max(), 6)

    # Store the selected points
    selected_points = []

    # Loop over the grid and find the closest point to the center of each cell
    closest_point_indices=[]
    for j in range(5):
        for i in range(5):
            x_center = (x_grid[i] + x_grid[i+1]) / 2
            y_center = (y_grid[j] + y_grid[j+1]) / 2
            distances = np.sqrt((x - x_center)**2 + (y - y_center)**2)
            closest_point_index = np.argmin(distances)
            closest_point_indices.append(closest_point_index)
            selected_points.append(transformed_data[closest_point_index])

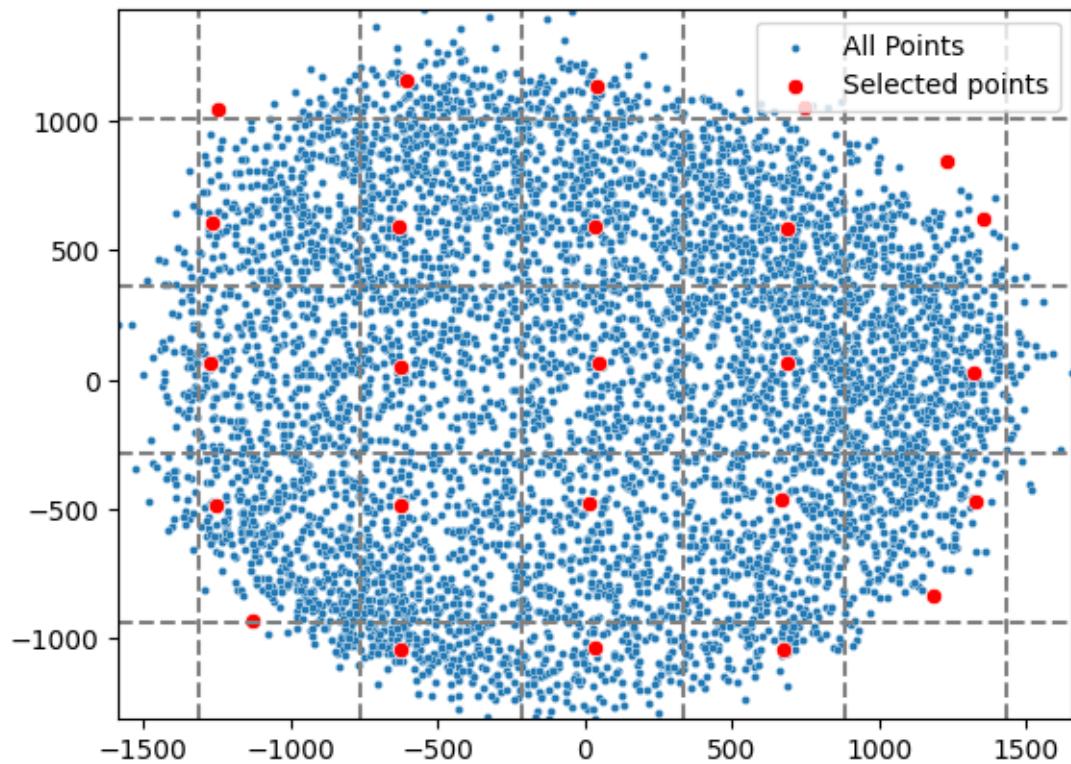
    # Convert the list to a numpy array
    selected_points = np.array(selected_points)
    for i in range(6):
        plt.axhline(x_grid[i],ls='--',c='gray')
        plt.axvline(y_grid[i],ls='--',c='gray')

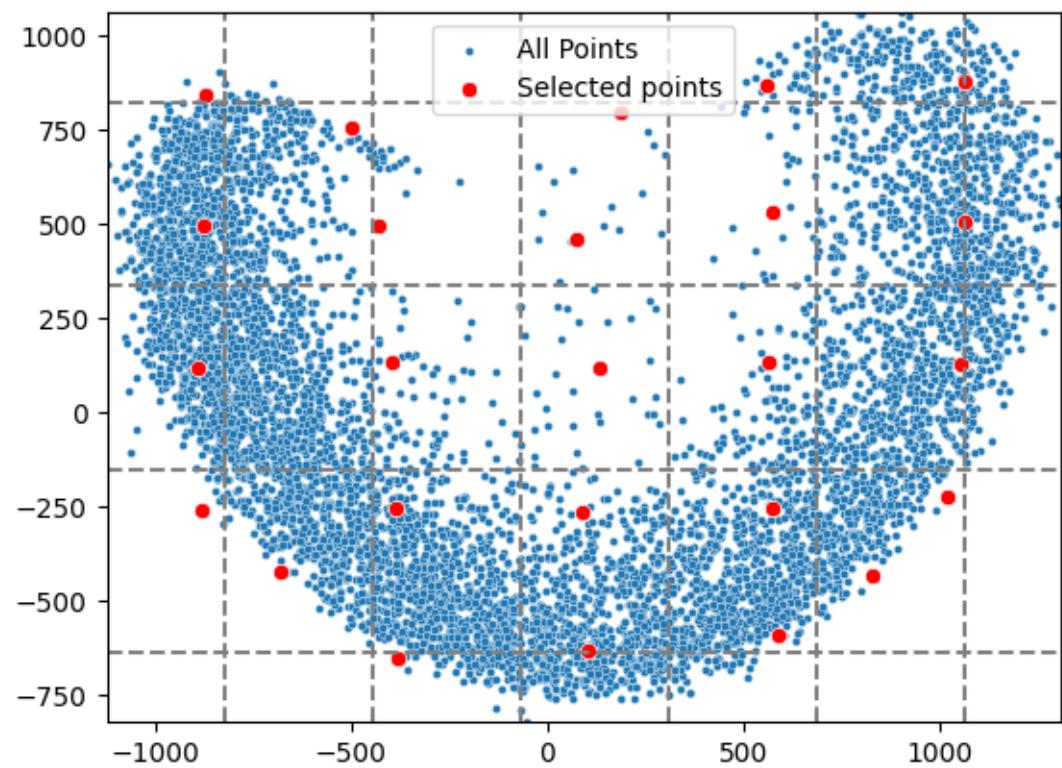
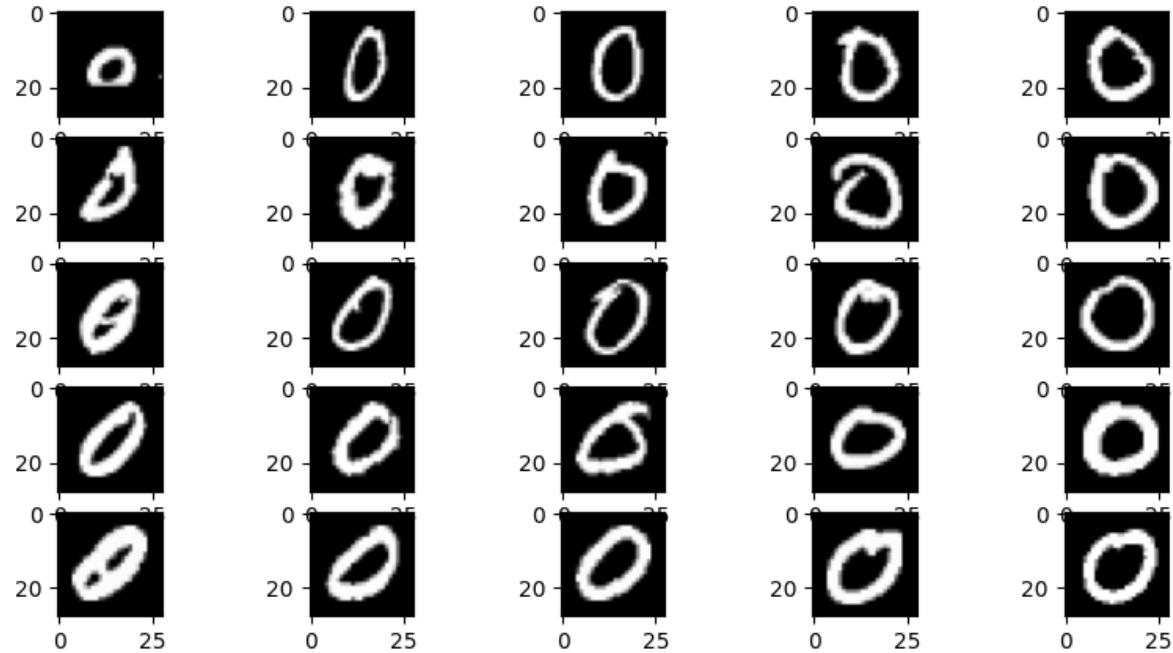
    # Plot all points
    sns.scatterplot(x=x,y=y,s=10,label="All Points")

