```
In [1]: import pandas as pd
import numpy as np
import statsmodels.api as sm
import matplotlib.pyplot as plt
import warnings
import random
import time
from sklearn.neighbors import kneighbors_graph
from sklearn.cluster import KMeans
from scipy.sparse.linalg import eigsh
from sklearn.datasets import fetch_openml
warnings.filterwarnings("ignore")
from sklearn.preprocessing import normalize
from sklearn.metrics import adjusted_rand_score
from sklearn.metrics import normalized_mutual_info_score
```

Data Import

```
In [2]: ### Handwritten Digits data mentioned in the paper
mnistX, mnisty = fetch_openml('mnist_784', version=1, return_X_y=True)
mnistX = mnistX / 255.0
```

```
In [3]: ### Shaped Data Given for Clustering Computational Assignment #2
shaped_data=pd.read_csv('ShapedData.csv',header=None)
```

Plot Function - To show clustering of shaped data

```
In [4]: def plot_clusters(clusters,k,title='Spectral Clustering'):
    plt.figure(figsize=(10, 6))
    cmap=plt.cm.get_cmap('tab10',k)
    for cluster_label in clusters['cluster'].unique():
        cluster_points=clusters[clusters['cluster']==cluster_label]
        plt.scatter(cluster_points[0],cluster_points[1],alpha=0.7,c=cmap(clu
    plt.title(title)
    plt.xlabel('Dimension 1')
    plt.ylabel('Dimension 2')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Part 1. Before moving onto big, real datasets like MNIST, we would like to test the performance on a visually comparable dataset - ShapedData.csv given for comp assignment 2.

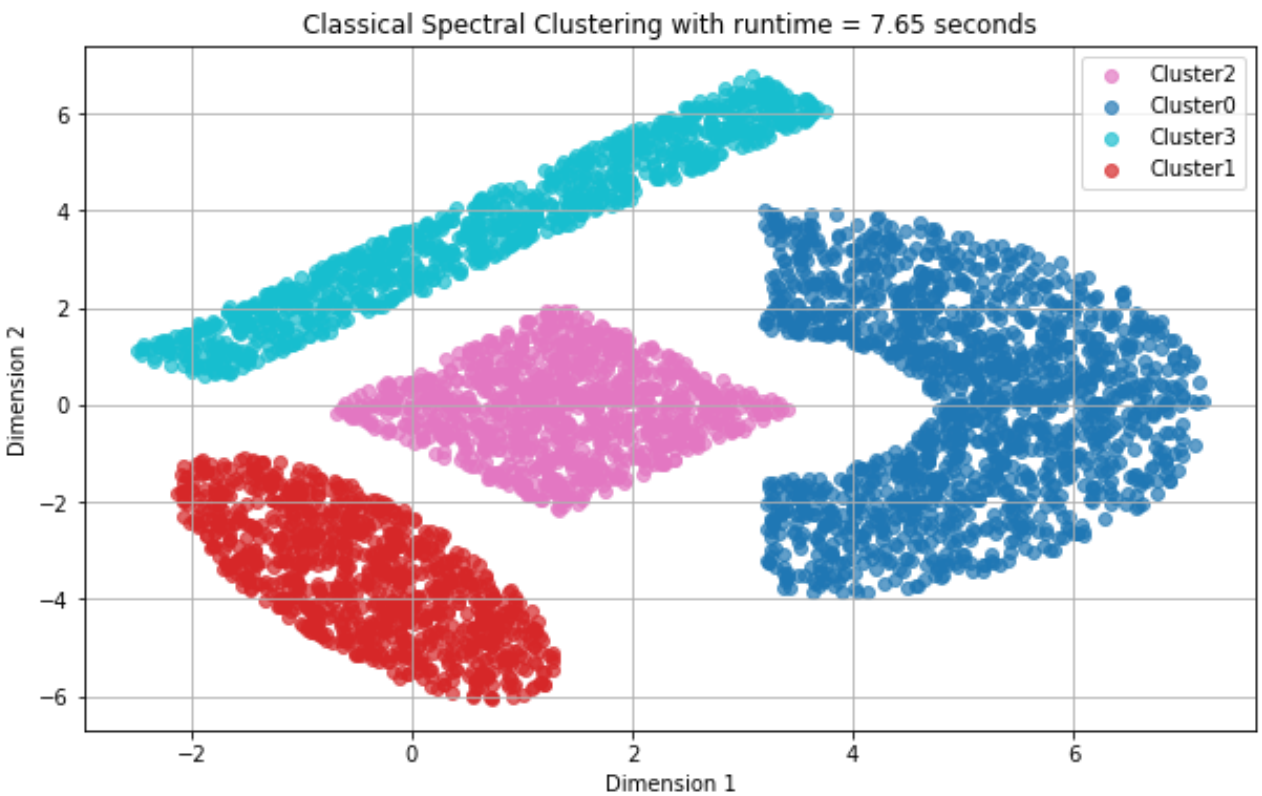
We are doing an extension of the assignment by also testing the performance (running time) & clustering accuracy for the fast and simple method making use of power method

Classical Clustering Algorithm

In [5]:

```
#### Weighted Adjacency Matrix, Diagonal Matrix & Laplacian Matrix

X=np.matrix(shaped_data)

t1=time.time()
#K Nearest Neighbours
K=100
sigma=2
W=np.zeros((len(X),len(X))) #Weighted Matrix
for i in range(len(W)):
    dis=np.linalg.norm(X-X[i], axis=1)
    k_nearest_idx=np.argsort(dis)[1:K+1]
    for j in k_nearest_idx:
        W[i,j]=np.exp(-dis[j]**2 /(2*(sigma**2))) # Setting Gaussian similar
W = 0.5 * (W + W.T)

D=np.sum(W,axis=1)
Dsinv=np.diag(1/np.sqrt(D)) #Inverse of square root
Lnorm=np.eye(len(X))-np.dot(Dsinv,np.dot(W,Dsinv))

eigenvalues, eigenvectors = eigsh(Lnorm, k=4, which='SM')

# Clustering
runs=0
mini=np.inf
while runs<10:
    runs+=1
    km = KMeans(n_clusters=4, random_state=42,init='k-means++')
    C_temp = km.fit_predict(eigenvectors)
    if km.inertia_<runs:
        mini=km.inertia_
        C=C_temp

t2=time.time()
runningtime=t2-t1

#Plotting
shaped_data['cluster']=C
plot_clusters(shaped_data,k=4,title=f'Classical Spectral Clustering with run
```

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should
be avoided as value-mapping will have precedence in case its length matches
with *x* & *y*.  Please use the *color* keyword-argument or provide a 2D arr
ay with a single row if you intend to specify the same RGB or RGBA value for
all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should
be avoided as value-mapping will have precedence in case its length matches
with *x* & *y*.  Please use the *color* keyword-argument or provide a 2D arr
ay with a single row if you intend to specify the same RGB or RGBA value for
all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should
be avoided as value-mapping will have precedence in case its length matches
with *x* & *y*.  Please use the *color* keyword-argument or provide a 2D arr
ay with a single row if you intend to specify the same RGB or RGBA value for
all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should
be avoided as value-mapping will have precedence in case its length matches
with *x* & *y*.  Please use the *color* keyword-argument or provide a 2D arr
ay with a single row if you intend to specify the same RGB or RGBA value for
all points.
```



