Sheet 1

In [17]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import pyplot as plt
import matplotlib.patches as mpatches
%matplotlib inline
```

1 Principal Component Analysis

(a)

In [18]:
```
# TODO: implement PCA (fill in the blanks in the function below)

def pca(data, n_components=None):
    """
    Principal Component Analysis on a p x N data matrix.

    Parameters
    ----------
    data : np.ndarray
        Data matrix of shape (p, N).
    n_components : int, optional
        Number of requested components. By default returns all components.

    Returns
    -------
    np.ndarray, np.ndarray
        the pca components (shape (n_components, p)) and the projection (shape (n_components, N))

    """
    # set n_components to p by default
    n_components = data.shape[0] if n_components is None else n_components
    assert n_components <= data.shape[0], f"Got n_components larger than dimensionality of data!"
```

```
    # center the data

    n = data.shape[1]
    X = data - 1/n * data @ np.ones((n,n))

    # compute X times X transpose

    XX = X @ X.T

    # compute the eigenvectors and eigenvalues

    w, v = np.linalg.eig(XX)

    # sort the eigenvectors by eigenvalue and take the n_components largest ones

    w, v = zip(*sorted(zip(w,v.T), reverse=True))
    components = np.array(v[:n_components])

    # compute X_projected, the projection of the data to the components

    X_projected = X.T @ components.T

    return components, X_projected.T  # return the n_components first components and the pca projection of the data
```

Good

In [19]:
```
# Example data to test your implementation
# All the asserts on the bottom should go through if your implementation is correct

data = np.array([
    [ 1,  0,  0, -1,  0,  0],
    [ 0,  3,  0,  0, -3,  0],
    [ 0,  0,  5,  0,  0, -5]
], dtype=np.float32)


# add a random offset to all samples. it should not affect the results
data += np.random.randn(data.shape[0], 1)

n_components = 2
components, projection = pca(data, n_components=n_components)  # apply your implementation
```

```
# the correct results are known (up to some signs)
true_components = np.array([[0, 0, 1], [0, 1, 0]], dtype=np.float32)
true_projection = np.array([
    [ 0,  0,  5,  0,  0, -5],
    [ 0,  3,  0,  0, -3,  0]
], dtype=np.float32)

# check that components match, up to sign
assert isinstance(components, np.ndarray), f'Expected components to be numpy array but got {type(components)}'
assert components.shape == true_components.shape, f'{components.shape}!={true_components.shape}'
assert np.allclose(np.abs(components * true_components).sum(1), np.ones(n_components)), f'Components not matching'

# check that projections agree, taking into account potentially flipped components
assert isinstance(projection, np.ndarray), f'Expected projection to be numpy array but got {type(projection)}'
assert projection.shape == (n_components, data.shape[1]), f'Incorrect shape of projection: Expected {(n_components, data.shape
assert np.allclose(projection, true_projection * (components * true_components).sum(1, keepdims=True), atol=1e-6), f'Projectic

print('Test successful!')
```

```
Test successful!
```

(b)

Load the data (it is a subset of the data at https://opendata.cern.ch/record/4910#)

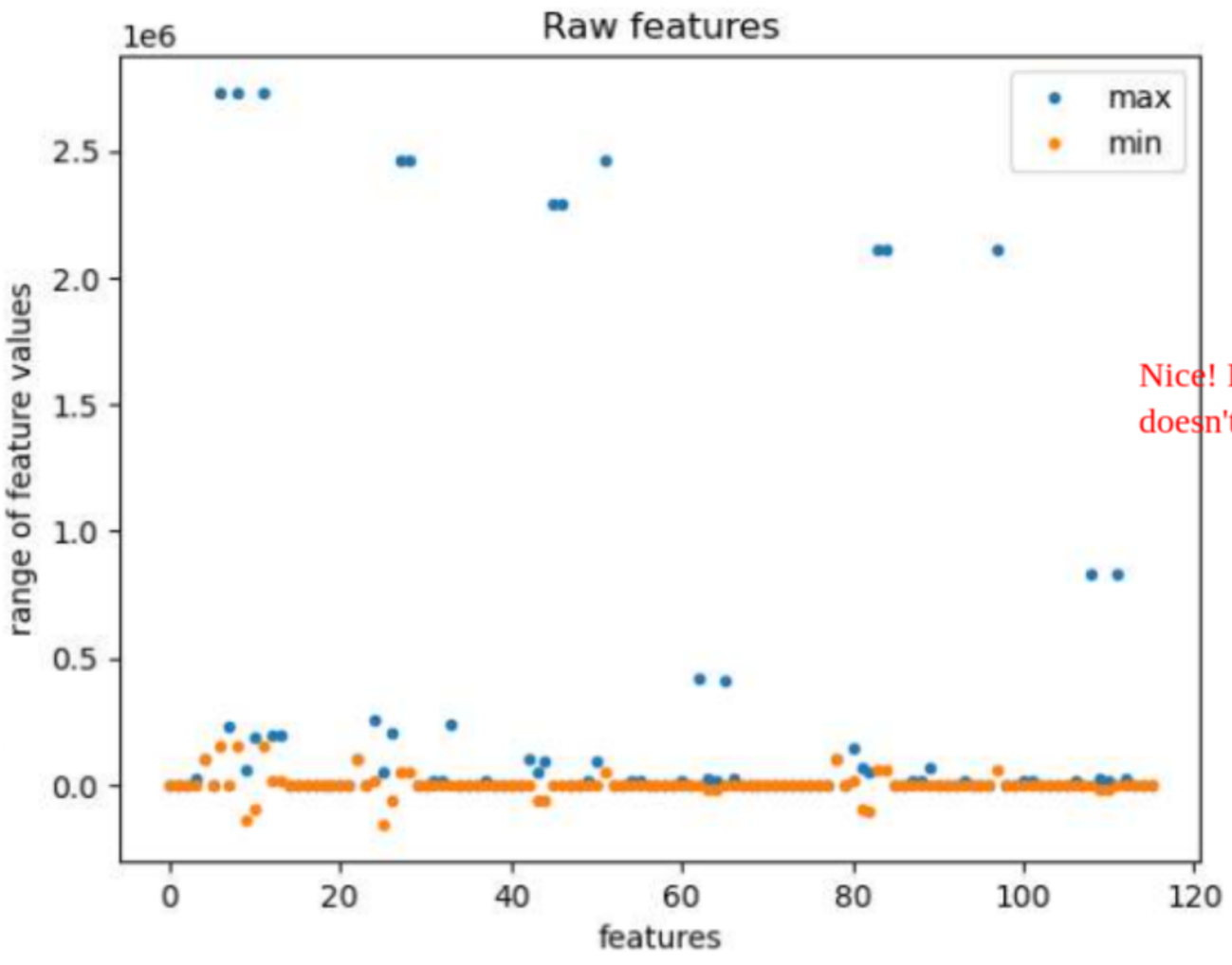
In [20]:
```
features = np.load('data/dijet_features.npy')
labels = np.load('data/dijet_labels.npy')
label_names = ['b', 'c', 'q']  # bottom, charm or light quarks

