ESE_2240_Lab_13

April 28, 2025

1 Lab 13: Signal Processing on Graphs - Classification of Cancer Types

[10]:
```
import numpy as np
import scipy as scp
import matplotlib as mpl
import matplotlib.pylab as pl
import networkx as nx
import matplotlib.pyplot as plt
import scipy.spatial.distance as ssd
from scipy.spatial.distance import pdist, squareform
from scipy.stats import mode
import scipy.spatial.distance as ssd
import statistics
```

[11]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

2 2: Genetic network

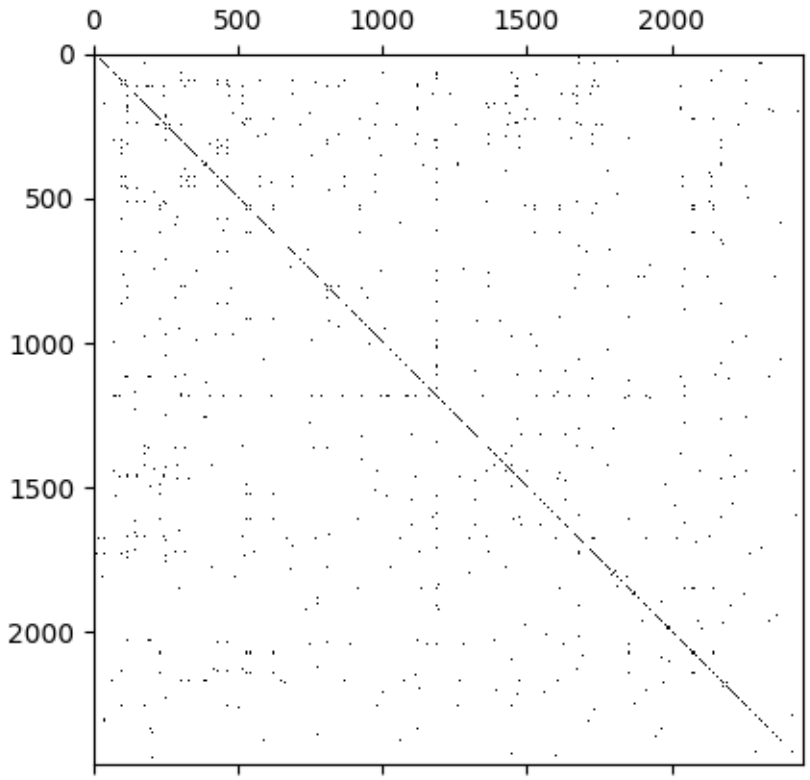
2.1 2.1: Understanding the data

[12]:
```
geneNetwork = scp.io.loadmat('/content/drive/My Drive/geneNetwork_rawPCNCI.mat')
```

[13]:
```
A = geneNetwork['geneNetwork_rawPCNCI']
pl.spy(A) # Displays the adjacency matrix
print(A)
```