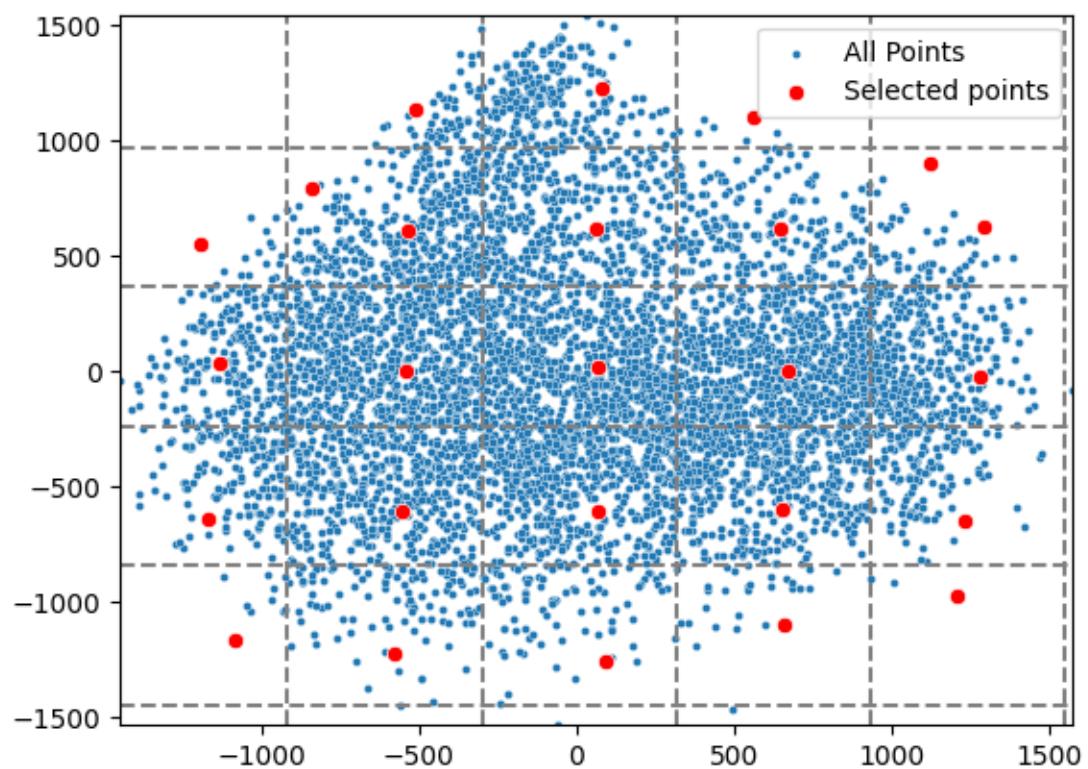
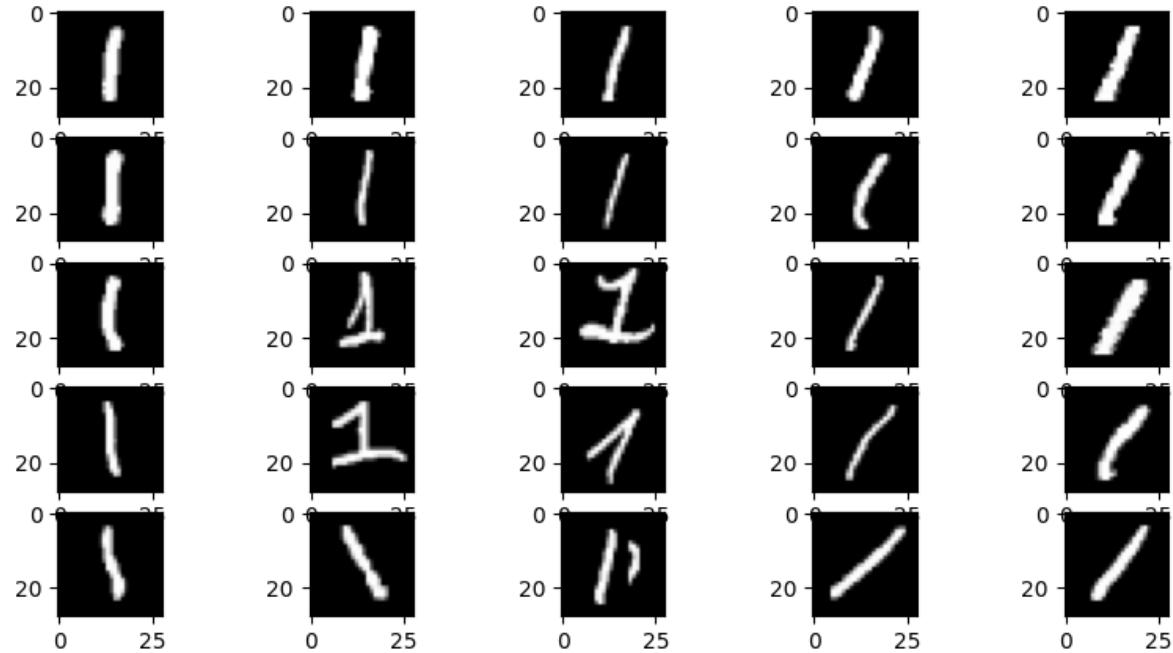
```

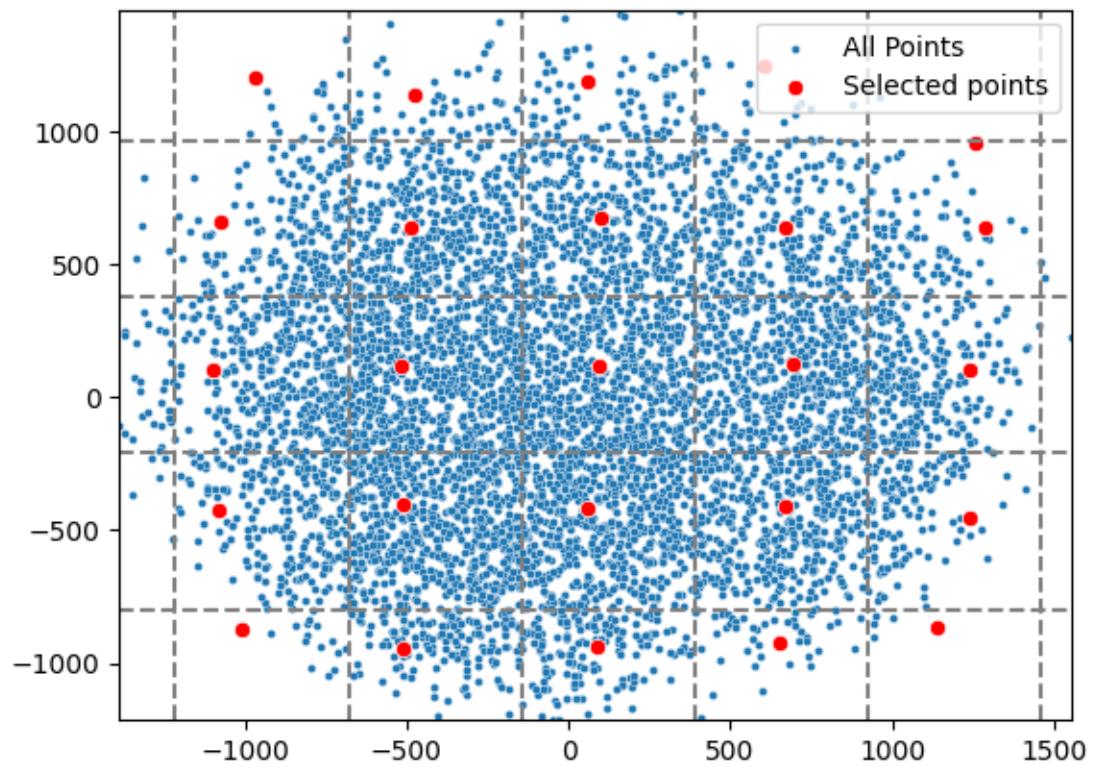
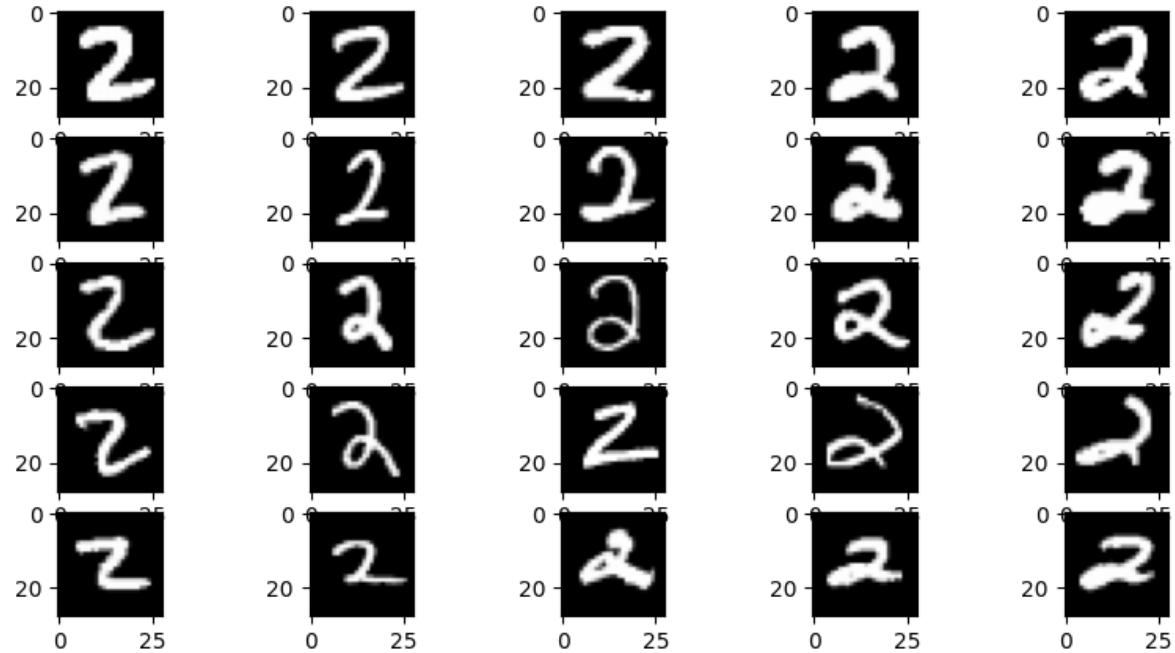
```
# Plot the selected points
sns.scatterplot(x=selected_points[:, 0],y= selected_points[:, 1], color='red', label='Selected points')
plt.xlim(x.min(),x.max())
plt.ylim(y.min(),y.max())
fig,axs=plt.subplots(5,5,figsize=(10,5))
axs=axs.ravel()
selected=set[closest_point_indices,:,:,:]
for i in range(25):
    axs[i].imshow(selected[i],cmap='gray')