print(f'{features.shape=}, {labels.shape=}')  # print the shapes

# TODO: print how many samples of each class are present in the data (hint: numpy.unique)

print(f"# of samples of each class ({np.unique(labels, return_counts = True)[0]}) : {np.unique(labels, return_counts = True)[1
```

```
features.shape=(116, 2233), labels.shape=(2233,)
# of samples of each class ([0. 1. 2.]) : [999 864 370]
```

Normalize the data

In [21]:
```
# TODO: report range of features and normalize the data to zero mean and unit variance
max = np.max(features, axis=1)
min = np.min(features, axis=1)

plt.plot(np.arange(0,116,1), max, ".", label = "max")
plt.plot(np.arange(0,116,1), min, ".", label = "min")
plt.legend()
plt.xlabel("features")
plt.ylabel("range of feature values")
plt.title("Raw features")
```

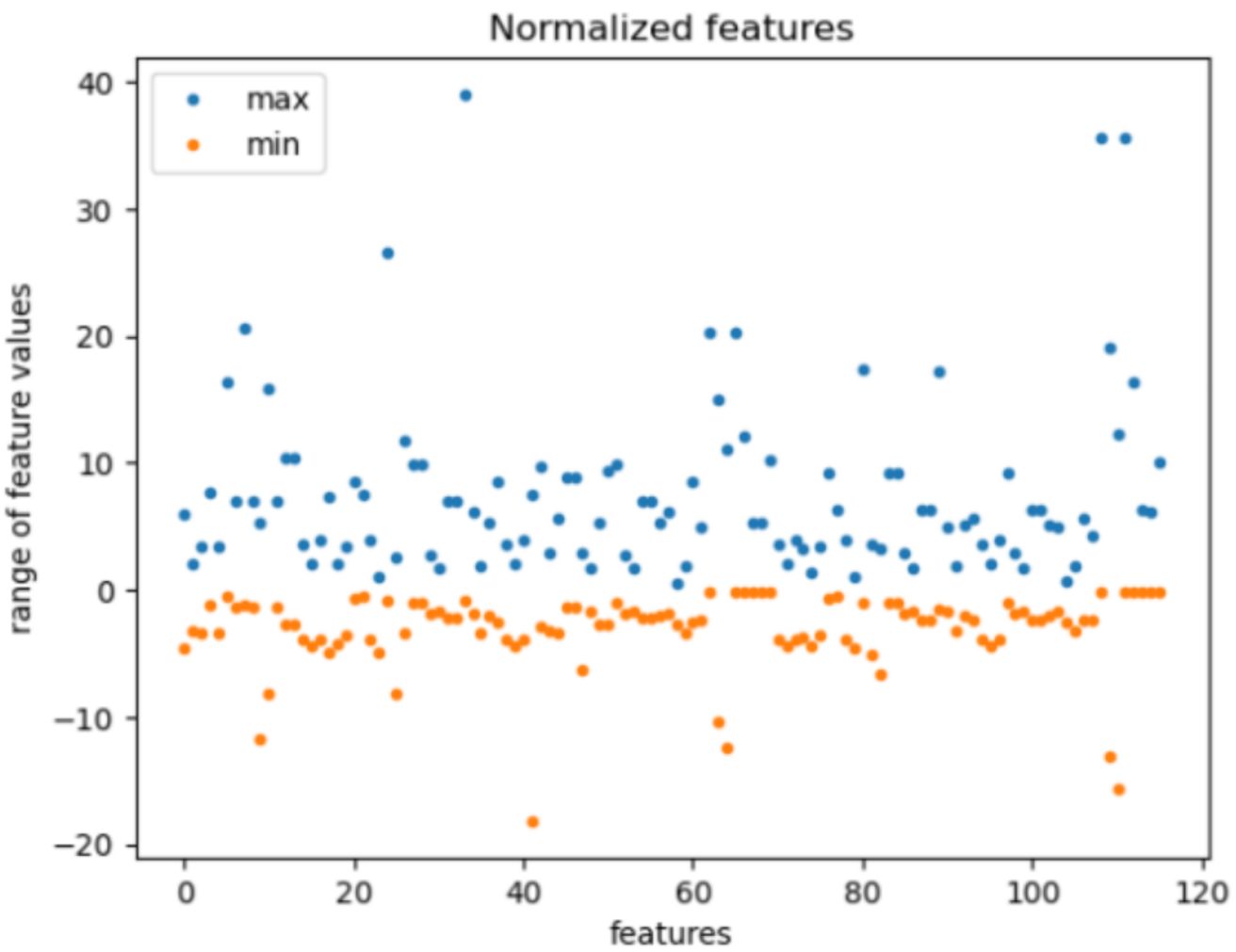
Out[21]: Text(0.5, 1.0, 'Raw features')



Nice! Plotting the min and max values on the same axis probably doesn't make much sense though due to the loss in resolution

In [22]:
```
means = np.mean(features, axis=1)
variances = np.std(features, axis=1)