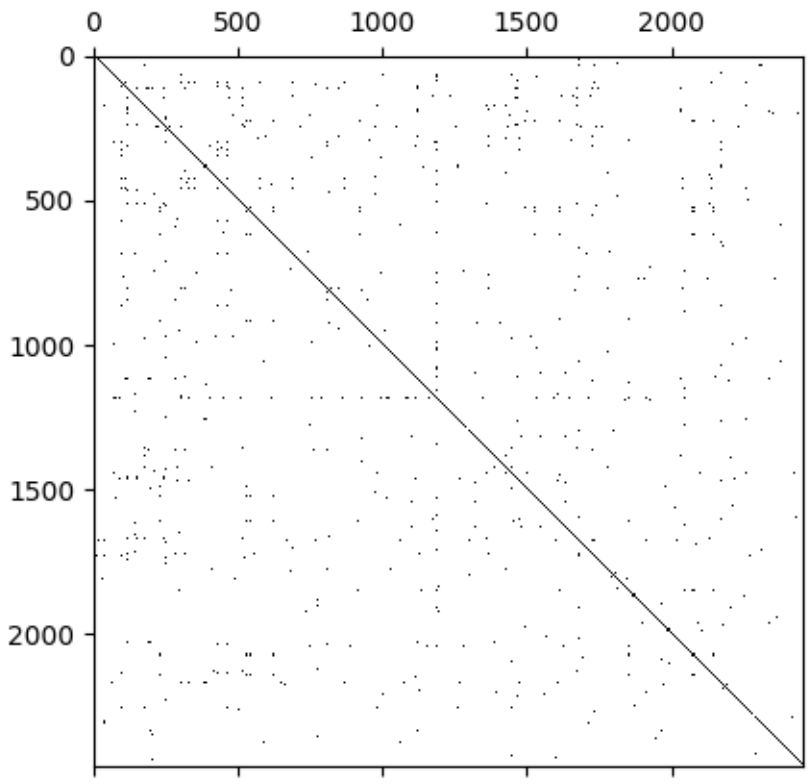
```
[[1 0 0 … 0 0 0]
 [0 0 0 … 0 0 0]
 [0 0 0 … 0 0 0]
 …
 [0 0 0 … 0 0 0]
```

```
[0 0 0 … 0 0 0]
[0 0 0 … 0 0 0]]
```



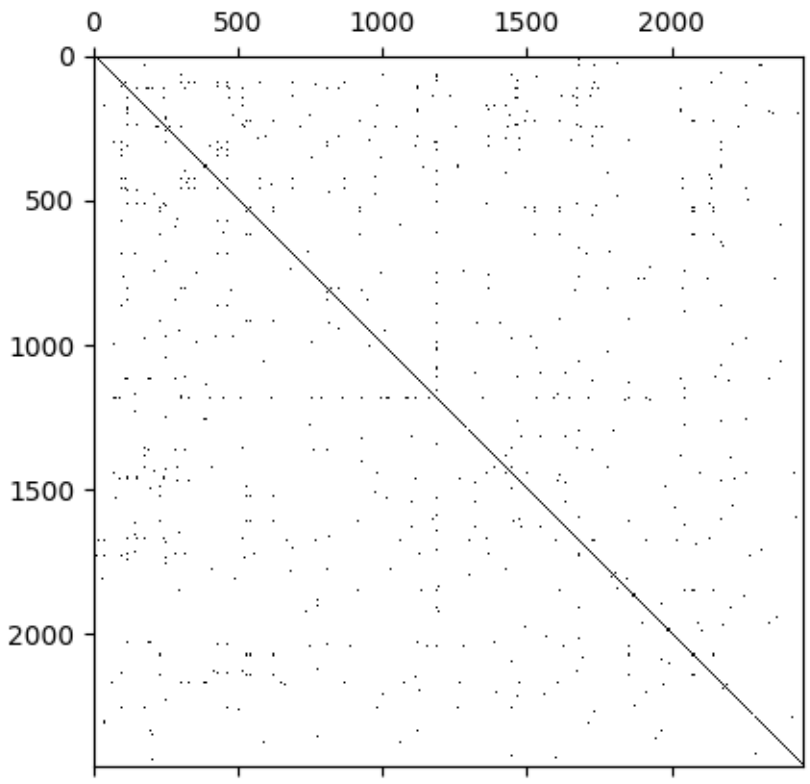
[14]:
```
G = nx.from_numpy_array(A) # Converts to graph format
D = np.diag(np.sum(A, axis=1))   # Degree matrix
L = D-A # TODO: Calculate Laplacian of the gene network graph
pl.spy(L)
```

[14]: <matplotlib.image.AxesImage at 0x7932fba72b90>



[15]:
```
A_hat = A.copy() # TODO: remove all self-loops from the graph
np.fill_diagonal(A_hat, 0)
G_hat = nx.from_numpy_array(A_hat) # TODO: convert A_hat to graph
D_hat = np.diag(np.sum(A_hat, axis=1))
L_hat = D_hat - A_hat# TODO: calculate Laplacians
pl.spy(L)
```

[15]: <matplotlib.image.AxesImage at 0x79330a1dd690>



Comment: 1- Yes the graph contains self loops since there places that had no points (meaning their was an edge not equal to zero between the node and itself) 2- The graph is undirected as it is symmetric around the diagonal of it (0-0 1-1 2-2 etc..) 3- The graph is unweighted since all edges are either 1 or 0 4- No,when we make the diagonals of A zero it doesn't matter because when we subtract D-A the diagonal values of A cancels itself in D resulting in the same L so L and L_hat are the same thing.