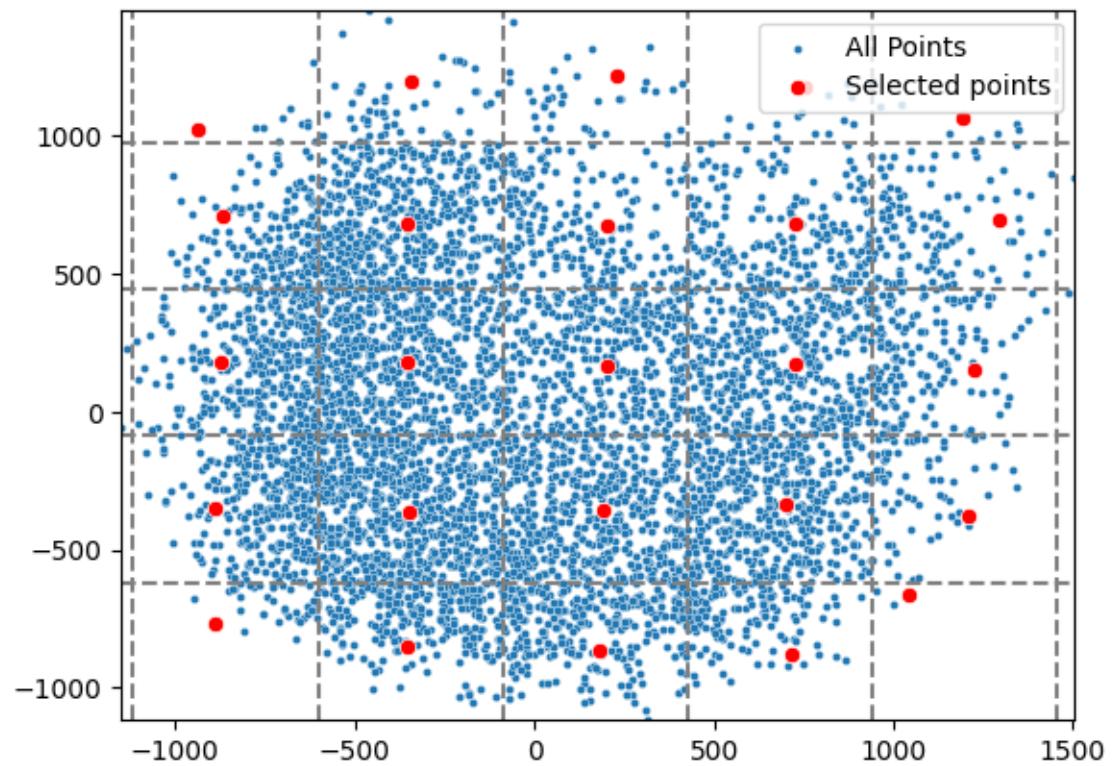
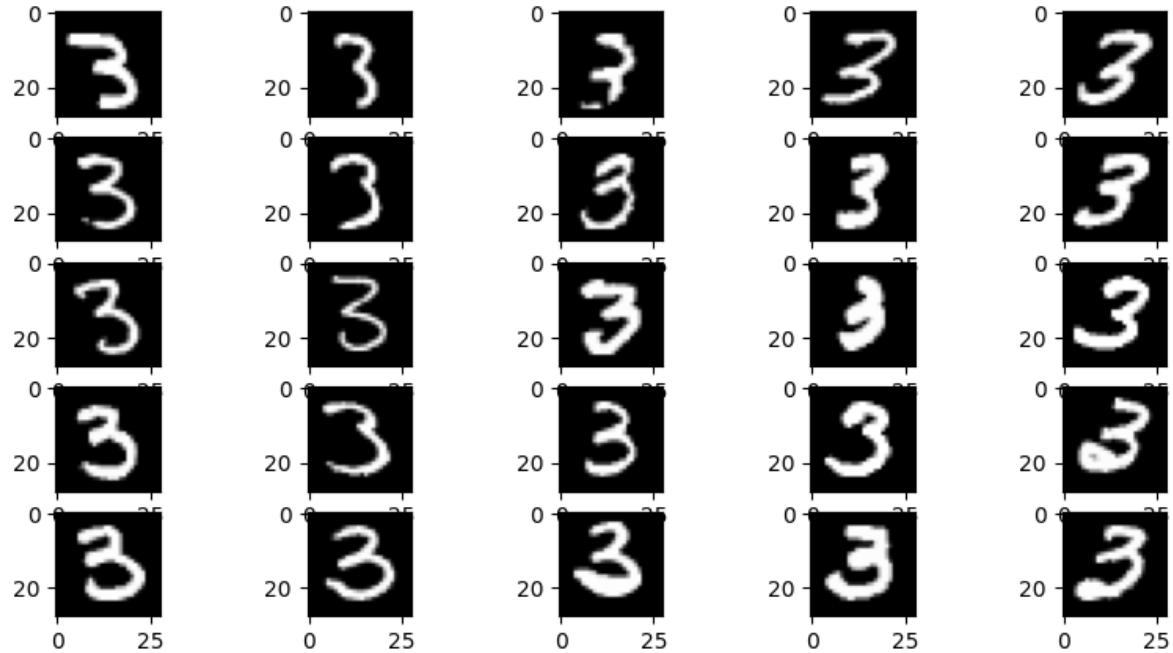
plt.show()
```

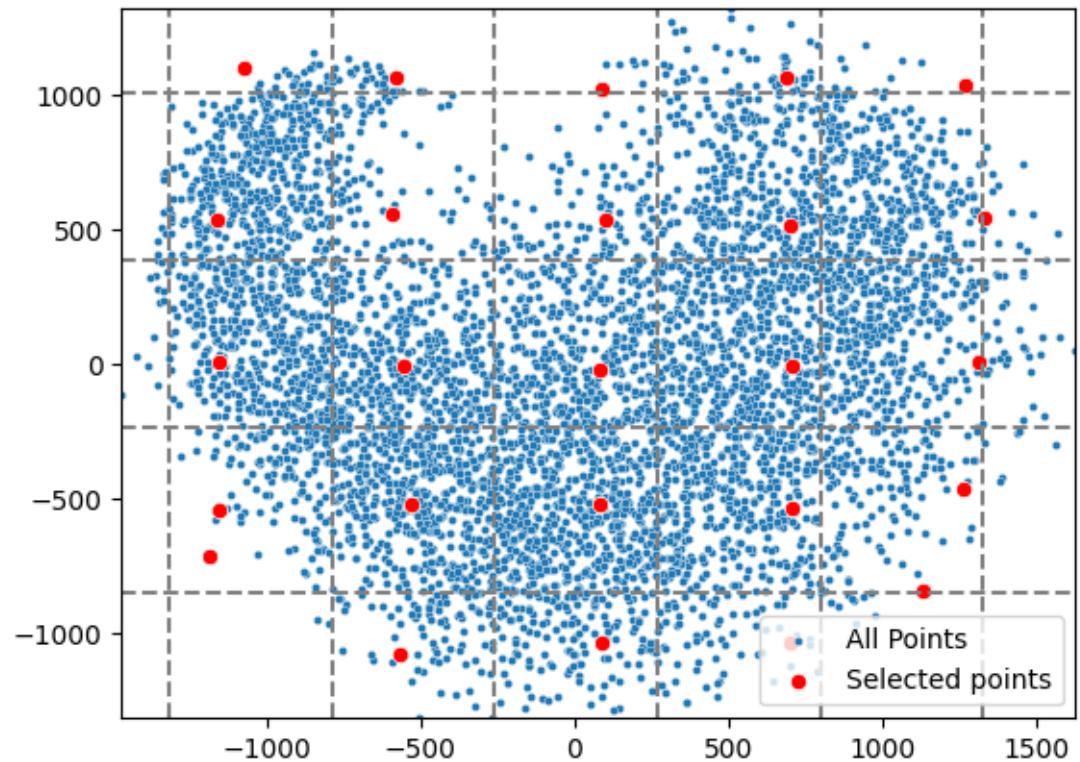
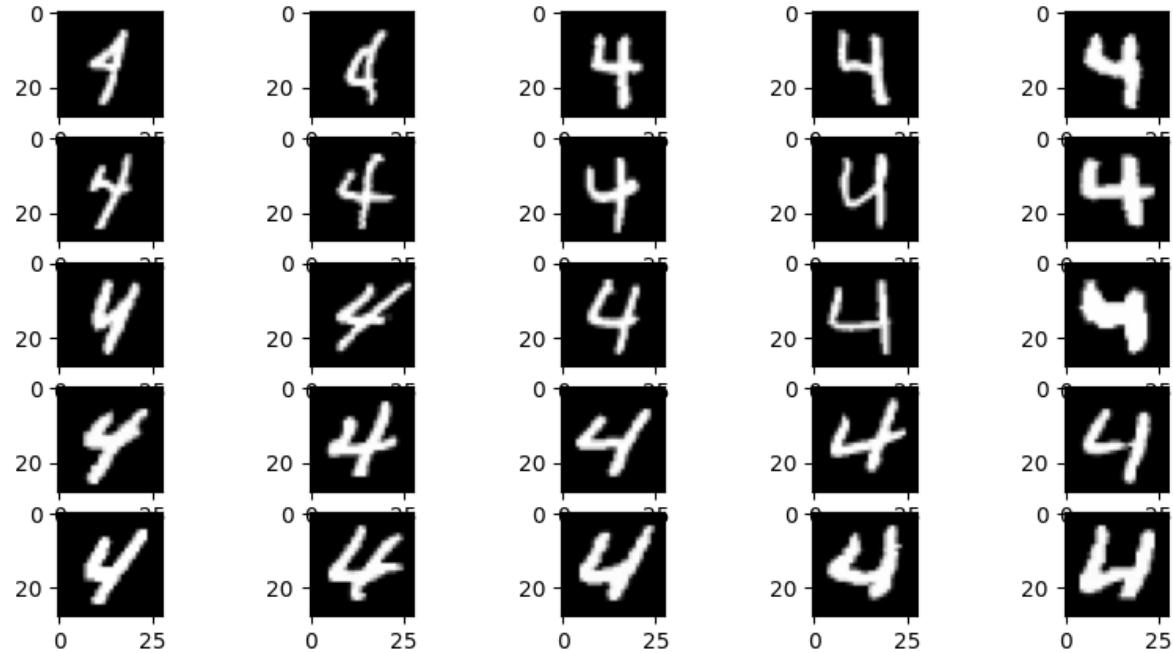


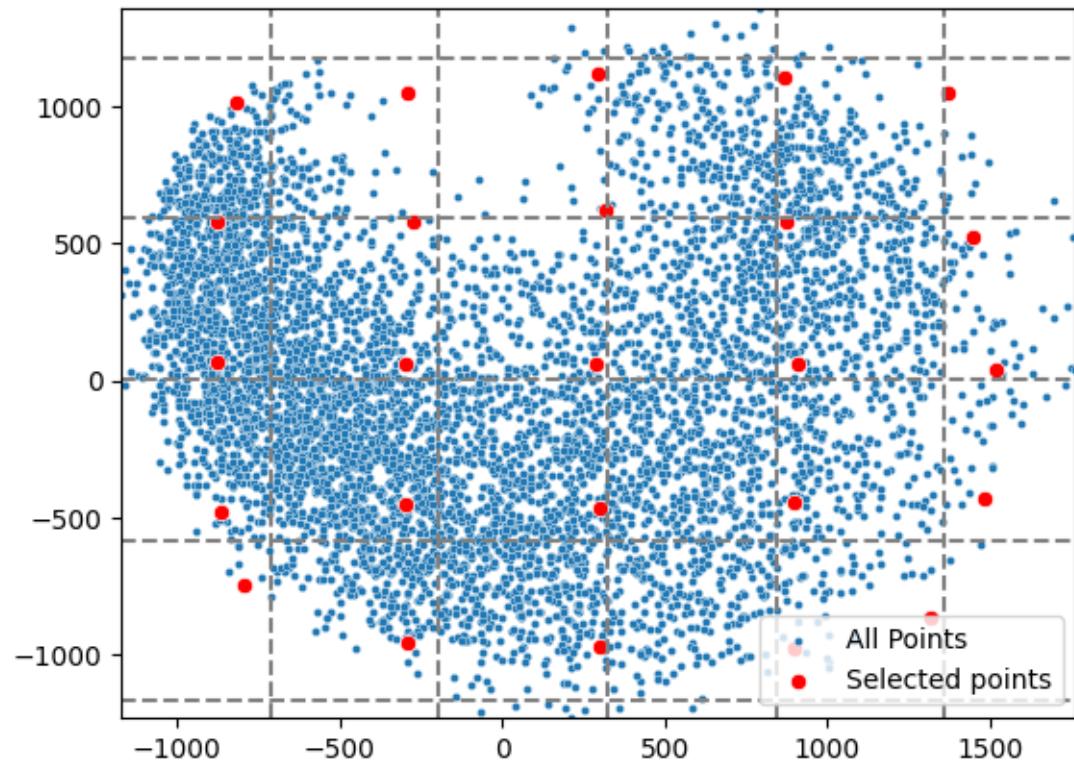
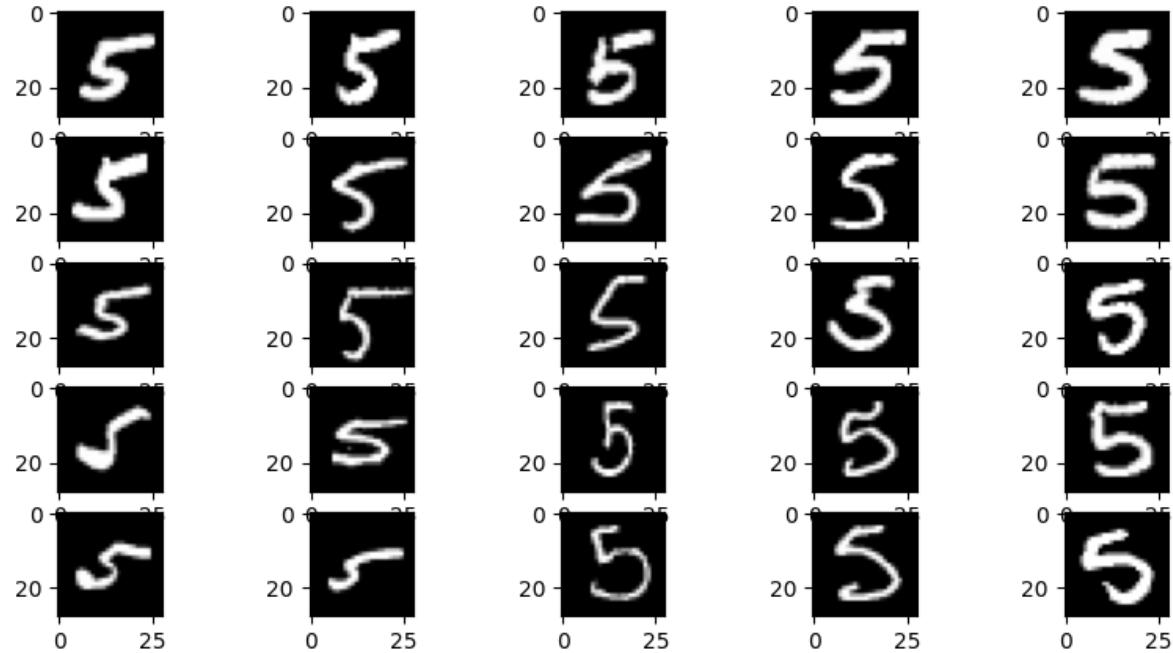


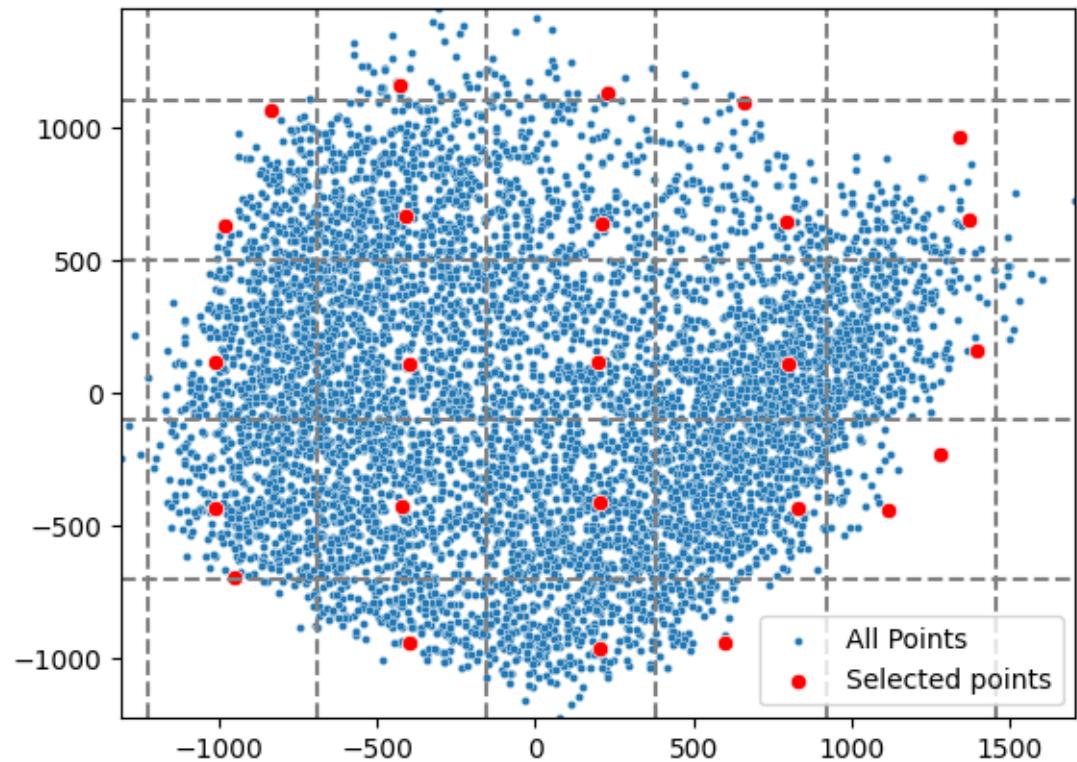
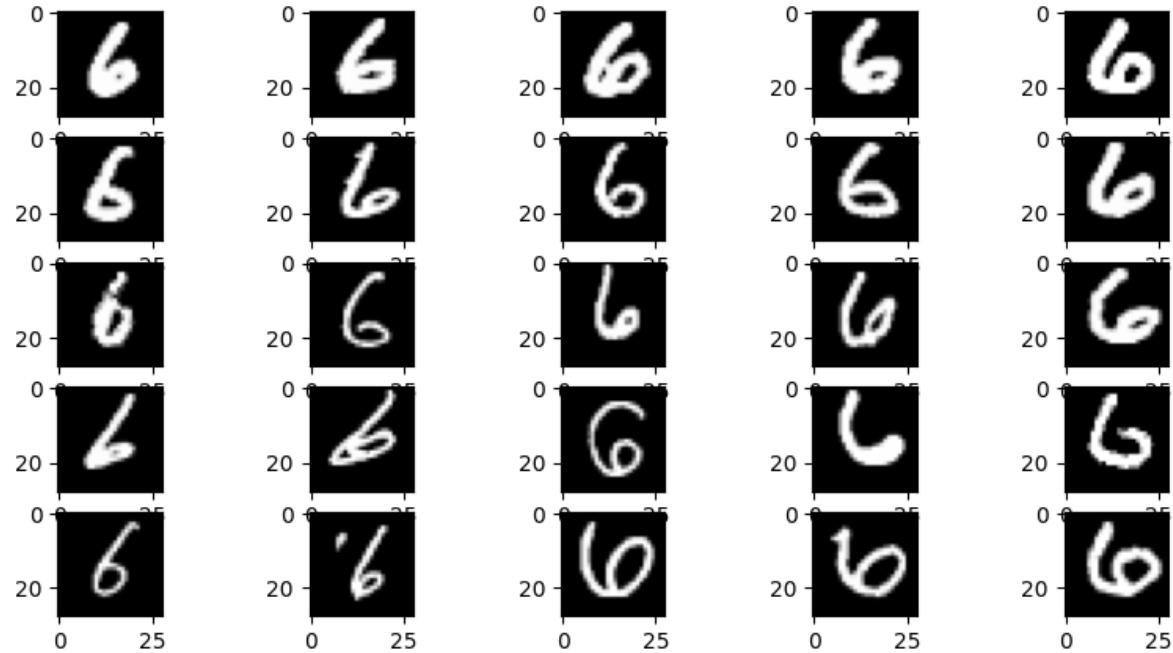


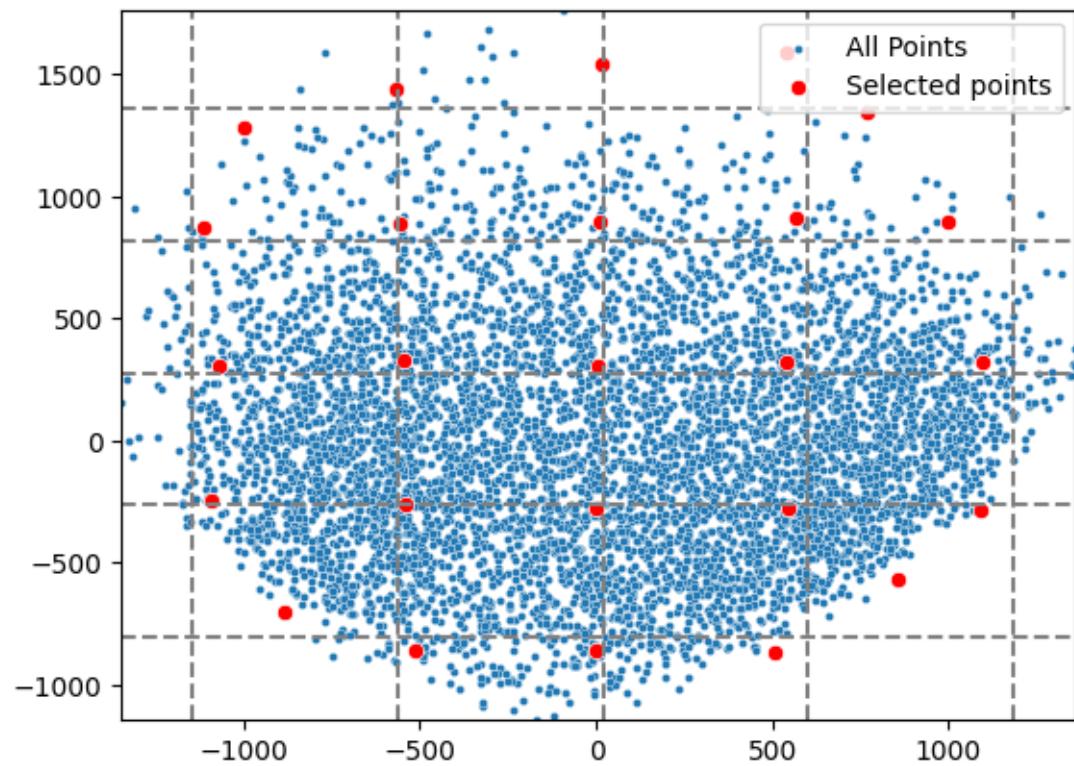
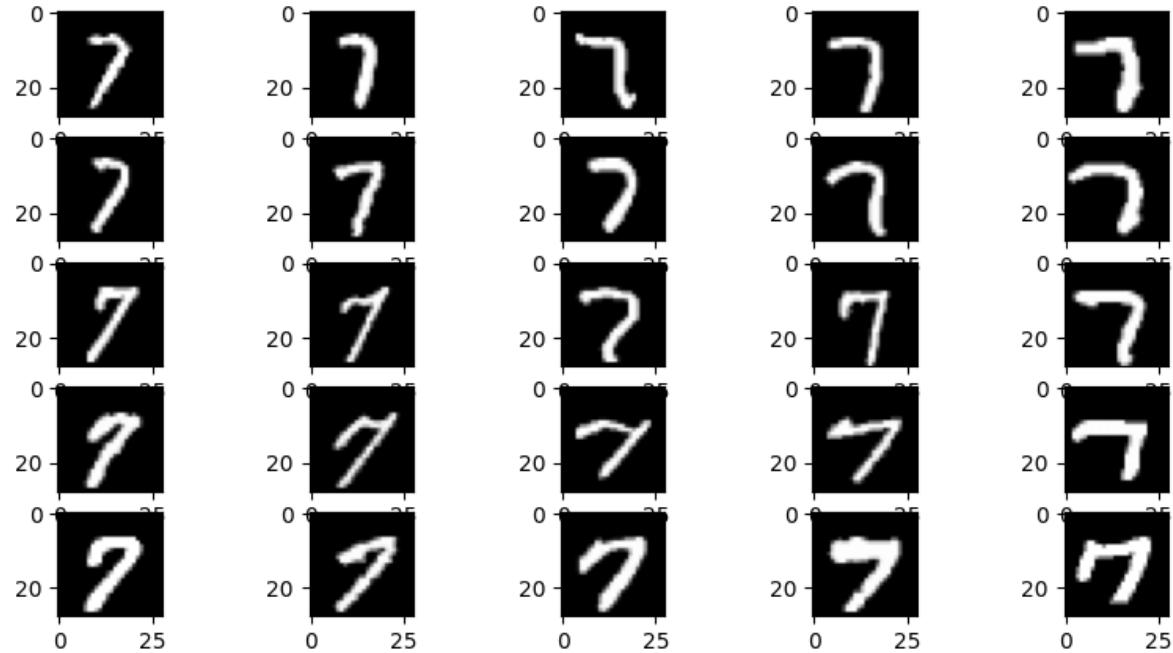


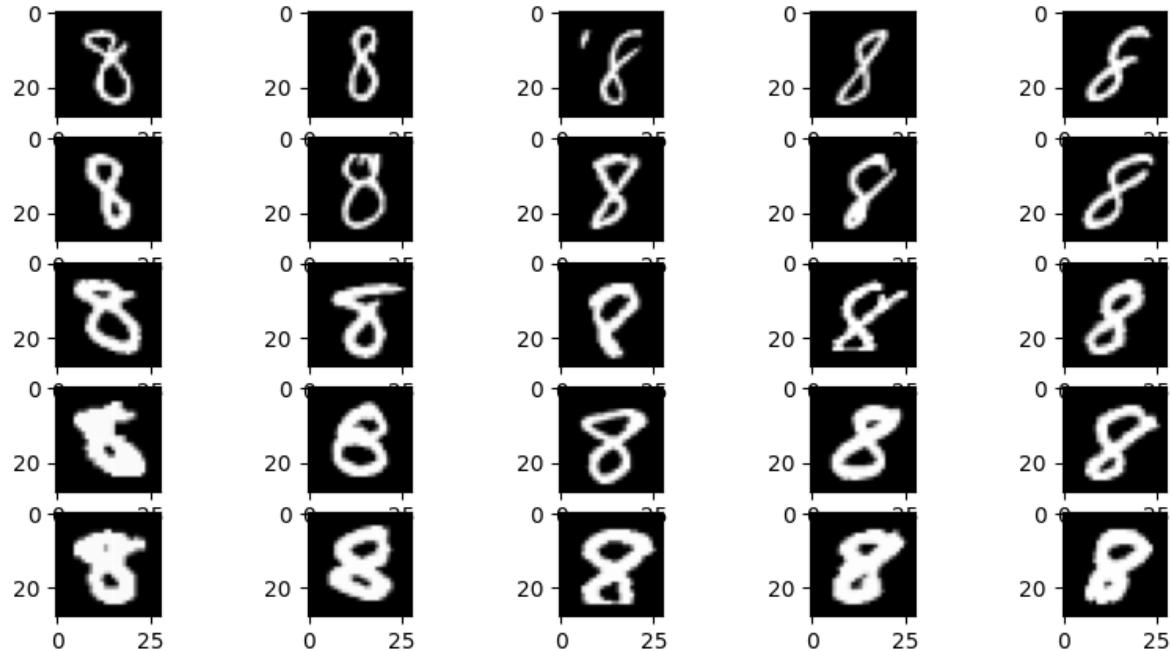












Websites:

1-

2-

PCA on Normalized and Raw data, Tissue and Project labeling

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

normalized_exp=pd.read_csv(r"expression_data/Leaf_Root_normalized_data.csv",index_col='Unnamed: 0')
raw_exp= pd.read_csv(r"expression_data/Leaf_Root_raw_data.csv",index_col='Unnamed: 0')
annot= pd.read_csv(r"expression_data/Leaf_Root_annotation.csv")
normalized_exp.loc[:, 'true_cluster']=np.array(annot.loc[annot['Sample']==normalized_exp.index,'Tissue'])
raw_exp.loc[:, 'true_cluster']=np.array(annot.loc[annot['Sample']==raw_exp.index,'Tissue'])
normalized_exp['true_cluster']=normalized_exp['true_cluster'].apply(lambda x:0 if x=="Root" else 1)
raw_exp['true_cluster']=normalized_exp['true_cluster'].apply(lambda x:0 if x=="Root" else 1)

normalized_exp.head()
```