normalized_features = ((features.T - means) / variances).T

max = np.max(normalized_features, axis=1)
min = np.min(normalized_features, axis=1)

plt.plot(np.arange(0,116,1), max, ".", label = "max")
plt.plot(np.arange(0,116,1), min, ".", label = "min")
plt.legend()
plt.xlabel("features")
plt.ylabel("range of feature values")
plt.title("Normalized features")
```

Out[22]:
```
Text(0.5, 1.0, 'Normalized features')
```



(c)

Compute a 2D PCA projection and make a scatterplot of the result, once without color, once coloring the dots by label. Interpret your results.

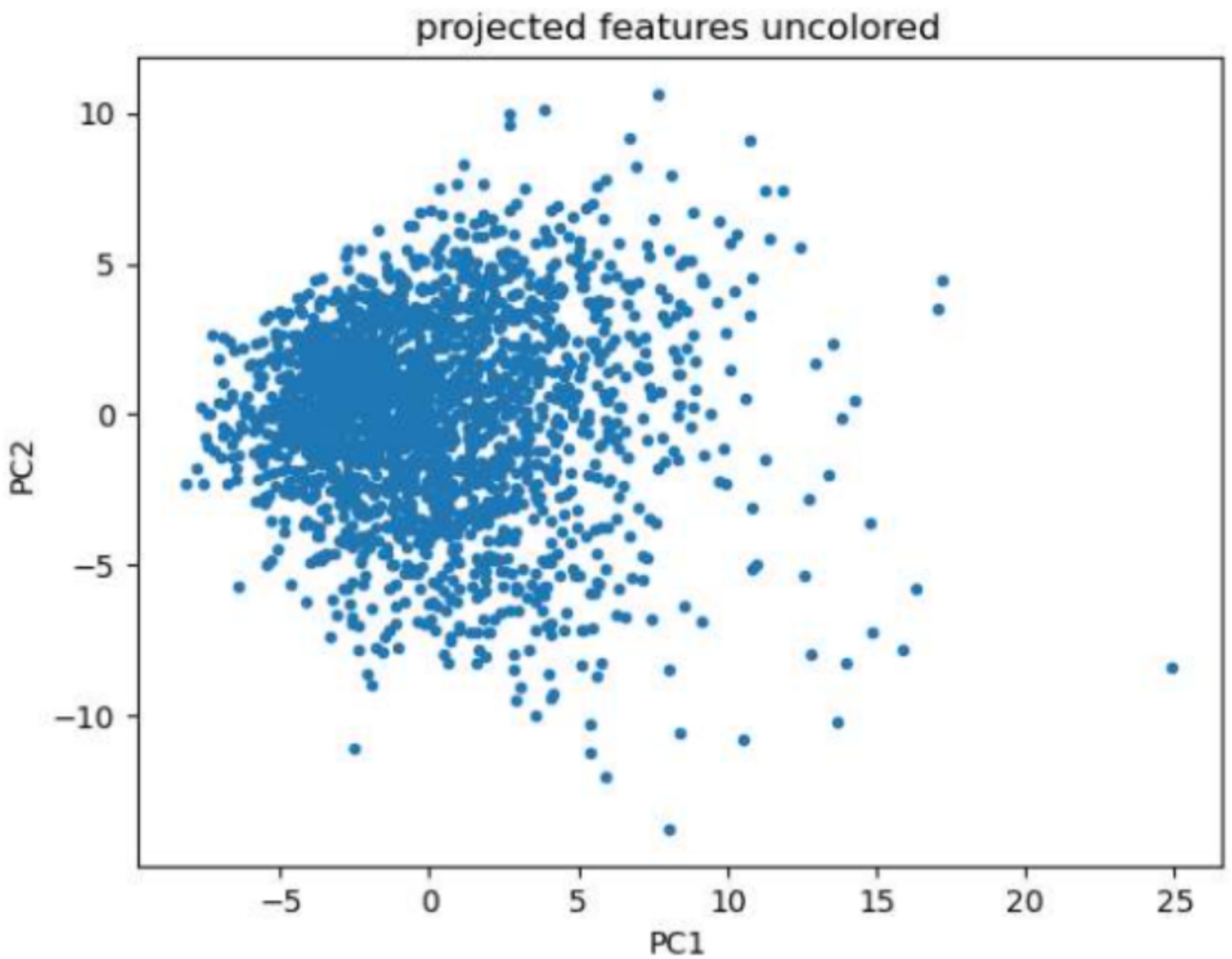
In [23]:
```
# TODO: apply PCA as implemented in (a)
components, X_projected = pca(normalized_features, 2)
```

In [24]:
```
# TODO: make a scatterplot of the PCA projection