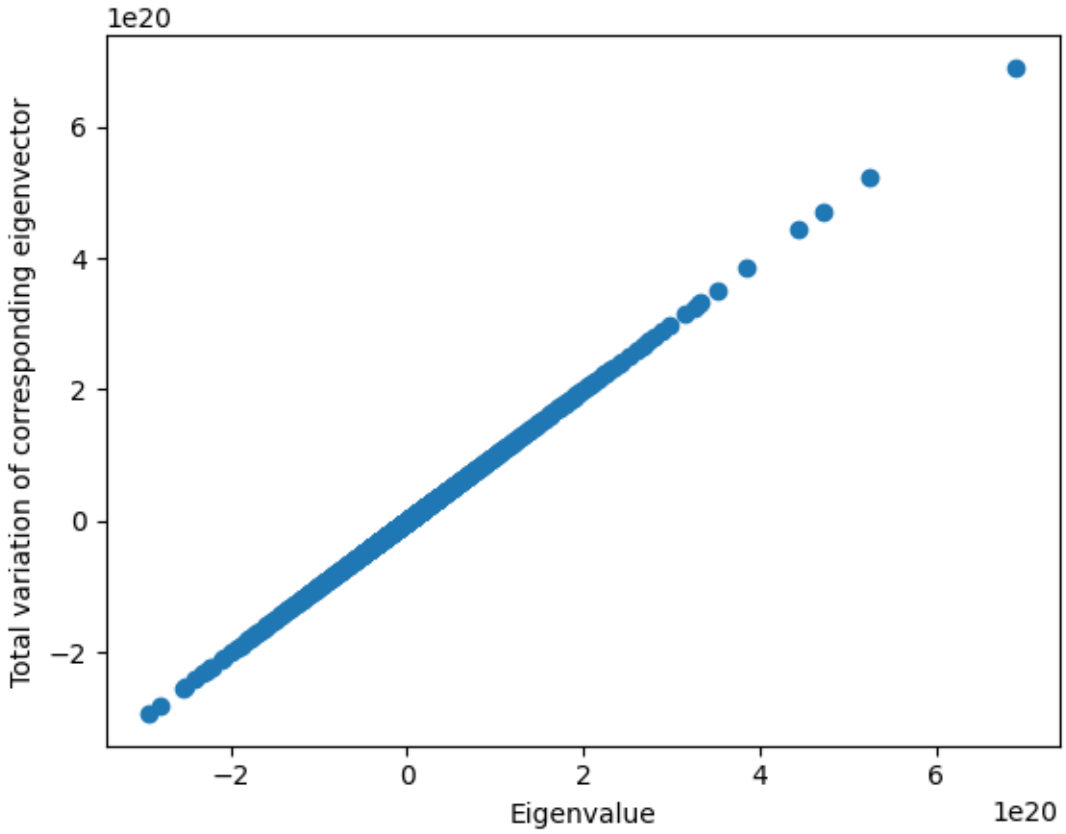
2.2 2.2: Total variations

[16]:
```
L_eigenValues, L_eigenVectors =  np.linalg.eigh(L) # TODO: Calculate␣
↪eigenvectors of L
```

[17]:
```
TV = []
for k in range(L_eigenVectors.shape[1]):
    v_k = L_eigenVectors[:, k]
    tv_k = v_k.T @ L @ v_k  # total variation formula
    TV.append(tv_k)

TV =  np.array(TV)# TODO: Calculate total variation for each eigenvector (feel␣
↪free to use a for loop)
```

[18]:
```
# Plot total variation
plt.scatter(L_eigenValues, TV)
plt.xlabel("Eigenvalue")
plt.ylabel("Total variation of corresponding eigenvector")
```

[18]:
```
Text(0, 0.5, 'Total variation of corresponding eigenvector')
```



Eigenvectors associated with larger eigenvalues oscillate faster.