Fast & Simple Spectral CLustering Using Power Method

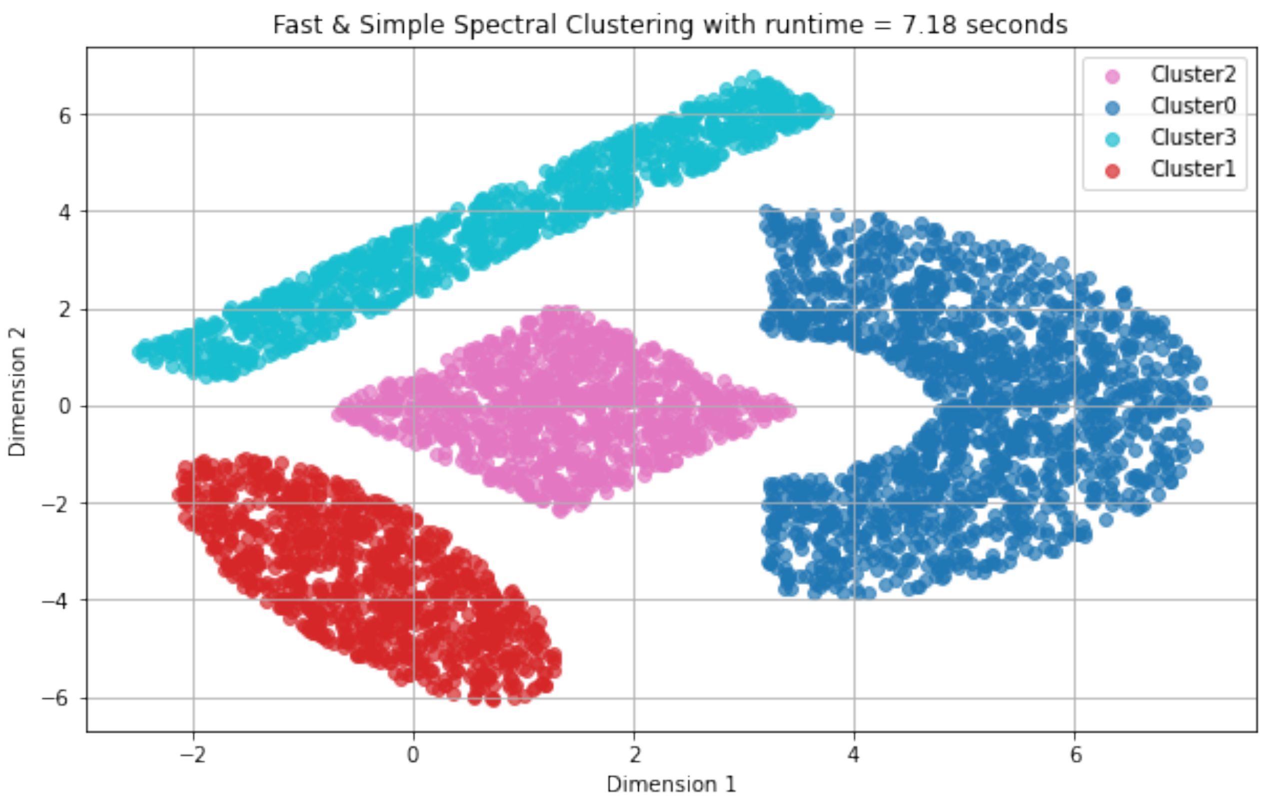
In [6]:
```
### Power Function
def power_method(M, x0, t):
  for _ in range(t):
    x0 = M @ x0
  return x0
```

In [7]:
```
### Only change from previous Laplacian is that we make this a signless lapl
```

```
shaped_data=pd.read_csv('ShapedData.csv',header=None)
X=np.matrix(shaped_data)

t1=time.time()

#K Nearest Neighbours
K=100
sigma=2
k=4
W=np.zeros((len(X),len(X))) #Weighted Matrix
for i in range(len(W)):
    dis=np.linalg.norm(X-X[i], axis=1)
    k_nearest_idx=np.argsort(dis)[1:K+1]
    for j in k_nearest_idx:
        W[i,j]=np.exp(-dis[j]**2 /(2*(sigma**2))) # Setting Gaussian similar
W = 0.5 * (W + W.T)

D=np.sum(W,axis=1)
Dsinv=np.diag(1/np.sqrt(D)) #Inverse of square root
Lnorm=np.eye(len(X))-np.dot(Dsinv,np.dot(W,Dsinv))

M = np.eye(X.shape[0]) - 0.5 * Lnorm #Signless Laplacian


l = int(k)
t = 10*int(np.log(len(M)/ k))
print(l,t)

Y = []
for _ in range(l):
    x0 = np.random.randn(M.shape[0])
    y = power_method(M, x0, t)
    Y.append(y)
Y = np.array(Y).T
Y = normalize(Y, norm='l2')



# Clustering with 10 trials
runs=0
mini=np.inf
while runs<10:
    runs+=1
    km = KMeans(n_clusters=4, random_state=42,init='k-means++')
    C_temp = km.fit_predict(Y)
    if km.inertia_<runs:
        mini=km.inertia_
        C=C_temp

t2=time.time()
runningtime=t2-t1

shaped_data['cluster']=C
plot_clusters(shaped_data,k=4,title=f'Fast & Simple Spectral Clustering with
```

```
4 70
```

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should
be avoided as value-mapping will have precedence in case its length matches
with *x* & *y*.  Please use the *color* keyword-argument or provide a 2D arr
ay with a single row if you intend to specify the same RGB or RGBA value for
all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should
be avoided as value-mapping will have precedence in case its length matches
with *x* & *y*.  Please use the *color* keyword-argument or provide a 2D arr
ay with a single row if you intend to specify the same RGB or RGBA value for
all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should
be avoided as value-mapping will have precedence in case its length matches
with *x* & *y*.  Please use the *color* keyword-argument or provide a 2D arr
ay with a single row if you intend to specify the same RGB or RGBA value for
all points.
*c* argument looks like a single numeric RGB or RGBA sequence, which should
be avoided as value-mapping will have precedence in case its length matches
with *x* & *y*.  Please use the *color* keyword-argument or provide a 2D arr
ay with a single row if you intend to specify the same RGB or RGBA value for
all points.
```



Application on realworld Datasets.

Now that we have seen that the fast and simple clustering method works as good/with close approximation to the classical spectral clustering, our next step is to extend this to the MNIST dataset and compare the running time performance for different lengths of data.

In [8]:

```
### Function for Classical Spectral Clustering

def classical_spectral_clustering(X, k):

  A = kneighbors_graph(X, n_neighbors=10, include_self=False)

  # Calculate Degree Matrix (D)
  degrees = np.asarray(A.sum(axis=1)).flatten()  # d(v) for each vertex
  D = np.diag(degrees)

  # Calculate Normalized Laplacian (N)
  D_inv_sqrt = np.diag(1 / np.sqrt(degrees))
  LNorm = D_inv_sqrt @ (D - A) @ D_inv_sqrt

  print(LNorm.shape)
  eigenvalues, eigenvectors = eigsh(LNorm, k=k, which="SM")

  embedding = eigenvectors
  print(embedding.shape)

  km = KMeans(n_clusters=10, random_state=42,init='k-means++')

  mini=np.inf
  for trials in range(0,10):
        labels = km.fit_predict(embedding)  # Use predict to get cluster ass
        if km.inertia_<mini:
            final_labels=labels
  return final_labels
```

In [9]:
```
### Function for Fast & Simple Clustering

def power_method(M, x0, t):
    for _ in range(t):
        x0 = M @ x0
        x0 /= np.linalg.norm(x0)  # Normalize the vector to prevent overflow
    return x0

def fast_spectral_clustering(X, k, epsilon=0.1, max_iterations=100,d=1):

  n = X.shape[0]  # |V|, number of vertices
  #l = int(np.log(k))  # Number of random vectors
  l = int(np.log(k))
  max_iterations=int(10*np.log2(n/k))  # Number of random vectors
  # Calculate Adjacency Matrix (A) using k-nearest neighbors
  A = kneighbors_graph(X, n_neighbors=10, include_self=False)

  # Calculate Degree Matrix (D)
  degrees = np.asarray(A.sum(axis=1)).flatten()  # d(v) for each vertex
  D = np.diag(degrees)

  # Calculate Normalized Laplacian (N)
  D_inv_sqrt = np.diag(1 / np.sqrt(degrees))
  N = D_inv_sqrt @ (D - A) @ D_inv_sqrt

  # Calculate Signless Laplacian (M)
  M = np.eye(n) - 0.5 * N
  #print(M.shape)
  Y = np.zeros((n, l))
  for i in range(l):
        x0 = np.random.normal(size=(n, 1))
        #print(x0.shape)
        y = power_method(M, x0, max_iterations)
        #print(y.shape)
        Y[:, i] = y.flatten() # Flatten and append
        #print(f"Shape of vector {i}: {y.shape}")

  print(Y.shape)
  Y = normalize(Y, norm='l2')  # Normalize the rows of Y

  #kmeans = KMeans(n_clusters=k, random_state=0).fit(embedding)
  km = KMeans(n_clusters=10, random_state=42,init='k-means++')
  mini=np.inf
  for trials in range(0,20):
        labels = km.fit_predict(Y)  # Use predict to get cluster assignments
        if km.inertia_<mini:
            final_labels=labels
  return final_labels
```

In [ ]:

In [10]:

```
results=[]
for length in [4000,5000,6000,7000,8000,9000,10000,11000,12000]:
    X=mnistX[:length]
    y=mnisty[:length]
    # Classical Spectral Clustering
    start_time = time.time()
    labels_classical = classical_spectral_clustering(X,k=10)
    time_classical = time.time() - start_time

    ari_classical = adjusted_rand_score(y, labels_classical)
    nmi_classical = normalized_mutual_info_score(y, labels_classical)
    #print(f"Classical Spectral Clustering: Time = {time_classical:.2f}s, AR
    results.append({'Clustering Method':'Classical','Length of Data':length,


    # Fast Spectral Clustering
    start_time = time.time()
    labels_fast = fast_spectral_clustering(X, k=10, epsilon=0.1, max_iterati
    time_fast = time.time() - start_time

    ari_fast = adjusted_rand_score(y, labels_fast)
    nmi_fast = normalized_mutual_info_score(y, labels_fast)