plt.scatter(X_projected[0], X_projected[1], marker = ".")
plt.xlabel("PC1")
```

```
plt.ylabel("PC2")
plt.title("projected features uncolored")
```

```
c:\Users\timwe\anaconda3\lib\site-packages\matplotlib\collections.py:192: ComplexWarning: Casting complex values to real discar
ds the imaginary part
  offsets = np.asanyarray(offsets, float)
```

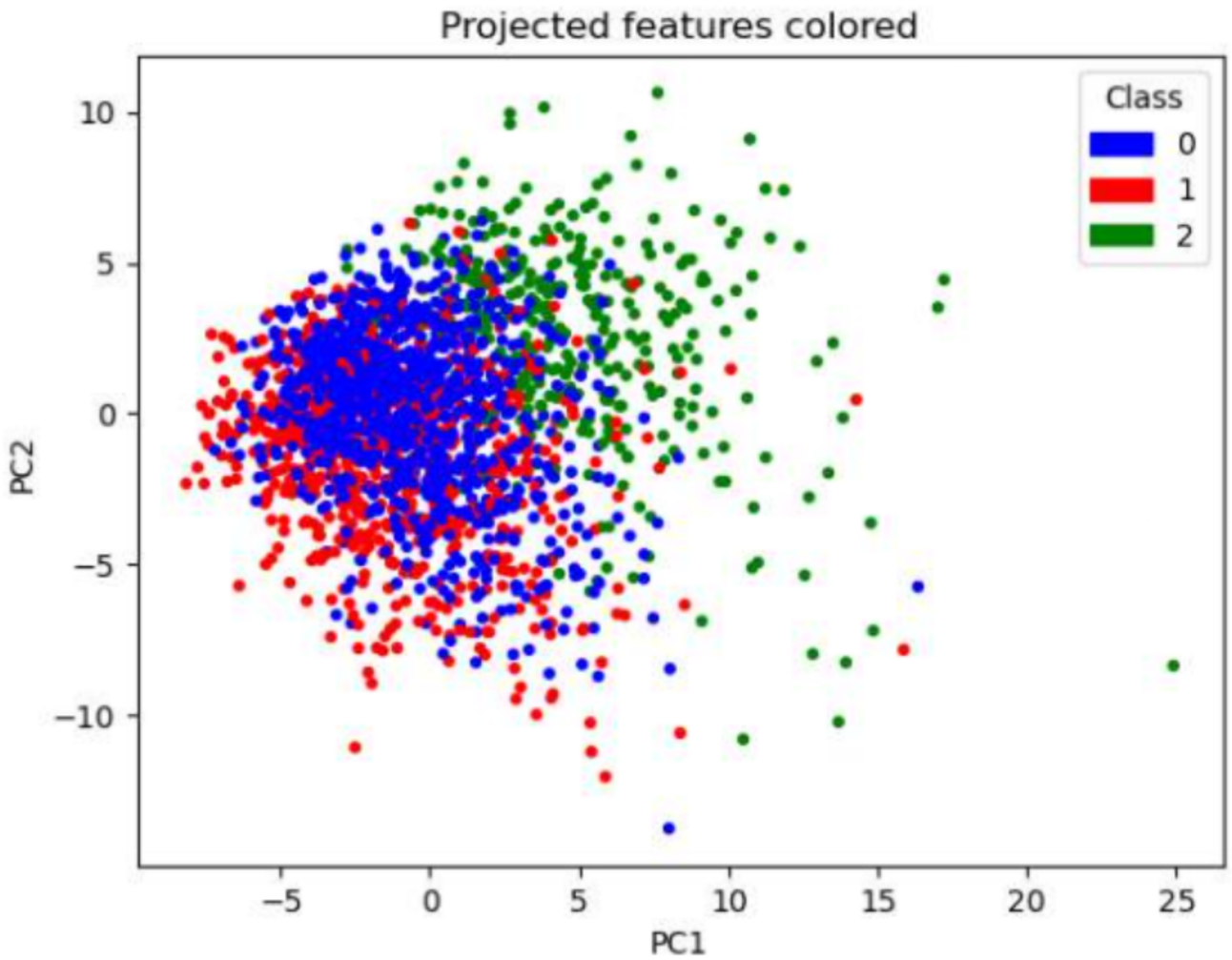
Out[24]: Text(0.5, 1.0, 'projected features uncolored')



In [25]:
```
color_box = ["blue", "red", "green"]
colors = []
for label in labels:
    colors.append(color_box[int(label)])
```

```
plt.scatter(X_projected[0], X_projected[1], marker = ".", color = colors)
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.title("Projected features colored")

blue = mpatches.Patch(color='blue', label='0')
red = mpatches.Patch(color='red', label='1')
green = mpatches.Patch(color='green', label='2')