3 3: Genetic profiles

[19]:
```
signal_mutation = scp.io.loadmat('/content/drive/My Drive/signal_mutation.mat')
X_sm = signal_mutation['signal_mutation']
X_sm_T = X_sm.T # Ensure that each column denotes a different person, not row

histology_subtype = scp.io.loadmat('/content/drive/My Drive/histology_subtype.
↪mat')
y_hs = histology_subtype['histology_subtype']
y_hs = y_hs.flatten()
```

3.1 3.1: Distinguishing power

[20]:
```
def get_GFT(X, V):
  return np.conj(V.T) @ X
  # TODO: calculate GFT of inputs X using shift operator V
```

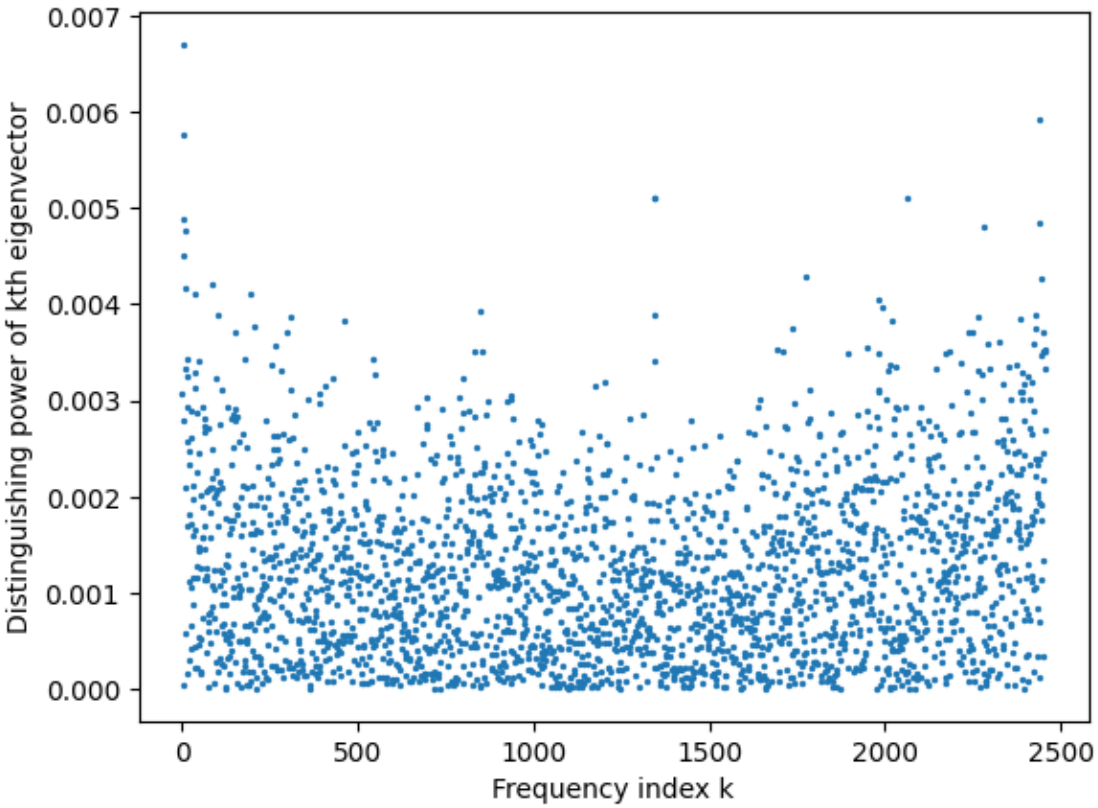
[21]:
```
X_sm_GFT_T = get_GFT(X_sm_T, L_eigenVectors) # TODO: calculate GFT of inputs in␣
 ↪X_sm_T using L_eigenVectors
```