    #print(f"Fast Spectral Clustering: Time = {time_fast:.2f}s, ARI = {ari_f
    results.append({'Clustering Method':'Power Methode','Length of Data':len
resultsdf=pd.DataFrame(results)
```

```
(4000, 4000)
(4000, 10)
(4000, 2)
(5000, 5000)
(5000, 10)
(5000, 2)
(6000, 6000)
(6000, 10)
(6000, 2)
(7000, 7000)
(7000, 10)
(7000, 2)
(8000, 8000)
(8000, 10)
(8000, 2)
(9000, 9000)
(9000, 10)
(9000, 2)
(10000, 10000)
(10000, 10)
(10000, 2)
(11000, 11000)
(11000, 10)
(11000, 2)
(12000, 12000)
(12000, 10)
(12000, 2)
```

In [11]:
```python
results2=[]
for length in [4000,5000,6000,7000,8000,9000,10000,11000,12000]:
    X=mnistX[:length]
    y=mnisty[:length]
    # Classical Spectral Clustering
    start_time = time.time()
    labels_classical = classical_spectral_clustering(X,k=np.log(10))
    time_classical = time.time() - start_time

    ari_classical = adjusted_rand_score(y, labels_classical)
    nmi_classical = normalized_mutual_info_score(y, labels_classical)
    #print(f"Classical Spectral Clustering: Time = {time_classical:.2f}s, AR
    results2.append({'Clustering Method':'Classical logK eig','Length of Dat


    # Fast Spectral Clustering
    start_time = time.time()
    labels_fast = fast_spectral_clustering(X, k=10, epsilon=0.1, max_iterati
    time_fast = time.time() - start_time

    ari_fast = adjusted_rand_score(y, labels_fast)
    nmi_fast = normalized_mutual_info_score(y, labels_fast)

    #print(f"Fast Spectral Clustering: Time = {time_fast:.2f}s, ARI = {ari_f
    results2.append({'Clustering Method':'Power Methode logK vectors','Lengt
results2df=pd.DataFrame(results2)
```

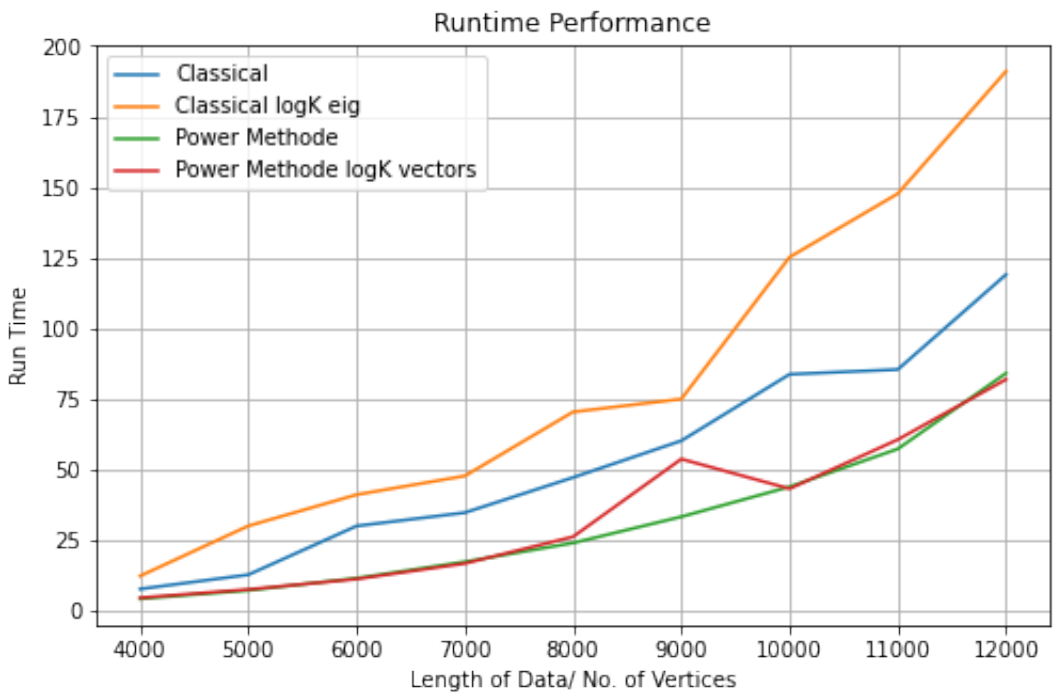
```
(4000, 4000)
(4000, 2)
(4000, 2)
(5000, 5000)
(5000, 2)
(5000, 2)
(6000, 6000)
(6000, 2)
(6000, 2)
(7000, 7000)
(7000, 2)
(7000, 2)
(8000, 8000)
(8000, 2)
(8000, 2)
(9000, 9000)
(9000, 2)
(9000, 2)
(10000, 10000)
(10000, 2)
(10000, 2)
(11000, 11000)
(11000, 2)
(11000, 2)
(12000, 12000)
(12000, 2)
(12000, 2)
```

In [12]:

```
resultsfinal=pd.concat([resultsdf,results2df.reset_index()],axis=0)
resultsfinal.sort_values(by=['Length of Data','Clustering Method'],inplace=T
```

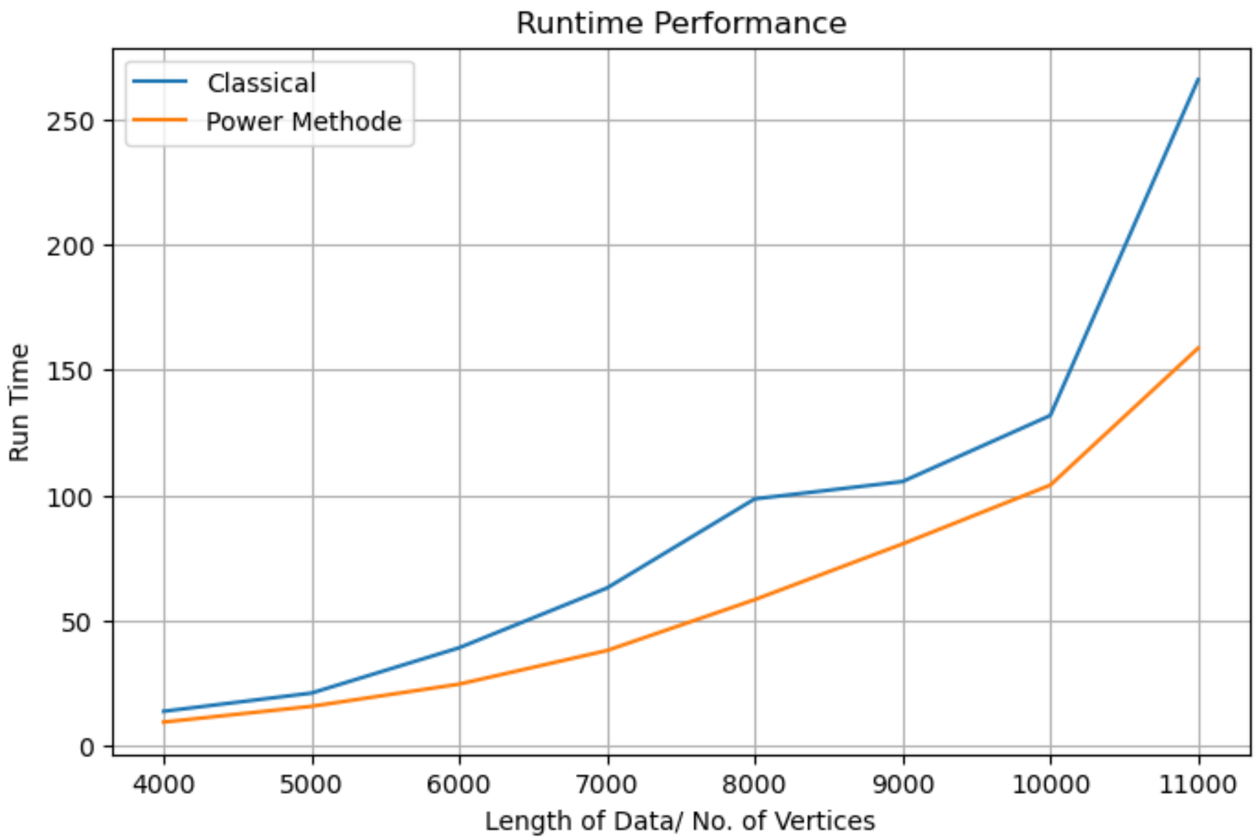
In [13]:

```
plt.figure(figsize=(8,5))
cmap=plt.cm.get_cmap('viridis')
for method in resultsfinal['Clustering Method'].unique():
    cluster_points=resultsfinal[resultsfinal['Clustering Method']==method]
    plt.plot(cluster_points['Length of Data'],cluster_points['Running Time']
plt.title('Runtime Performance')
plt.xlabel('Length of Data/ No. of Vertices')
plt.ylabel('Run Time')
plt.legend()
plt.grid(True)
plt.show()
```



In [806…