plt.legend(handles=[blue, red, green], title = "Class")
```

Out[25]: <matplotlib.legend.Legend at 0x20047576d90>



2 Nonlinear Dimension Reduction

In [26]:
```
import umap ### du musst umap-learn installieren
```

In [27]:
```
# if you have not done 1(b) yet, you can load the normalized features directly:
features = np.load('data/dijet_features_normalized.npy')
labels = np.load('data/dijet_labels.npy')
label_names = ['b', 'c', 'q']  # bottom, charm or light quarks
```

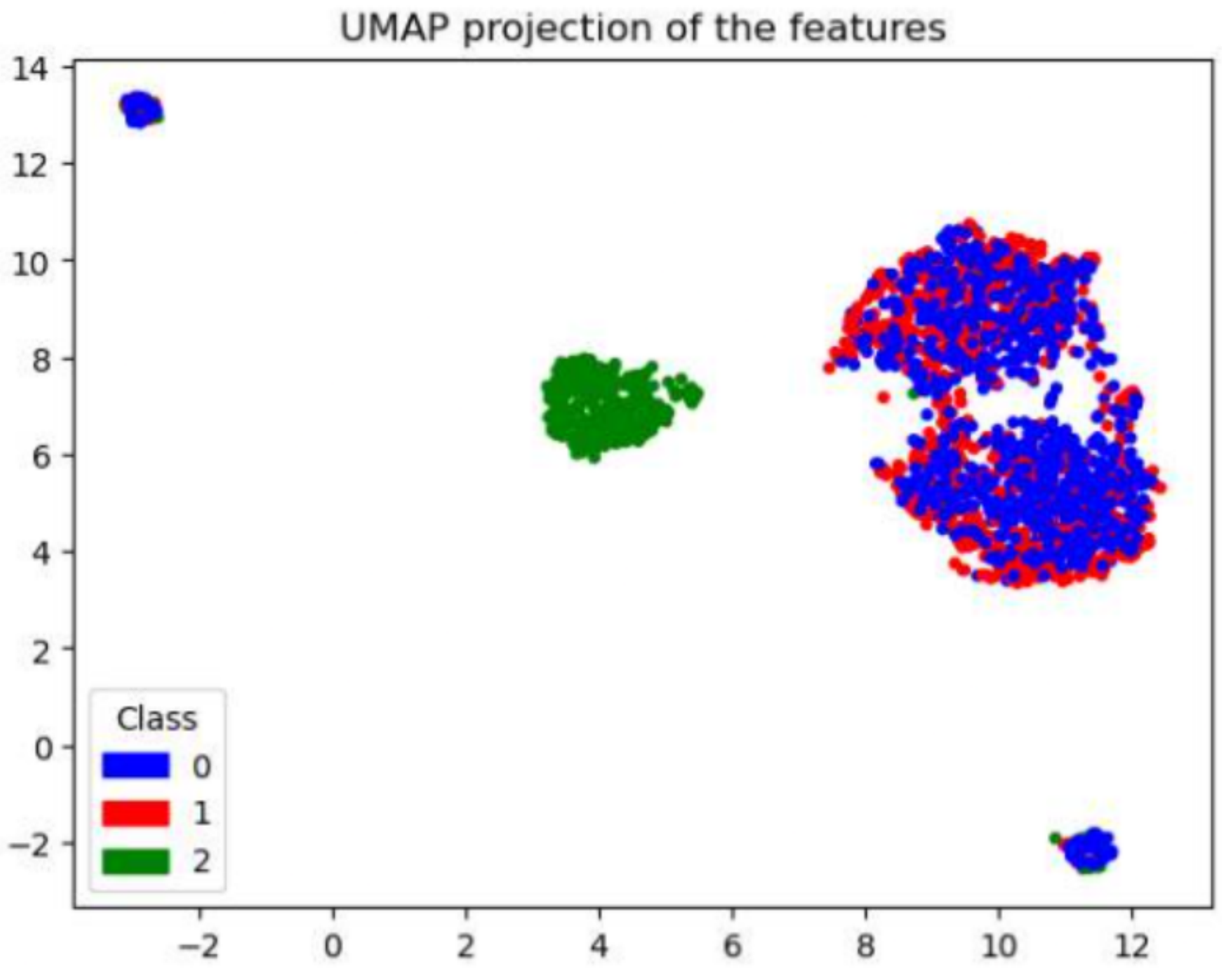
(a)

In [28]:
```
# TODO: Apply umap on the normalized jet features from excercise 1. It will take a couple of seconds.
# note: umap uses a different convention regarding the feature- and sample dimension, N x p instead of p x N!

reducer = umap.UMAP()

embedding = reducer.fit_transform(features.T)
embedding.shape
```

Out[28]:
```
(2233, 2)
```

In [29]:
```
# TODO: make a scatterplot of the UMAP projection

plt.scatter(embedding[:, 0], embedding[:, 1], marker=".", c=colors)
#plt.scatter(embedding[0], embedding[1], marker=".", c=labels)
plt.title('UMAP projection of the features')
plt.legend(handles=[blue, red, green], title = "Class")