[22]:
```
# Get distinguishing power of the kth eigenvector
# Input: the index of the eigenvector k, the GFTs of all patients X, the actual␣
 ↪histology vector (labels) y
# Output: distinguishing power of kth eigenvector
def get_dp(k, X, y):
    serous_idx = (y == 1)
    endometrioid_idx = (y == 2)

    serous_mean = np.mean(X[k, serous_idx])
    endometrioid_mean = np.mean(X[k, endometrioid_idx])

    numerator = serous_mean - endometrioid_mean
    denominator = np.sum(np.abs(X[k, :]))

    return np.abs(numerator / denominator)
  # TODO: Calculate distinguishing power
```

[23]:
```
dps = np.array([get_dp(k, X_sm_GFT_T, y_hs) for k in range(X_sm_GFT_T.
 ↪shape[0])])# TODO: get distinguishing power for each eigenvector
```

[24]:
```
plt.scatter(range(X_sm_GFT_T.shape[0]), dps, s = 2)
plt.xlabel("Frequency index k")
plt.ylabel("Distinguishing power of kth eigenvector")
```

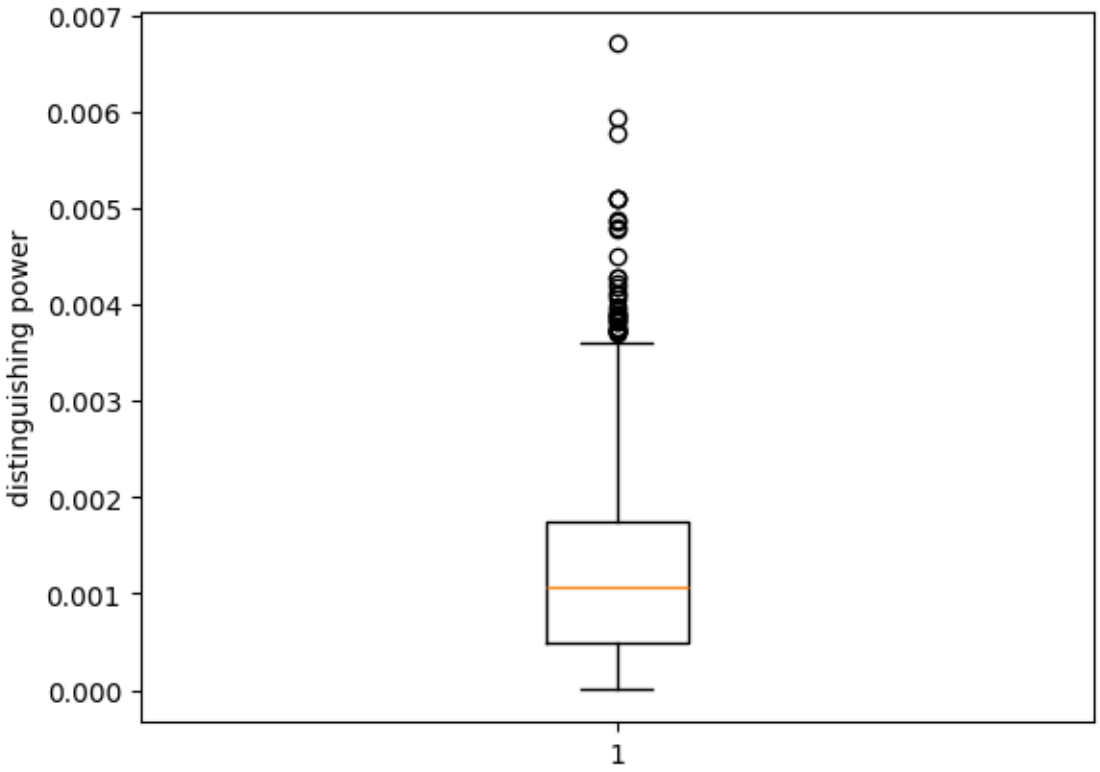
[24]:
```
Text(0, 0.5, 'Distinguishing power of kth eigenvector')
```



3.2 3.2: Interpretation

[25]:
```
plt.boxplot(dps)
plt.ylabel("distinguishing power")
```

[25]:
```
Text(0, 0.5, 'distinguishing power')
```



Comments: The boxplot of DP(v_k) shows that the majority of oscillation modes have very low distinguishing power, clustered near zero. This indicates that most modes do not significantly differentiate between the serous and endometrioid subtypes. However, there are a few modes with much higher distinguishing power, appearing as outliers in the boxplot. These modes contain important distinguishing features of the two subtypes.