```
plt.figure(figsize=(8,5))
cmap=plt.cm.get_cmap('viridis')
for method in resultsdf['Clustering Method'].unique():
    cluster_points=resultsdf[resultsdf['Clustering Method']==method]
    plt.plot(cluster_points['Length of Data'],cluster_points['Running Time']
plt.title('Runtime Performance')
plt.xlabel('Length of Data/ No. of Vertices')
plt.ylabel('Run Time')
plt.legend()
plt.grid(True)
plt.show()
```



In [809…

```
results2=[]
for length in [4000,5000,6000,7000,8000,9000,10000,11000,12000]:
    X=mnistX[:length]
    y=mnisty[:length]
    # Classical Spectral Clustering
    start_time = time.time()
    labels_classical = classical_spectral_clustering(X,k=np.log(10))
    time_classical = time.time() - start_time

    ari_classical = adjusted_rand_score(y, labels_classical)
    nmi_classical = normalized_mutual_info_score(y, labels_classical)
    print(f"Classical Spectral Clustering: Time = {time_classical:.2f}s, ARI
    results2.append({'Clustering Method':'Classical logK eig','Length of Dat


    # Fast Spectral Clustering
    start_time = time.time()
    labels_fast = fast_spectral_clustering(X, k=10, epsilon=0.1, max_iterati
    time_fast = time.time() - start_time

    ari_fast = adjusted_rand_score(y, labels_fast)
    nmi_fast = normalized_mutual_info_score(y, labels_fast)