# TODO: make a scatterplot, coloring the dots by their label and including a legend with the label names
```

Out[29]:
```
<matplotlib.legend.Legend at 0x20047c12df0>
```

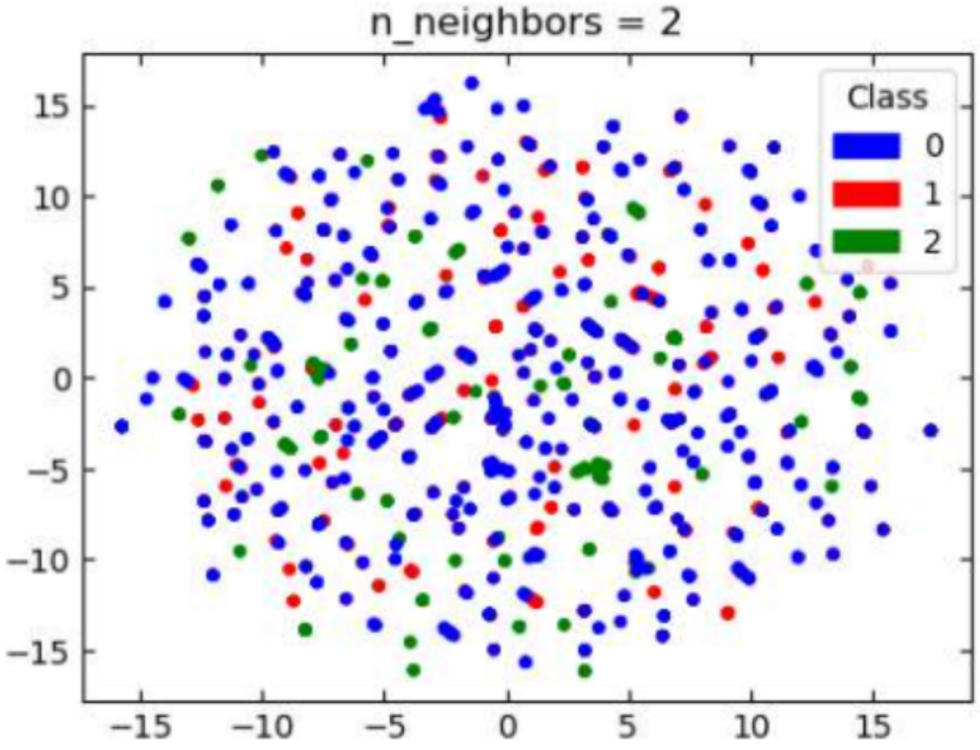


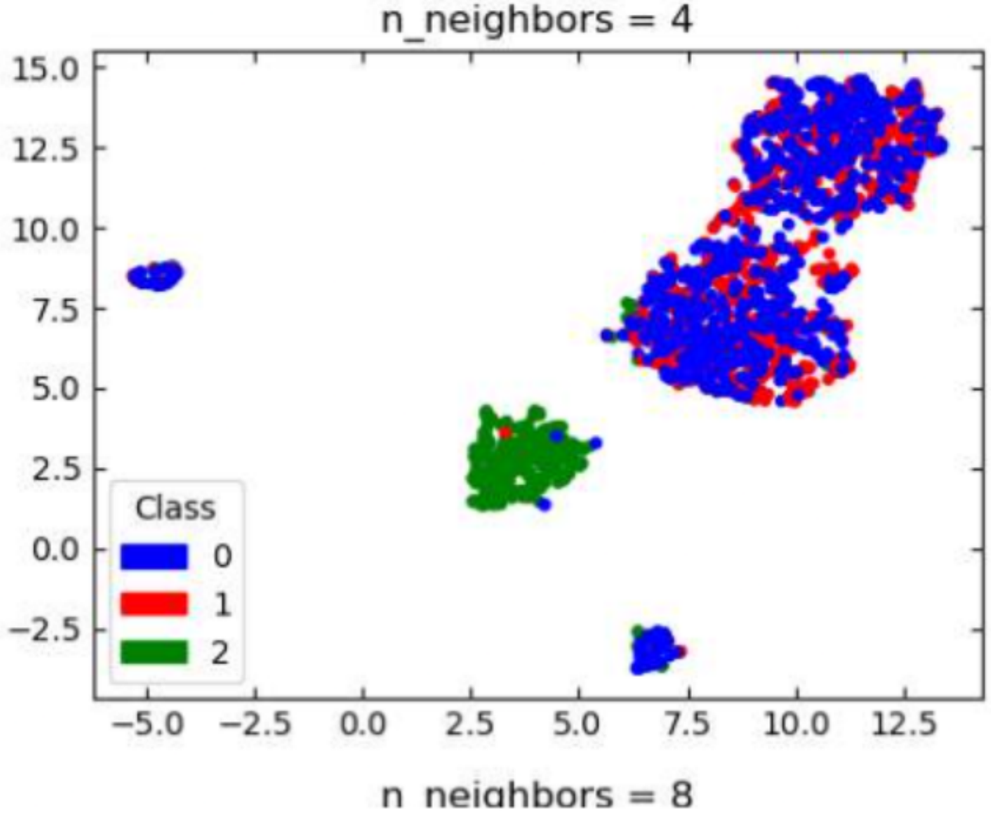
(b)

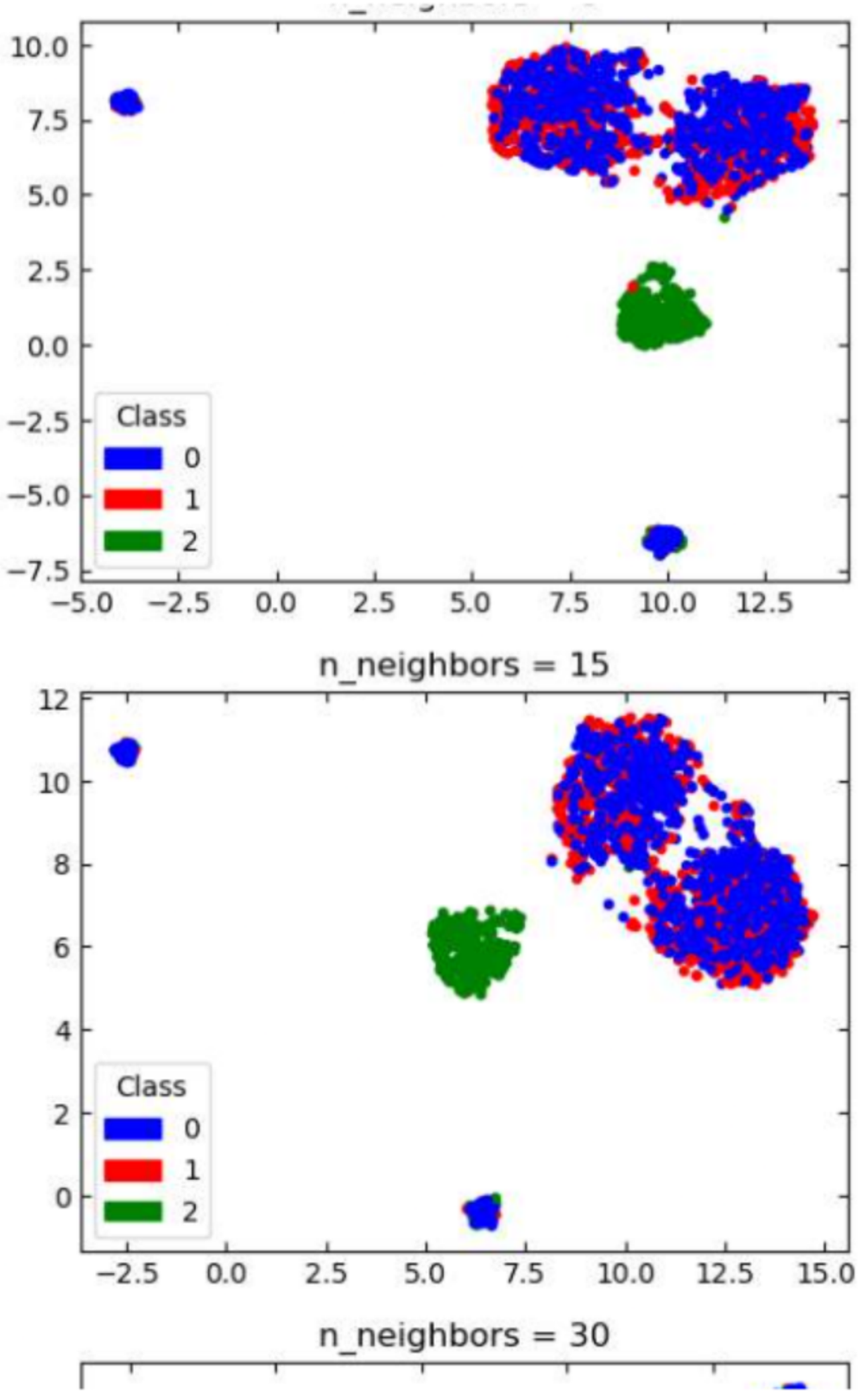
In [30]:
```
embedding_list = []