A good thing to use low pass filter.

[26]:
```
# Optional: estimate outliers in DP
```

4 4: Improving classifications using graph filter

4.1 4.1: k-NN for leave-one-out cross validation

[27]:
```
# Predict accuracy of a KNN classifier using leave one out cross validation
# Inputs: number of neighbors to use k, dataset X, labels y
def knn_predict_accuracy(k, X, y):
    d = ssd.squareform(ssd.pdist(X.T, 'euclidean')) # pairwise distances␣
 ↪between all patients
    nn = d.argsort(axis=0) # sort distances to find nearest neighbors
    nn_labels = y[nn[1:(k+1), :]] # get labels of k nearest neighbors (skip␣
 ↪first index which is the patient itself)
```

```
    # Getting most common label (mode) among k nearest neighbors
    predictions = np.zeros(len(y))
    for i in range(len(y)):
        predictions[i] = statistics.mode(nn_labels[:, i])

    return np.mean(predictions == y)
```

[28]:
```
# Test it a bit for k = [3,5,7]
# transposed genetic mutation data: X_sm_T
# cancer subtype labels: y_hs
knn_predict_accuracy(4, X_sm_T, y_hs)
```

[28]:
```
np.float64(0.8541666666666666)
```

This means that for each patient in the dataset, look at 4 closest genetic profiel matches, predict cancer subtype based on these, and 85% prediction were correct.

[30]:
```
# Test with different k values
k_values = [3, 5, 7]
for k in k_values:
    accuracy = knn_predict_accuracy(k, X_sm_T, y_hs)
    print(f"Original data with k={k}: Accuracy = {accuracy:.4f}")
```

```
Original data with k=3: Accuracy = 0.8667
Original data with k=5: Accuracy = 0.8708
Original data with k=7: Accuracy = 0.8583
```

Accuracy initially increases from k=3 to k=5 but then decreases for k=7, suggesting a potential overfitting when using too many neighbors.

4.2 4.2: Graph filters

[31]:
```
# Project onto the top k eigenvectors in distinguishing power
# Note that the k for choosing topk here is different from that used in KNN
# Inputs: how many vectors to use k, distinguishing power list dp, datapoints␣
 ↪to project X_sm_T,
#         eigenvectors V
# Outputs: X_sm_T, projected onto the top k eigenvectors
def filter_topk_dp(k, dp, X_sm_T, V):
    top_k_indices = np.argsort(dp)[-k:] # indices of top k
    X_gft = get_GFT(X_sm_T, V) # GFT of input data

    # Zero out coefficients except for top k
    X_filtered_gft = np.zeros_like(X_gft)
    X_filtered_gft[top_k_indices, :] = X_gft[top_k_indices, :]

    # Transform back to original domain
```

```
    return V @ X_filtered_gft
```

[32]:
```
# Filter to the top p percentage DP
# Like the above, but instead of taking the top k, it takes the top p␣
 ↪percentage of eigenvectors
def filter_topp_dp(p, dp, X_sm_T, V):
    threshold = np.percentile(dp, 100-p*100) # threshold for top p percentile
    top_indices = np.where(dp >= threshold)[0] # indices above threshold
    X_gft = get_GFT(X_sm_T, V)

    # Zero out coefficients except for those above threshold
    X_filtered_gft = np.zeros_like(X_gft)
    X_filtered_gft[top_indices, :] = X_gft[top_indices, :]

    # Transform back to original domain
    return V @ X_filtered_gft
```