	rna-gnl WGS:JAHRHJ evm2.model.Chr01.1	rna-gnl WGS:JAHRHJ evm2.model.Chr01.2	rrn-gnl WGS:JAHRHJ evm2.model.Chr01.3
SRR10913924	-0.271727	1.500167	-0.271727
SRR10913925	-0.271727	1.267299	-0.271727
SRR10913926	-0.271727	1.405773	-0.271727
SRR10913927	-0.271727	1.369847	-0.271727
SRR10913928	-0.271727	1.607823	-0.271727

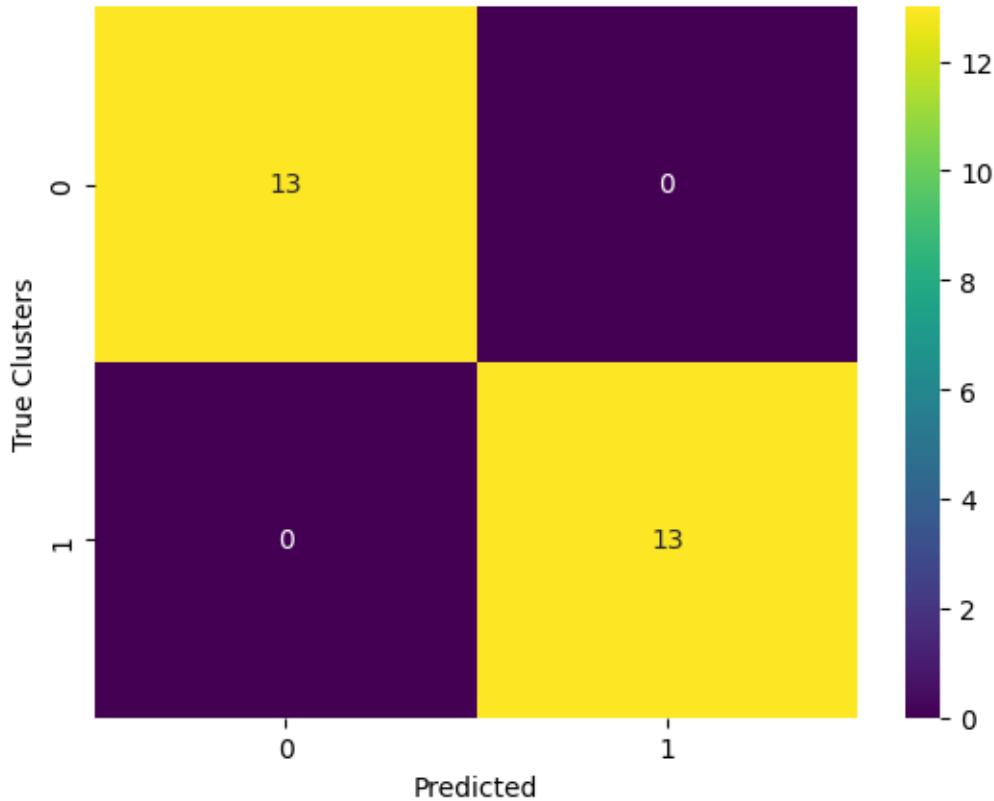
Normalized Data

Normalized: Tissue Label

```
from sklearn.cluster import KMeans
from sklearn.metrics import confusion_matrix,classification_report
tissue_label= pd.factorize(annot['Tissue'])[0]
project_label=pd.factorize(annot['Project'])[0]
k=2
kmeans=KMeans(n_clusters=k,n_init='auto')
kmeans.fit(normalized_exp.iloc[:,0:44770])
#column=f"kmeans_cluster_k{k}"
#normalized_exp[column]=kmeans.labels_
predicted=kmeans.labels_
confusion_mat=confusion_matrix(tissue_label, kmeans.labels_)

hp=sns.heatmap(confusion_mat,annot=True,cmap="viridis")
hp.set(ylabel='True Clusters', xlabel='Predicted')

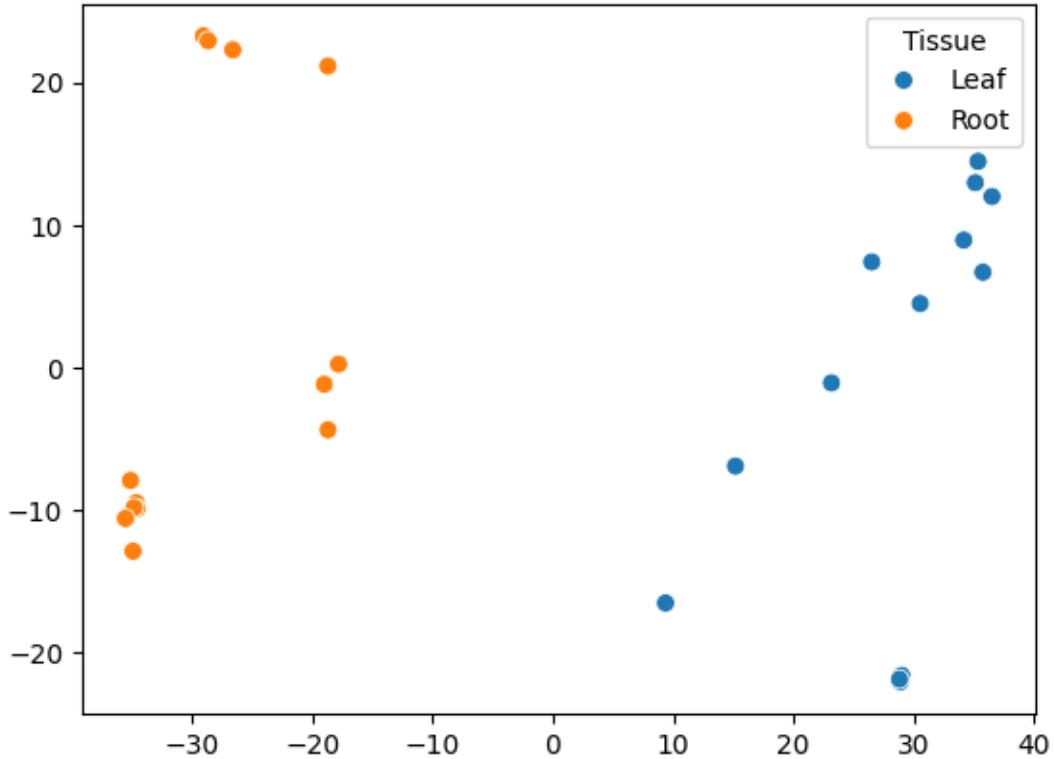
[Text(50.72222222222214, 0.5, 'True Clusters'),
 Text(0.5, 23.52222222222222, 'Predicted')]
```



```
from sklearn.decomposition import PCA
pca = PCA()
transformed_data=pca.fit_transform(normalized_exp.iloc[:,0:44770])

sns.scatterplot(x=transformed_data[:,0],y=transformed_data[:,1],s=50,hue=annot['Tissue'])
```

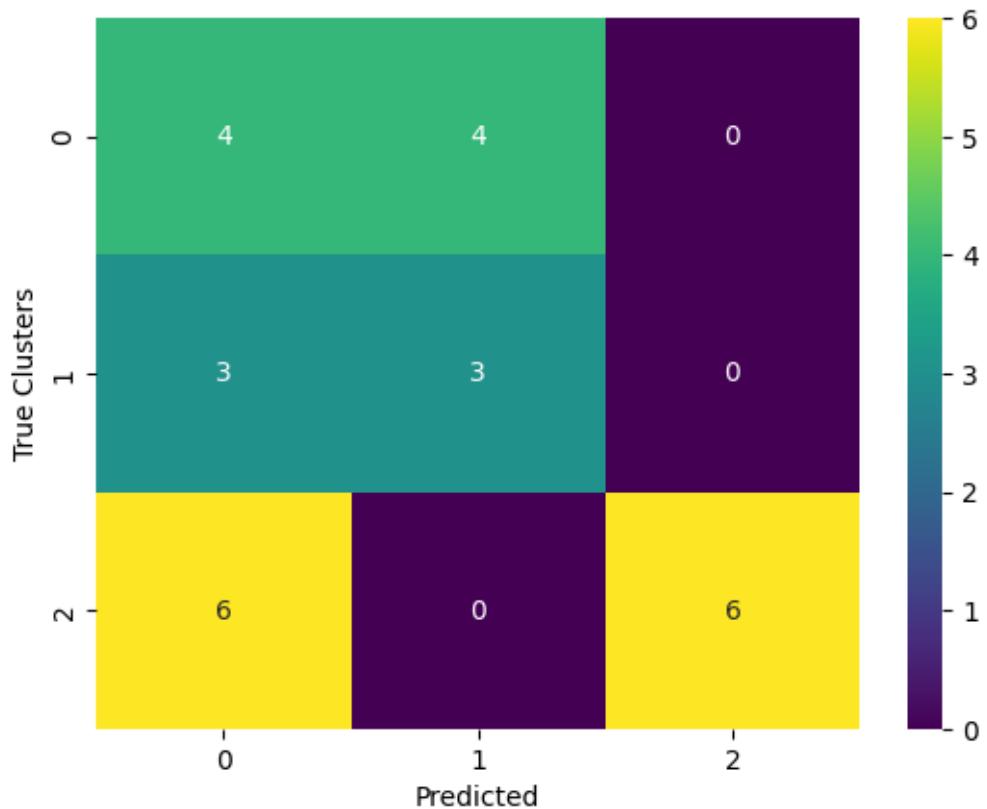
<Axes: >



Normalized: Project Label

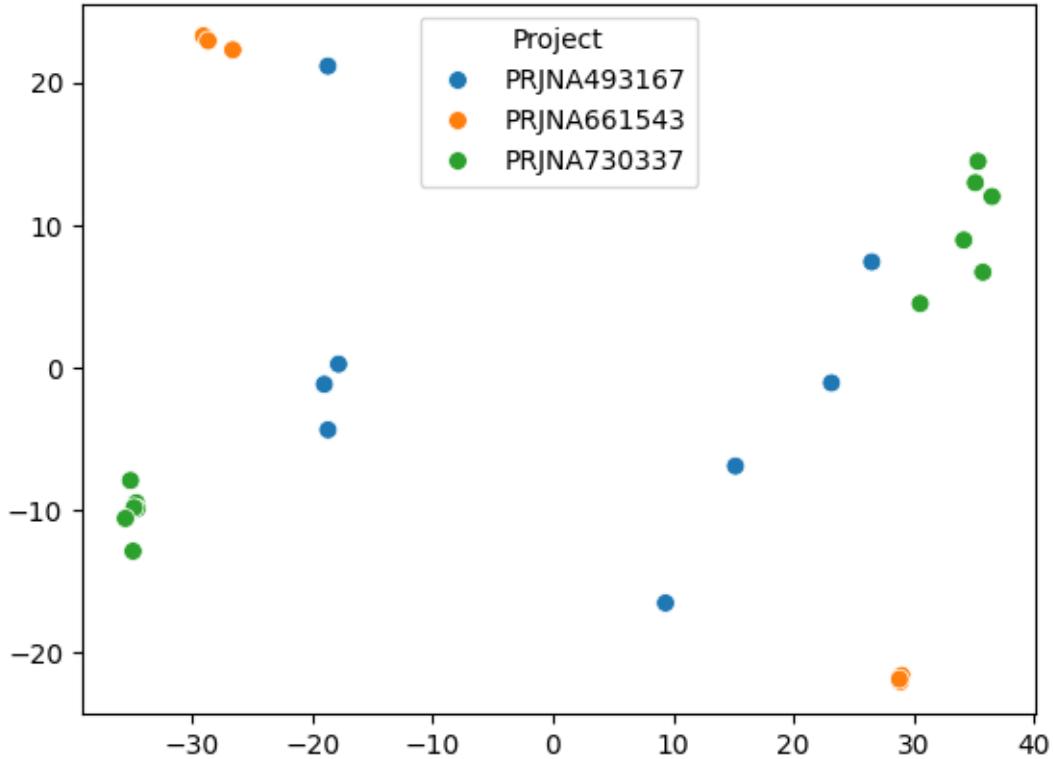
```
k=3
kmeans=KMeans(n_clusters=k,n_init='auto')
kmeans.fit(normalized_exp.iloc[:,0:44770])
#column=f"kmeans_cluster_k{k}"
predicted=kmeans.labels_
confusion_mat=confusion_matrix(project_label, predicted)
hp=sns.heatmap(confusion_mat,annot=True,cmap="viridis")
hp.set(ylabel='True Clusters', xlabel='Predicted')
```

```
[Text(50.72222222222214, 0.5, 'True Clusters'),
Text(0.5, 23.52222222222222, 'Predicted')]
```



```
sns.scatterplot(x=transformed_data[:,0],y=transformed_data[:,1],s=50,hue=annot['Project'])
```

```
<Axes: >
```



Raw Data

Raw: Tissue Label

```

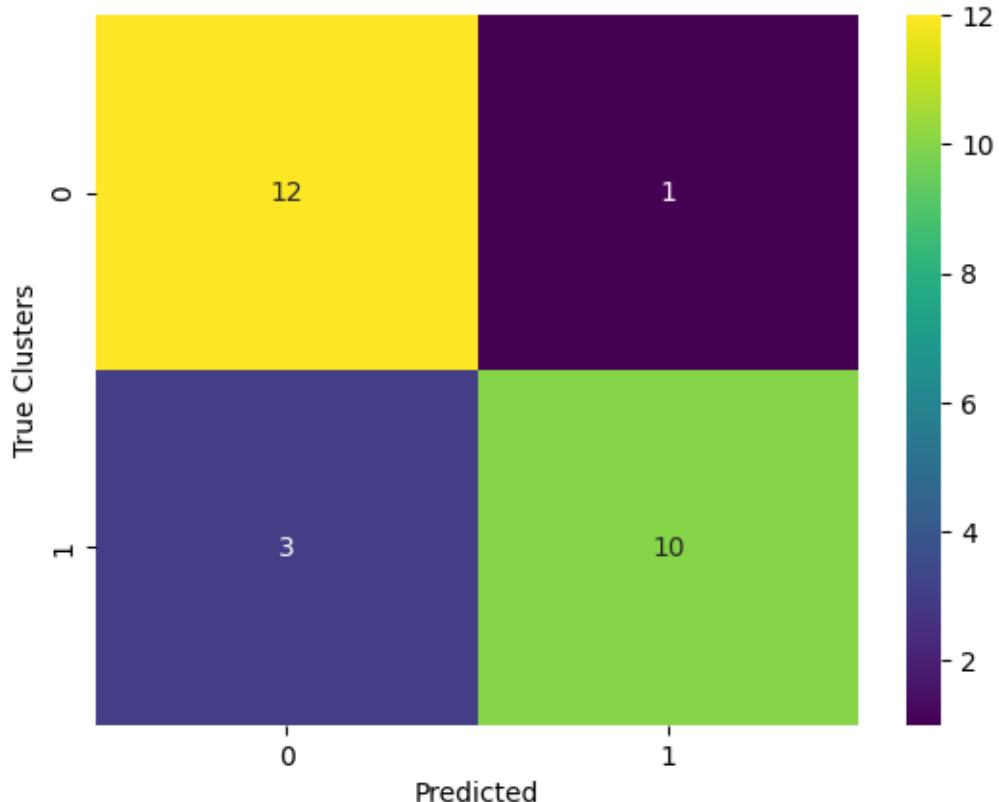
from sklearn.cluster import KMeans
from sklearn.metrics import confusion_matrix,classification_report
tissue_label= pd.factorize(annot['Tissue'])[0]
project_label=pd.factorize(annot['Project'])[0]
k=2
kmeans=KMeans(n_clusters=k,n_init='auto')
kmeans.fit(raw_exp.iloc[:,0:44770])
#column=f"kmeans_cluster_k{k}"
#normalized_exp[column]=kmeans.labels_
predicted=kmeans.labels_
confusion_mat=confusion_matrix(tissue_label, kmeans.labels_)

hp=sns.heatmap(confusion_mat,annot=True,cmap="viridis")

```

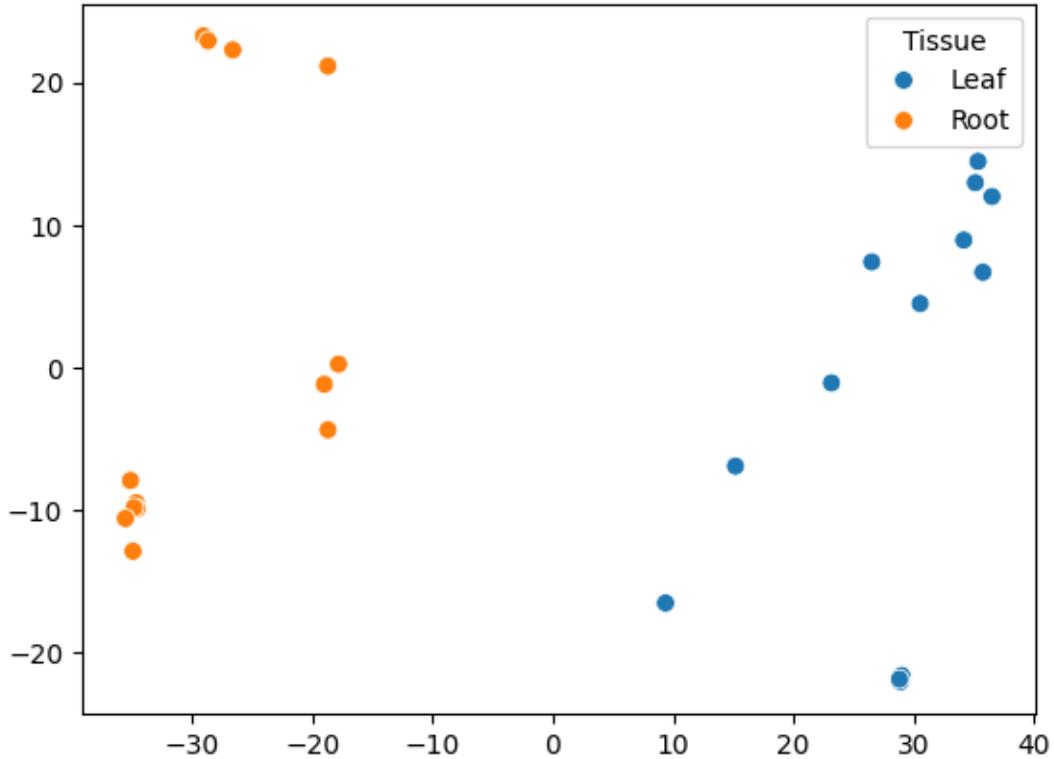
```
hp.set(ylabel='True Clusters', xlabel='Predicted')
```

```
[Text(50.72222222222214, 0.5, 'True Clusters'),  
Text(0.5, 23.52222222222222, 'Predicted')]
```



```
sns.scatterplot(x=transformed_data[:,0],y=transformed_data[:,1],s=50,hue=annot['Tissue'])
```