fig, axs = plt.subplots(7, figsize = (5, 30))

n_neighbors_list = [2, 4, 8, 15, 30, 60, 100]

for i, n_neighbors in enumerate(n_neighbors_list):
    # TODO: repeat the above, varying the n_neighbors parameter of UMAP
    reducer = umap.UMAP(n_neighbors=n_neighbors)
    embedding = reducer.fit_transform(features.T)
    axs[i].scatter(embedding[:, 0], embedding[:, 1], marker=".", c=colors)
```

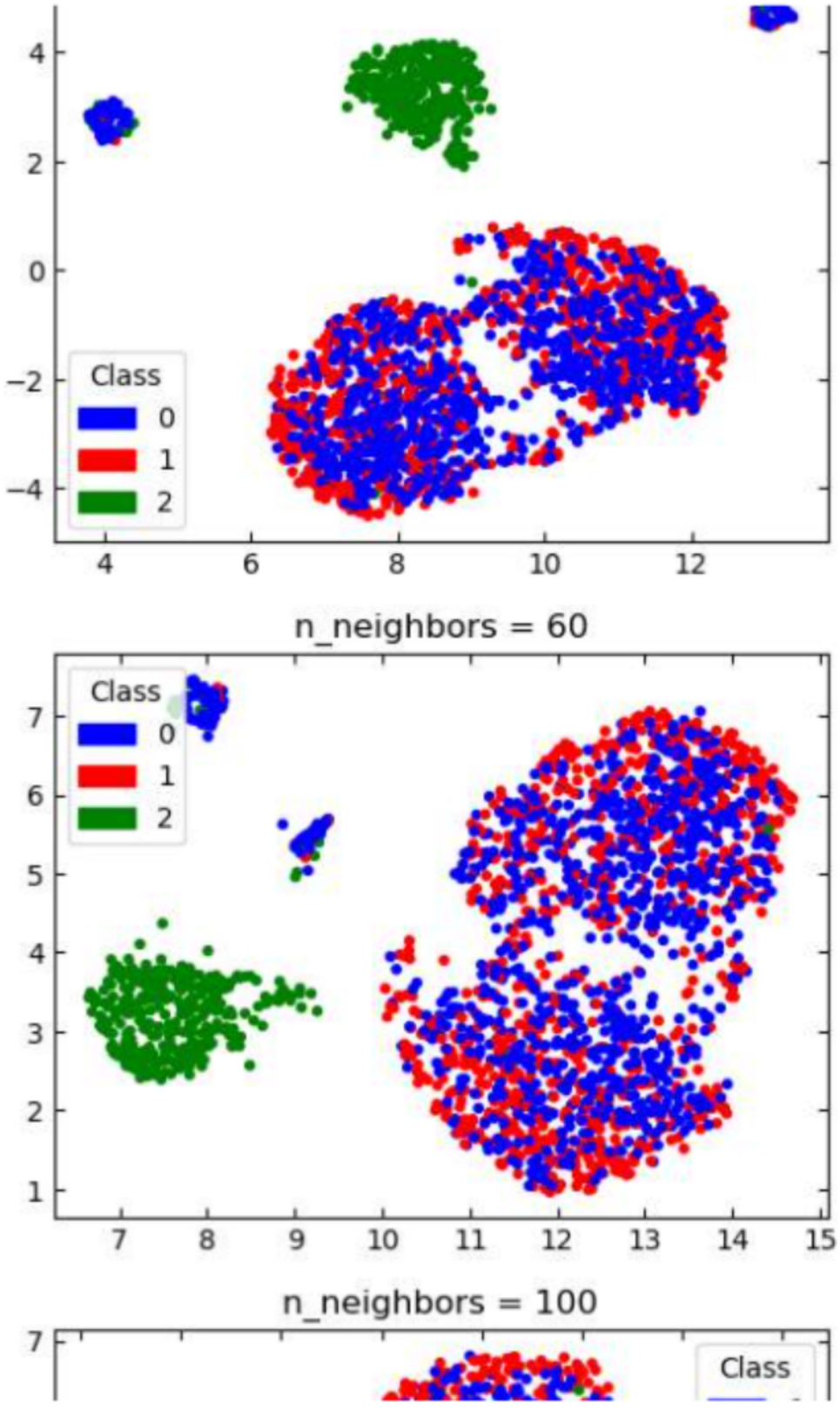
```
    embedding_list.append(embedding)