[35]:
```
# Function to help you with printing accuracies
# Inputs: amount of neighbors to use for knn knn_amt, filter_param is either k␣
 ↪or p depending on the filter specified,
#         filter = 'k' or 'p' depending on the kind of filtering
def get_and_print_accuracies(knn_amt, filter_param, filter='k'):
  if filter == 'k':
    filterer = filter_topk_dp
  elif filter == 'p':
    filterer = filter_topp_dp
  else:
    print("only k or p type filters supported")
    return
  for i in knn_amt:
    filtered = filterer(filter_param, dps, X_sm_T, L_eigenVectors) # TODO: Use␣
 ↪filterer to get appropriate projection of X_sm_T
    acc = knn_predict_accuracy(i, filtered, y_hs) # TODO: Predict leave one out␣
 ↪accuracy using filtered as the dataset
    print(str(filter_param) + ", filter = " + filter + ", k = " + str(i) + ",␣
 ↪accuracy: " + str(acc))
```

[36]:
```
k_filters = [3, 5, 7] # feel free to test larger values too
for j in k_filters:
  get_and_print_accuracies([3,5,7], j, filter = 'k') # note the [3,5,7] here␣
 ↪refer to number of knn neighbors
  print("-----")
```

```
3, filter = k, k = 3, accuracy: 0.8708333333333333
3, filter = k, k = 5, accuracy: 0.8791666666666667
3, filter = k, k = 7, accuracy: 0.8833333333333333
-----
```

```
5, filter = k, k = 3, accuracy: 0.8708333333333333
5, filter = k, k = 5, accuracy: 0.8833333333333333
5, filter = k, k = 7, accuracy: 0.8916666666666667
-----
7, filter = k, k = 3, accuracy: 0.8666666666666667
7, filter = k, k = 5, accuracy: 0.8666666666666667
7, filter = k, k = 7, accuracy: 0.875
-----
```

The first number is how many top eigenvectors we kept in the filter.

The second number is how many neighbours were used in the kNN classification.

Recall these were the original accuracies: * Original data with k=3: Accuracy = 0.8667 * Original data with k=5: Accuracy = 0.8708 * Original data with k=7: Accuracy = 0.8583

For each k value, accuracy improved with filtering for all filtered cases, although it was just a small improvement. By keeping only the most discriminative eigenvectors (graph frequencies), we're achieving better classification performance.

Notably, filtering only the 5 key eigenvectors achieved the best performance across the board. This suggests that additional eigenvectors when we take 7 capture noisy or irrelevant frequencies instead of signal.

[37]:
```
# Now let's test out percentage based filtering.
p = [0.75, 0.8, 0.85, 0.9, 0.95] # feel free to test other values too
for j in p:
  get_and_print_accuracies([3,5,7], j, filter = 'p')
  print("-----")
```

```
0.75, filter = p, k = 3, accuracy: 0.9125
0.75, filter = p, k = 5, accuracy: 0.8875
0.75, filter = p, k = 7, accuracy: 0.8958333333333334
-----
0.8, filter = p, k = 3, accuracy: 0.8958333333333334
0.8, filter = p, k = 5, accuracy: 0.8833333333333333
0.8, filter = p, k = 7, accuracy: 0.8875
-----
0.85, filter = p, k = 3, accuracy: 0.9
0.85, filter = p, k = 5, accuracy: 0.8791666666666667
0.85, filter = p, k = 7, accuracy: 0.8708333333333333
-----
0.9, filter = p, k = 3, accuracy: 0.8958333333333334
0.9, filter = p, k = 5, accuracy: 0.875
0.9, filter = p, k = 7, accuracy: 0.8666666666666667
-----
0.95, filter = p, k = 3, accuracy: 0.9
0.95, filter = p, k = 5, accuracy: 0.8708333333333333
0.95, filter = p, k = 7, accuracy: 0.8583333333333333
-----
```

The percentage based filtering does achieve better results, with k=3 topping 90% accuracy. The 75% filter has the highest accuracy across all k values.