    print(f"Fast Spectral Clustering: Time = {time_fast:.2f}s, ARI = {ari_fa
    results2.append({'Clustering Method':'Power Methode logK vectors','Lengt
```

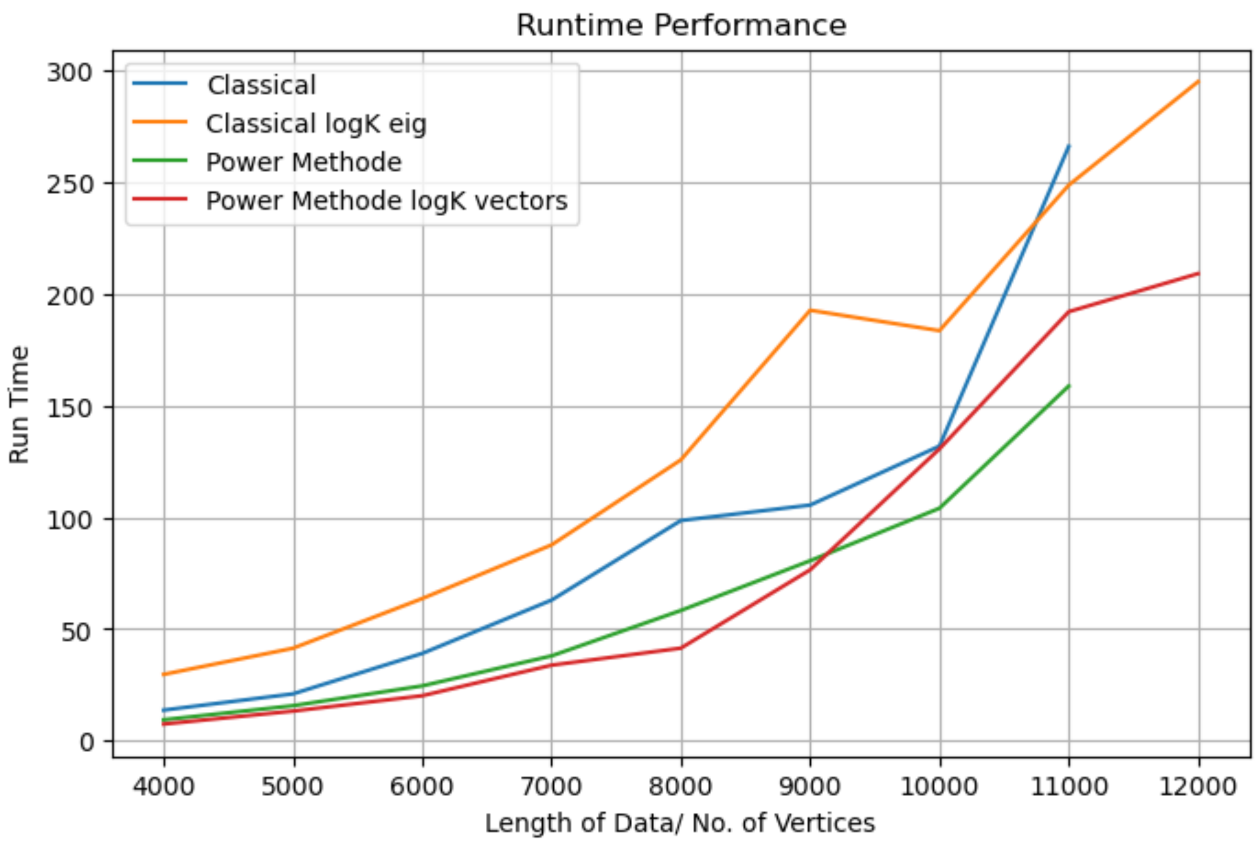
```
(4000, 4000)
(4000, 2)
Classical Spectral Clustering: Time = 29.69s, ARI = 0.2322, NMI = 0.4085
(4000, 2)
Fast Spectral Clustering: Time = 7.47s, ARI = 0.3026, NMI = 0.4401
(5000, 5000)
(5000, 2)
Classical Spectral Clustering: Time = 41.40s, ARI = 0.2407, NMI = 0.4278
(5000, 2)
Fast Spectral Clustering: Time = 13.27s, ARI = 0.2798, NMI = 0.4266
(6000, 6000)
(6000, 2)
Classical Spectral Clustering: Time = 63.63s, ARI = 0.2475, NMI = 0.4152
(6000, 2)
Fast Spectral Clustering: Time = 20.16s, ARI = 0.3068, NMI = 0.4275
(7000, 7000)
(7000, 2)
Classical Spectral Clustering: Time = 87.62s, ARI = 0.1876, NMI = 0.3833
(7000, 2)
Fast Spectral Clustering: Time = 33.82s, ARI = 0.2484, NMI = 0.3970
(8000, 8000)
(8000, 2)
Classical Spectral Clustering: Time = 125.66s, ARI = 0.3347, NMI = 0.5064
(8000, 2)
Fast Spectral Clustering: Time = 41.44s, ARI = 0.2495, NMI = 0.4020
(9000, 9000)
(9000, 2)
Classical Spectral Clustering: Time = 192.60s, ARI = 0.2715, NMI = 0.4355
(9000, 2)
Fast Spectral Clustering: Time = 76.49s, ARI = 0.3801, NMI = 0.4962
(10000, 10000)
(10000, 2)
Classical Spectral Clustering: Time = 183.47s, ARI = 0.1980, NMI = 0.3584
(10000, 2)
Fast Spectral Clustering: Time = 130.59s, ARI = 0.2452, NMI = 0.3995
(11000, 11000)
(11000, 2)
Classical Spectral Clustering: Time = 248.57s, ARI = 0.1954, NMI = 0.3579
(11000, 2)
Fast Spectral Clustering: Time = 191.96s, ARI = 0.3444, NMI = 0.4965
(12000, 12000)
(12000, 2)
Classical Spectral Clustering: Time = 294.85s, ARI = 0.1051, NMI = 0.2747
(12000, 2)
Fast Spectral Clustering: Time = 208.98s, ARI = 0.3967, NMI = 0.5088
```

In [821…

```
results2df=pd.DataFrame(results2)
resultsfinal=pd.concat([resultsdf2,results2df.reset_index()],axis=0)
resultsfinal.sort_values(by=['Length of Data','Clustering Method'],inplace=T
```

In [822…

```
plt.figure(figsize=(8,5))
cmap=plt.cm.get_cmap('viridis')
for method in resultsfinal['Clustering Method'].unique():
    cluster_points=resultsfinal[resultsfinal['Clustering Method']==method]
    plt.plot(cluster_points['Length of Data'],cluster_points['Running Time']
plt.title('Runtime Performance')
plt.xlabel('Length of Data/ No. of Vertices')
plt.ylabel('Run Time')
plt.legend()
plt.grid(True)
plt.show()
```



In [18]:

```
resultsfinal.drop(columns='index').reset_index(drop=True,inplace=True)
resultsfinal.groupby('Clustering Method').agg(Run_Time=('Running Time','mean
```

Out[18]:

	Clustering Method	Run_Time	Average_ARI	Average_NMI
0	Classical	53.396813	0.391639	0.569865
1	Classical logK eig	82.245914	0.251837	0.431163
2	Power Methode	31.411882	0.335207	0.459641
3	Power Methode logK vectors	33.954710	0.302774	0.431422