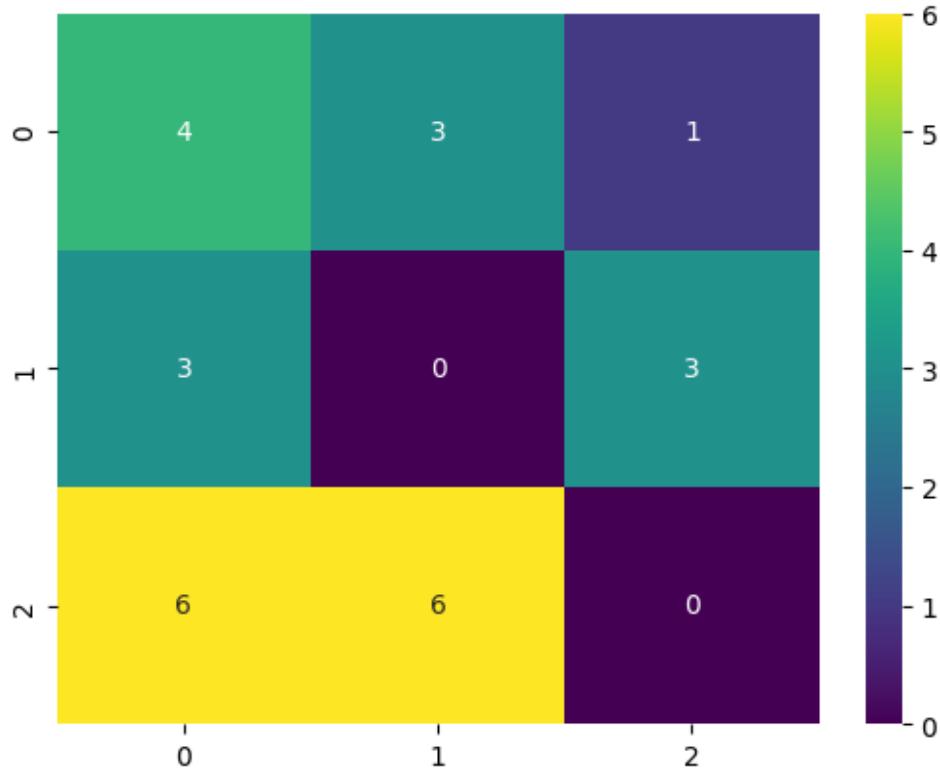
```
<Axes: >
```



Raw: Project Label

```
k=3
kmeans=KMeans(n_clusters=k,n_init='auto')
kmeans.fit(raw_exp.iloc[:,0:44770])
#column=f"kmeans_cluster_k{k}"
predicted=kmeans.labels_
confusion_mat=confusion_matrix(project_label, predicted)
sns.heatmap(confusion_mat, annot=True, cmap="viridis")
```

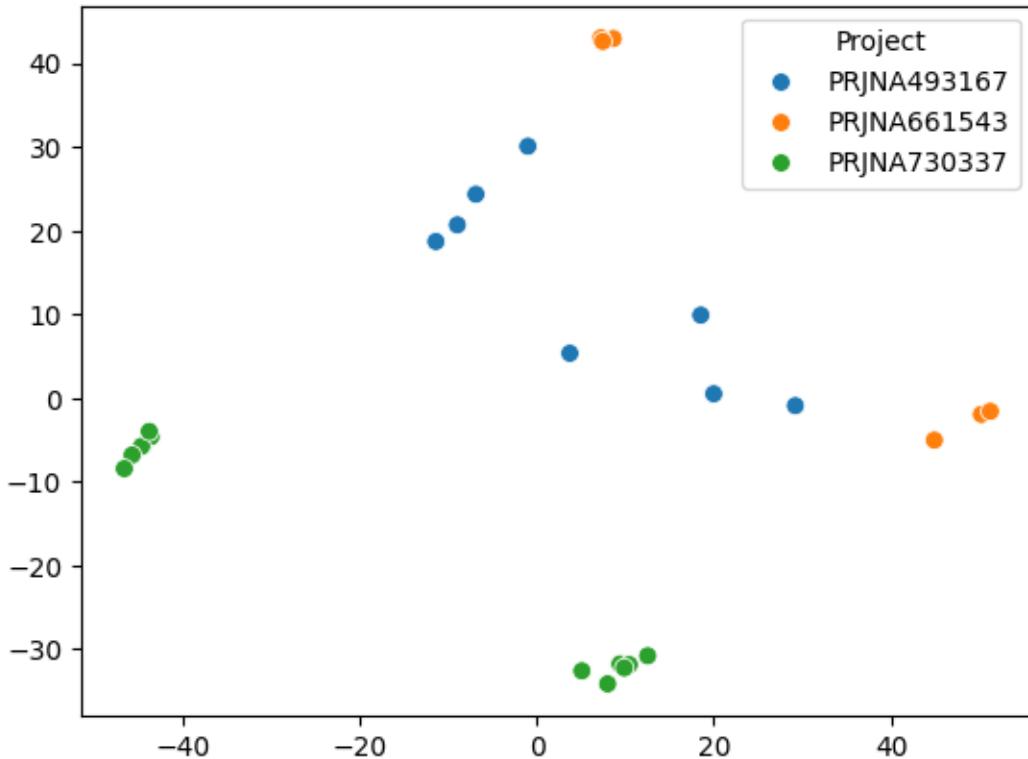
<Axes: >



```
transformed_data=pca.fit_transform(raw_exp.iloc[:,0:44770])
```

```
sns.scatterplot(x=transformed_data[:,0],y=transformed_data[:,1],s=50,hue=annot['Project'])
```

```
<Axes: >
```



References

Websites

ChatGPT

1- I want to apply Logistic regression on image data. (MNIST dataset from keras) . Do I have to flatten the features before hand? and convert 28*28 matrices to vectors? Yes, you're correct. When applying Logistic Regression to image data, such as the MNIST dataset, you need to flatten the 2D image matrices into 1D vectors¹².

The reason for this is that Logistic Regression, like many other machine learning algorithms, expects input data in a tabular form where each row represents an instance and each column represents a feature¹². In the case of image data, each pixel can be considered as a feature. So, a 28x28 pixel image from the MNIST dataset would be flattened into a 1D vector of 784 features (since $28 \times 28 = 784$).

Here's a simple example of how you might do this in Python:

Python

```
from keras.datasets import mnist import numpy## as np
```

Load the MNIST dataset

```
(train_images, train_labels), (test_images, test_labels) = mnist.lo##ad_data()
```

Flatten the images

```
train_images = train_images.reshape((60000, 28 * 28)) test_images = test_images.reshape((100##00, 28 * 28))
```

Convert to float and normalize

train_images = train_images.astype('float32') / 255 test_images = test_images.astype('float32') / 255 AI-generated code. Review and use carefully. More info on FAQ. In this code, we first load the MNIST dataset using Keras. Then, we reshape the images from 28x28 matrices into 1D vectors of length 784 using the reshape function. Finally, we normalize the pixel values to be between 0 and 1 by converting the data type to float32 and dividing by 255 (since pixel values range from 0 to 255).

Once you have your flattened features, you can then apply Logistic Regression or any other machine learning algorithm that expects input data in this format12.]

Questions:

Don't know how to interpret the results

Comparison of Different classification methods

```
from keras.datasets import mnist
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

WARNING:tensorflow:From D:\Python3.11\Lib\site-packages\keras\src\losses.py:2976: The name t

function= lambda x: x.flatten()
X_train= np.array(list(map(function,X_train)))
X_test= np.array(list(map(function,X_test)))

results= pd.DataFrame(data=y_test,columns=["true_value"])
```

Logistic Regression

```
from sklearn.linear_model import LogisticRegression
lg = LogisticRegression()
lg.fit(X_train,y_train)
predicted = lg.predict(X_test)
results['Log_Regression']=predicted
```

D:\Python3.11\Lib\site-packages\sklearn\linear_model_logistic.py:460: ConvergenceWarning: 11
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:

```
https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
n_iter_i = _check_optimize_result(
    predicted

array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)

y_train.shape

(60000,)
```

MLP

```
sklearn.neural_network.MLPClassifier(hidden_layer_sizes=(100,), activation='relu', *,
solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000)[source]¶
```

MLP with one hidden layer 128

```
from sklearn.neural_network import MLPClassifier
clf = MLPClassifier(alpha=1e-5,
                     hidden_layer_sizes=(128, ), random_state=1)
clf.fit(X_train,y_train)
predicted=clf.predict(X_test)
results['MLP_1']=predicted
```

MLP with two hidden layers 256 - 128

```
clf = MLPClassifier( alpha=1e-5,
                     hidden_layer_sizes=(128,256), random_state=1)
clf.fit(X_train,y_train)
predicted=clf.predict(X_test)
results['MLP_2']=predicted

results
```

	true_value	Log_Regression	MLP_1	MLP_2
0	7	7	7	7
1	2	2	2	2
2	1	1	1	1
3	0	0	0	0
4	4	4	4	4
...
9995	2	2	2	2
9996	3	3	3	3
9997	4	4	4	4
9998	5	5	5	5
9999	6	6	6	6

CNN

```
import torch
import torchvision
import torchvision.transforms as transforms

import torch.nn as nn
import torch.nn.functional as F

from torch.utils.data import DataLoader
loaders = {
    'train' : torch.utils.data.DataLoader(train_data,
                                          batch_size=100,
                                          shuffle=True,
```

```

        num_workers=1) ,

'test' : torch.utils.data.DataLoader(test_data,
                                      batch_size=100,
                                      shuffle=True,
                                      num_workers=1),
}

loaders

{'train': <torch.utils.data.dataloader.DataLoader at 0x1c289a791d0>,
 'test': <torch.utils.data.dataloader.DataLoader at 0x1c2899ea410>}

```

```

import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

```

Importing data

```

from torchvision import datasets
from torchvision.transforms import ToTensor
train_data = datasets.MNIST(
    root = 'data',
    train = True,
    transform = ToTensor(),
    download = True,
)
test_data = datasets.MNIST(
    root = 'data',
    train = False,
    transform = ToTensor()
)

from torch.utils.data import DataLoader
loaders = {
    'train' : torch.utils.data.DataLoader(train_data,
                                          batch_size=100,
                                          shuffle=False,
                                          num_workers=1),

```

Section 6.3.7

10.2.5 Multi-dimensional convolutions

So far we have considered convolutions over a single grey-scale image. For a colour image there will be three channels corresponding to the red, green, and blue colours. We can easily extend convolutions to cover multiple channels by extending the dimensionality of the filter. An image with $J \times K$ pixels and C channels will be described by a *tensor* of dimensionality $J \times K \times C$. We can introduce a filter described by a tensor of dimensionality $M \times M \times C$ comprising a separate $M \times M$ filter for each of the C channels. Assuming no padding and a stride of 1, this again gives a feature map of size $(J-M+1) \times (K-M+1)$, as is illustrated in Figure 10.6.

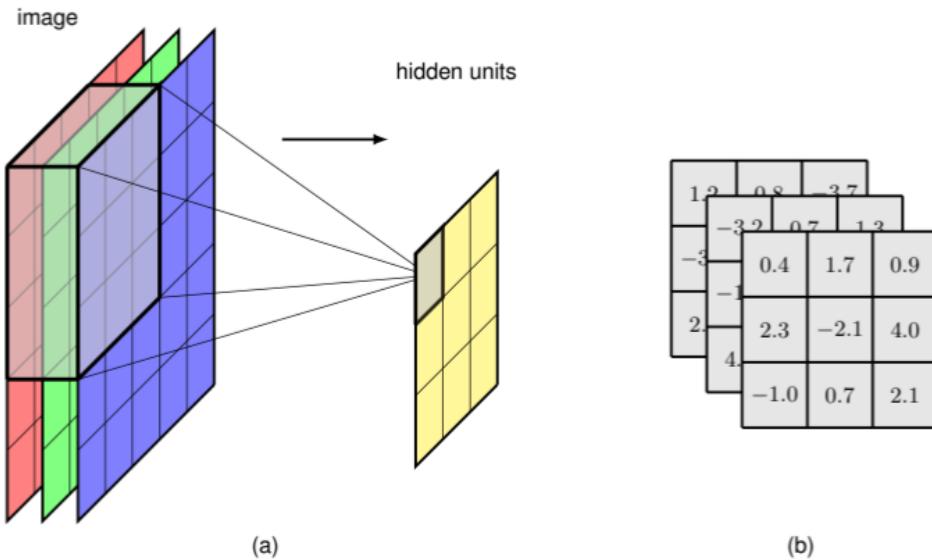


Figure 10.6 (a) Illustration of a multi-dimensional filter that takes input from across the R, G, and B channels. (b) The kernel here has 27 weights (plus a bias parameter not shown) and can be visualized as a $3 \times 3 \times 3$ tensor.