for i, ax in enumerate(axs):
    ax.tick_params(top=True, right=True, direction="in")
    ax.set_title(f"n_neighbors = {n_neighbors_list[i]}")
    ax.legend(handles=[blue, red, green], title = "Class")
```

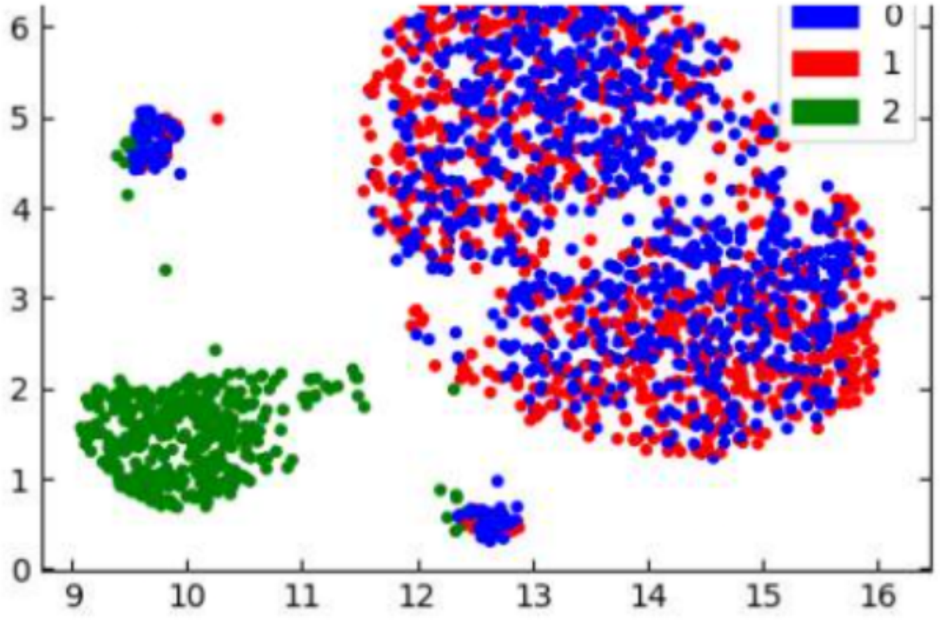
```
c:\Users\timwe\anaconda3\lib\site-packages\umap\spectral.py:550: UserWarning: Spectral initialisation failed! The eigenvector s
olver
failed. This is likely due to too small an eigengap. Consider
adding some noise or jitter to your data.

Falling back to random initialisation!
  warn(
c:\Users\timwe\anaconda3\lib\site-packages\umap\spectral.py:550: UserWarning: Spectral initialisation failed! The eigenvector s
olver
failed. This is likely due to too small an eigengap. Consider
adding some noise or jitter to your data.

Falling back to random initialisation!
  warn(
```


n_neighbors = 2
Class
0
1
2
15
10
5
0
−5
−10
−15
−15
−10
−5
0
5
10
15


n_neighbors = 4
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
Class
0
1
2
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5
n_neighbors = 8


Class
0
1
2
n_neighbors = 15
Class
0
1
2
n_neighbors = 30


Class
0
1
2
n_neighbors = 60
n_neighbors = 100



3 RANSAC

- $p \in (0, 1)$ is the fraction of inlier points.
- $1 - p$ is the fraction of outlier points.
- m is the size of each sampled subset.
- r is the number of repeats (subsets) sampled.

We want the probability of finding at least one outlier-free subset to be 99%, i.e. $P(\text{at least one outlier-free subsset}) = 0.99$.

The probability of sampling an outlier-free subset (all points being inliers) is

$$P(\text{outlier-free subset}) = p^m$$

Thus

$$P(\text{contaminated subset}) = 1 - p^m$$

The probability of all r subsets being contaminated is

$$P(\text{all contaminated}) = (1 - p^m)^r$$

Thus the the probability of finding at least one outlier-free subset is

$$P(\text{at least one outlier-free subsset}) = 1 - (1 - p^m)^r \stackrel{!}{=} 0.99$$

Solving for r gives:

$$r = \frac{\ln 0.01}{\ln 1 - p^m}$$

Correct

4 PCA meets Random Matrix Theory

a)

Since X consists of i.i.d. Gaussian entries, the matrix XX^T has full rotational symmetry, meaning that the directions of its eigenvectors are uniformly distributed over the unit sphere in $\mathbb{R}^p$. Thus, the first principal component (and all the subsequent ones) follows a uniform distribution over the unit sphere in $\mathbb{R}^p$. This is because each direction is equally likely due to the isotropic nature of the Gaussian noise.

Perfect

b)

Intuitively, the eigenvalues should be normally distributed around $\lambda = p/N$ with some variance σ^2 if the number of features p and the sample size N both go to infinity according to the central limit theorem.

Almost, the mean of the eigenvalues will remain centered at zero, while the width of the distribution is determined by lambda

c)