Figure 1: image.png

```

'test' : torch.utils.data.DataLoader(test_data,
                                      batch_size=100,
                                      shuffle=False,
                                      num_workers=1),
}

loaders

{'train': <torch.utils.data.dataloader.DataLoader at 0x1c2a9766150>,
 'test': <torch.utils.data.dataloader.DataLoader at 0x1c289589cd0>}

```

Calculating the feature map dimensions $28 - 5 + 1 + 2 * 2 = 28 / 2 > 14 \quad 14 - 5 + 1 + 2 * 2 = 14/2 = 7$

5 is the filter (kernel) dimensions. A padding of $2 * 2$ adds $2 * 2$ dimensions. A maxpool of (2,2) half the dimensions

```

import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=1,out_channels= 16, kernel_size=5,padding=2)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(in_channels=16,out_channels= 32, kernel_size=5,padding=2)
        self.fc1 = nn.Linear(32*7*7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        #print(x.shape)

        x = x.view(x.size(0), -1)# flatten all dimensions except batch
        #print(x.shape)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

```

```

net = Net()

import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.001)

from torch.autograd import Variable
num_epochs = 10
def train(num_epochs, net, loaders):

    net.train()

    # Train the model
    total_step = len(loaders['train'])

    for epoch in range(num_epochs):
        for i, (images, labels) in enumerate(loaders['train']):

            # gives batch data, normalize x when iterate train_loader
            b_x = Variable(images)      # batch x
            b_y = Variable(labels)      # batch y
            output = net(b_x)
            loss = criterion(output, b_y)

            # clear gradients for this training step
            optimizer.zero_grad()

            # backpropagation, compute gradients
            loss.backward()
            # apply gradients
            optimizer.step()

            if (i+1) % 100 == 0:
                print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                      .format(epoch + 1, num_epochs, i + 1, total_step, loss.item()))
                pass

    pass

```

```
    pass  
train(num_epochs, net, loaders)
```

```
Epoch [1/10], Step [100/600], Loss: 0.3870  
Epoch [1/10], Step [200/600], Loss: 0.1651  
Epoch [1/10], Step [300/600], Loss: 0.1293  
Epoch [1/10], Step [400/600], Loss: 0.0960  
Epoch [1/10], Step [500/600], Loss: 0.1082  
Epoch [1/10], Step [600/600], Loss: 0.0592  
Epoch [2/10], Step [100/600], Loss: 0.0218  
Epoch [2/10], Step [200/600], Loss: 0.0706  
Epoch [2/10], Step [300/600], Loss: 0.1215  
Epoch [2/10], Step [400/600], Loss: 0.0412  
Epoch [2/10], Step [500/600], Loss: 0.0093  
Epoch [2/10], Step [600/600], Loss: 0.0189  
Epoch [3/10], Step [100/600], Loss: 0.0544  
Epoch [3/10], Step [200/600], Loss: 0.1261  
Epoch [3/10], Step [300/600], Loss: 0.0042  
Epoch [3/10], Step [400/600], Loss: 0.0661  
Epoch [3/10], Step [500/600], Loss: 0.0916  
Epoch [3/10], Step [600/600], Loss: 0.0133  
Epoch [4/10], Step [100/600], Loss: 0.0274  
Epoch [4/10], Step [200/600], Loss: 0.0165  
Epoch [4/10], Step [300/600], Loss: 0.0170  
Epoch [4/10], Step [400/600], Loss: 0.0784  
Epoch [4/10], Step [500/600], Loss: 0.0035  
Epoch [4/10], Step [600/600], Loss: 0.0135  
Epoch [5/10], Step [100/600], Loss: 0.0156  
Epoch [5/10], Step [200/600], Loss: 0.0174  
Epoch [5/10], Step [300/600], Loss: 0.0129  
Epoch [5/10], Step [400/600], Loss: 0.0371  
Epoch [5/10], Step [500/600], Loss: 0.0282  
Epoch [5/10], Step [600/600], Loss: 0.0114  
Epoch [6/10], Step [100/600], Loss: 0.0117  
Epoch [6/10], Step [200/600], Loss: 0.0452  
Epoch [6/10], Step [300/600], Loss: 0.0030  
Epoch [6/10], Step [400/600], Loss: 0.0071  
Epoch [6/10], Step [500/600], Loss: 0.0033  
Epoch [6/10], Step [600/600], Loss: 0.0137  
Epoch [7/10], Step [100/600], Loss: 0.0057  
Epoch [7/10], Step [200/600], Loss: 0.0015  
Epoch [7/10], Step [300/600], Loss: 0.0522
```

```
Epoch [7/10], Step [400/600], Loss: 0.0294
Epoch [7/10], Step [500/600], Loss: 0.0021
Epoch [7/10], Step [600/600], Loss: 0.0029
Epoch [8/10], Step [100/600], Loss: 0.0401
Epoch [8/10], Step [200/600], Loss: 0.0223
Epoch [8/10], Step [300/600], Loss: 0.0030
Epoch [8/10], Step [400/600], Loss: 0.0076
Epoch [8/10], Step [500/600], Loss: 0.0013
Epoch [8/10], Step [600/600], Loss: 0.0238
Epoch [9/10], Step [100/600], Loss: 0.0022
Epoch [9/10], Step [200/600], Loss: 0.0245
Epoch [9/10], Step [300/600], Loss: 0.0274
Epoch [9/10], Step [400/600], Loss: 0.0143
Epoch [9/10], Step [500/600], Loss: 0.0037
Epoch [9/10], Step [600/600], Loss: 0.0181
Epoch [10/10], Step [100/600], Loss: 0.0867
Epoch [10/10], Step [200/600], Loss: 0.0259
Epoch [10/10], Step [300/600], Loss: 0.0014
Epoch [10/10], Step [400/600], Loss: 0.0053
Epoch [10/10], Step [500/600], Loss: 0.0015
Epoch [10/10], Step [600/600], Loss: 0.0076
```

```
predicted=[]
def test():
    # Test the model
    net.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in loaders['test']:
            test_output = net(images)
            pred_y = torch.max(test_output, 1)[1].data.squeeze()
            predicted.extend(list(pred_y.numpy()))
            accuracy = (pred_y == labels).sum().item() / float(labels.size(0))
            pass

    print('Test Accuracy of the model on the 10000 test images: %.2f' % accuracy)
    pass

test()
```

```
Test Accuracy of the model on the 10000 test images: 1.00
```

```
results['CNN']=predicted
```

```
results
```

	true_value	Log_Regression	MLP_1	MLP_2	CNN
0	7	7	7	7	7
1	2	2	2	2	2
2	1	1	1	1	1
3	0	0	0	0	0
4	4	4	4	4	4
...
9995	2	2	2	2	2
9996	3	3	3	3	3
9997	4	4	4	4	4
9998	5	5	5	5	5
9999	6	6	6	6	6

Comparison

```
from sklearn.metrics import confusion_matrix,classification_report
performance_metrics= pd.DataFrame()
i=0
accuracy={}
for method in results.columns[1:]:
    print(f"*****{method}*****")
    print(100*"*")
    fig,ax=plt.subplots(1,1,figsize=(10,10))
    confusion_mat= confusion_matrix(results['true_value'],results[method])
    report= classification_report(results['true_value'],results[method])
    report_dic=classification_report(results['true_value'],results[method],output_dict=True)
    accuracy[method]=report_dic["accuracy"]
    print(report)
    ax.set_title(f"{method}")
    sns.heatmap(confusion_mat,annot=True,cmap="viridis")
    fig.tight_layout(rect=[0, 0.03, 1, 0.95])
```

```
plt.show()
print(100*")")
```

```
*****Log_Regression*****
*****
precision    recall   f1-score   support
0            0.95     0.98      0.97      980
1            0.97     0.98      0.97     1135
2            0.93     0.90      0.91     1032
3            0.90     0.91      0.91     1010
4            0.93     0.93      0.93      982
5            0.90     0.87      0.89      892
6            0.94     0.95      0.95      958
7            0.93     0.93      0.93     1028
8            0.87     0.89      0.88      974
9            0.91     0.91      0.91     1009

accuracy                      0.93     10000
macro avg                      0.92     10000
weighted avg                   0.93     10000

*****
*****MLP_1*****
*****
precision    recall   f1-score   support
0            0.98     0.98      0.98      980
1            0.99     0.99      0.99     1135
2            0.96     0.97      0.97     1032
3            0.96     0.96      0.96     1010
4            0.98     0.96      0.97      982
5            0.96     0.96      0.96      892
6            0.98     0.97      0.98      958
7            0.97     0.96      0.97     1028
8            0.94     0.97      0.95      974
9            0.96     0.94      0.95     1009

accuracy                      0.97     10000
macro avg                      0.97     10000
```

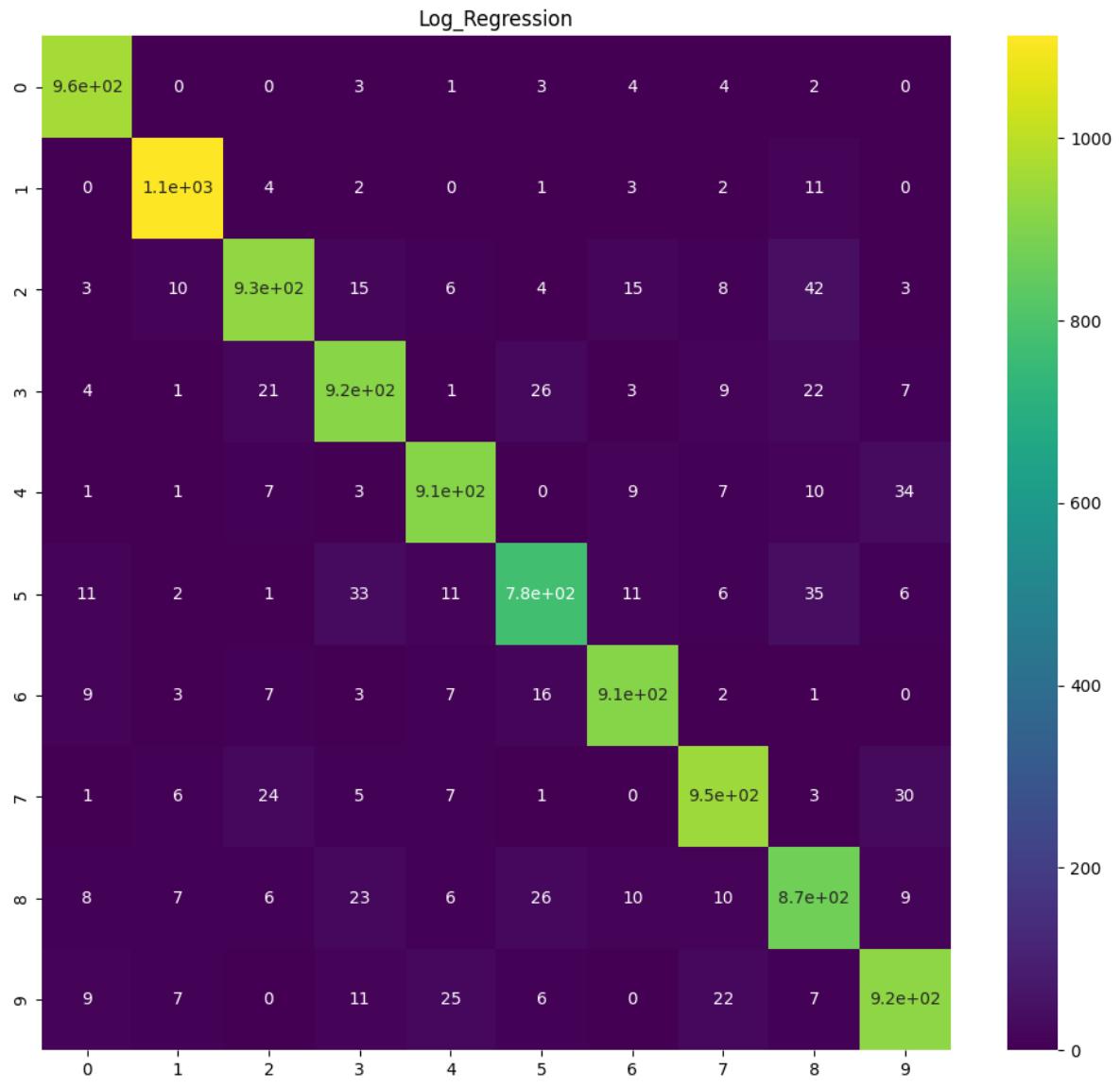
weighted avg	0.97	0.97	0.97	10000
--------------	------	------	------	-------

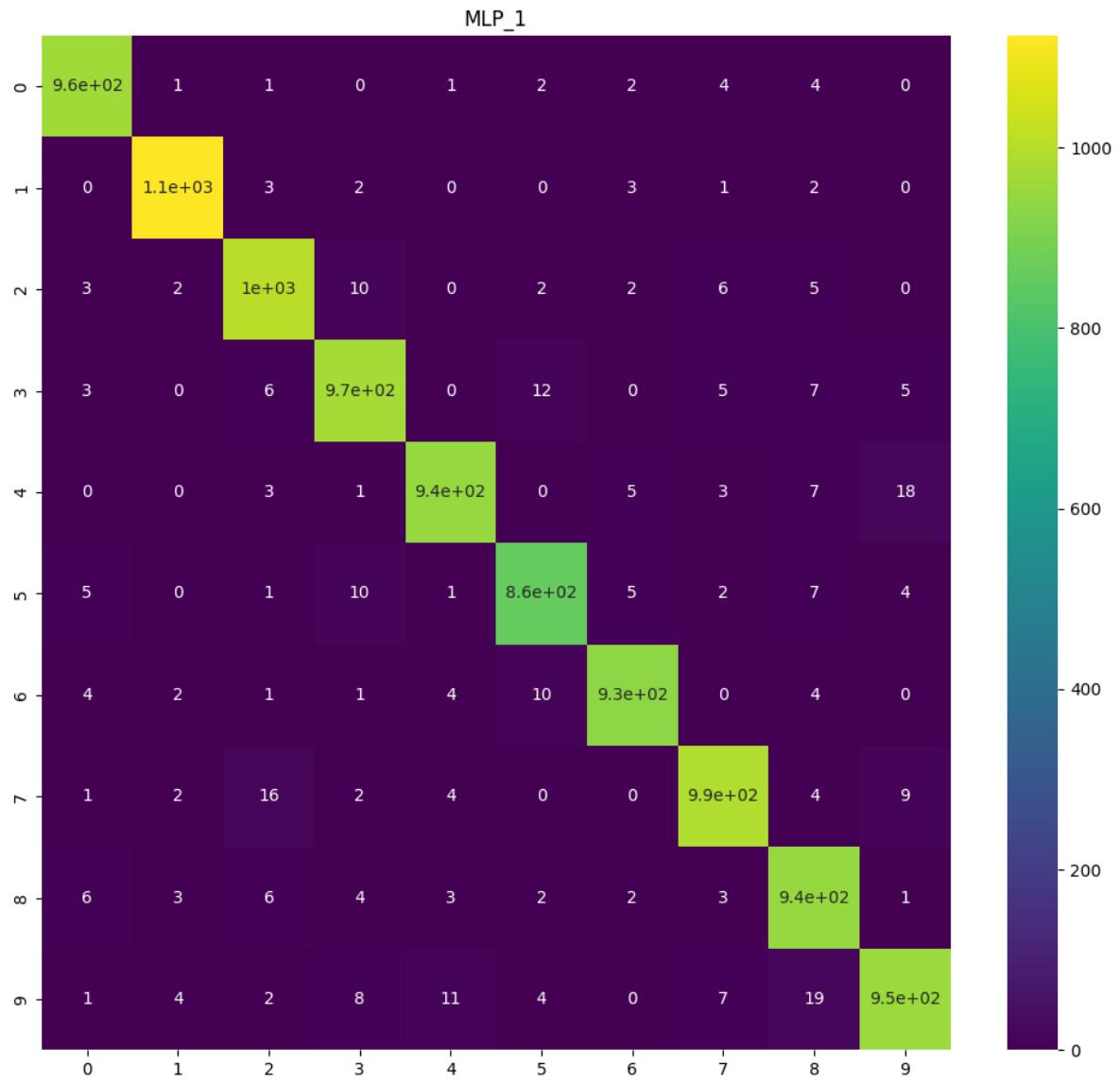
*****MLP_2*****

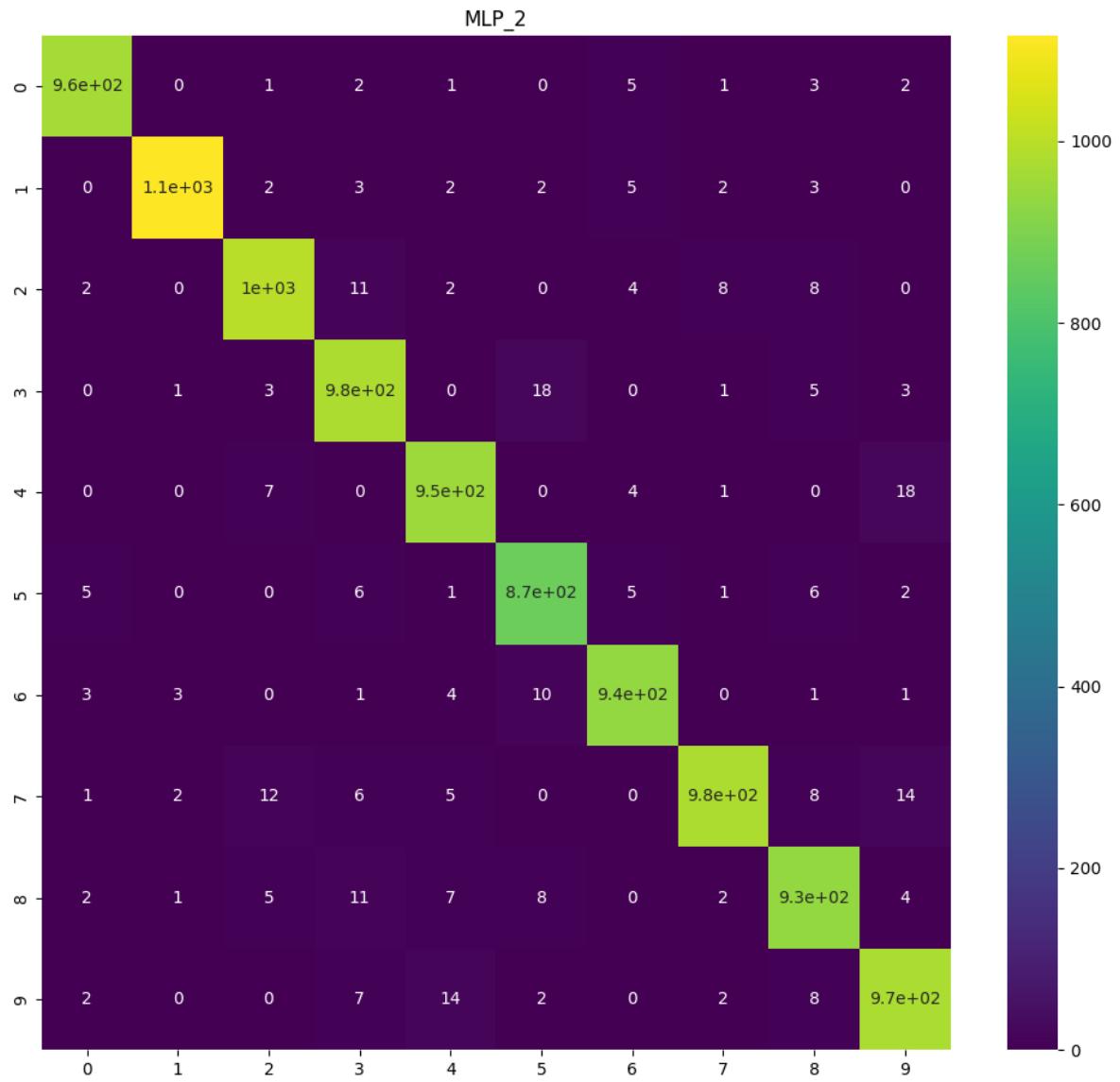
precision	recall	f1-score	support
0	0.98	0.98	0.98
1	0.99	0.98	0.99
2	0.97	0.97	0.97
3	0.95	0.97	0.96
4	0.96	0.97	0.97
5	0.96	0.97	0.96
6	0.98	0.98	0.98
7	0.98	0.95	0.97
8	0.96	0.96	0.96
9	0.96	0.97	0.96
accuracy		0.97	10000
macro avg	0.97	0.97	0.97
weighted avg	0.97	0.97	0.97

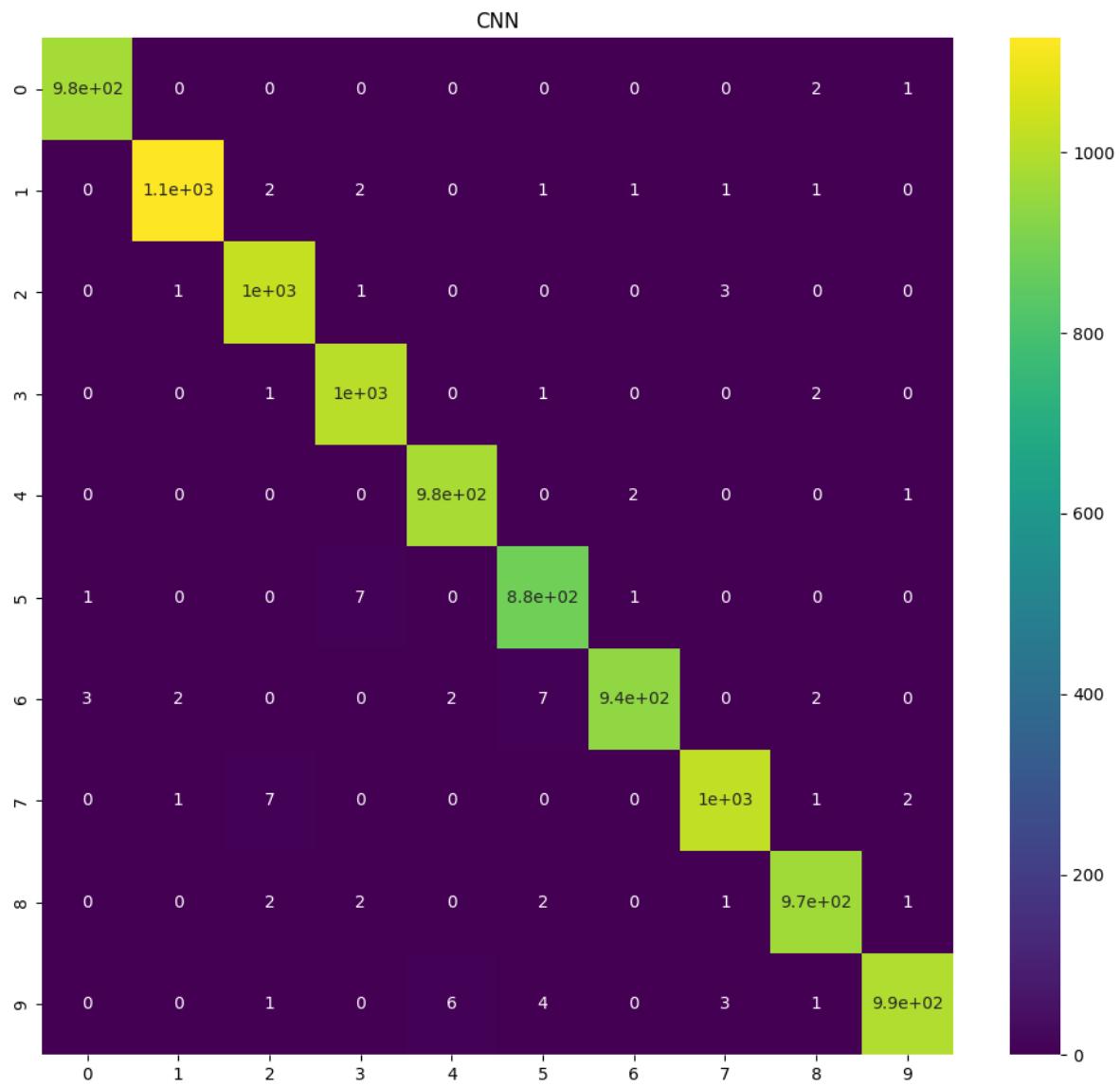
*****CNN*****

precision	recall	f1-score	support
0	1.00	1.00	1.00
1	1.00	0.99	0.99
2	0.99	1.00	0.99
3	0.99	1.00	0.99
4	0.99	1.00	0.99
5	0.98	0.99	0.99
6	1.00	0.98	0.99
7	0.99	0.99	0.99
8	0.99	0.99	0.99
9	0.99	0.99	0.99
accuracy		0.99	10000
macro avg	0.99	0.99	0.99
weighted avg	0.99	0.99	0.99









```
print(pd.DataFrame(data=accuracy,index=range(1)))
```

	Log_Regression	MLP_1	MLP_2	CNN
0	0.9255	0.9684	0.9698	0.9918

References:

Websites:

- 1- https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
- 2- https://scikit-learn.org/stable/modules/neural_networks_supervised.html
- 3- https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html
- 4- fastai
- 5- Elements of statistical learning and introduction to statistical learning

GPT prompts:

Asking to fix the dimensions of feature map of the CNN. Because initially I got the matrix multiplication incompatible dimensions error