Outline

Task 1: Neural networks ^

In [4]:

```
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
from scipy.sparse import linalg
import networkx as nx
from tabulate import tabulate
import seaborn as sns
```

In [230]:

```
def load_data():
    (x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
    x_train = x_train.astype('float32') / 255
    x_test = x_test.astype('float32') / 255
    # convert labels to categorical samples
    y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
    y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)
    return ((x_train, y_train), (x_test, y_test))
```

In [231]:

```
(x_train, y_train), (x_test, y_test) = load_data()  # Initialise the training and test datasets.
```

In Task 1, we will explore how two different neural network architectures perform in a supervised classification on the Fashion-MNIST dataset of images. This dataset consists of a training set with 60,000 images and a test set with 10,0000 images belonging to ten different classes of items sold by an online fashion store (such as trousers, t-shirt, dress etc.). Both the training and test dataset are well-balanced, meaning they contain 6,000 and 1,000 of each image respectively.

We begin by standardising the input data, using the mean and standard devaiation from the training dataset:

In [232]:

```
mu = x_train.mean(axis = 0)
sigma = x_train.std(axis = 0)

x_train = (x_train - mu) / sigma
x_test = (x_test - mu) / sigma  # Standardise the test data using the mean and standard deviation from the traini
ng data.
```

To aid understanding in the following section, we also note the respective shapes of the training and test arrays:

In [234]:

```
print(f'The training array has shape: {x_train.shape}')
print(f'The testing array has shape: {x_test.shape}')
```

```
The training array has shape: (60000, 28, 28)
The testing array has shape: (10000, 28, 28)
```

1.1 Multi-layer perceptron ^

We begin by implementing a multi-layer perceptron using only NumPy. The network will have the following architecture: an input layer, 5 hidden layers (each with 400 neurons), followed by the output layer with 10 neurons (one for each class). The activation function between all layers is the LeakyReLU(h) with a slope of 0.01 and the softmax function is the activation function on the output layer. We fix the optimisation method to be stochastic gradient descent (SGD) and define the loss function as categorical cross-entropy.

We begin by defining our activation functions `leaky_relu(x)` and `softmax(x)`, along with the differential of the LeakyReLU function `diff_leaky_relu(x)` which will be used for stochastic gradient descent during the backward pass. In 1-dimension these functions can be described as follows:

$$\text{LeakyReLU(x)} = max(0.01x, x) \qquad \text{softmax(x)} = \frac{1}{1+e^{-x}}$$

which is applied element-wise in the following implementations.

In [253]:

```
def leaky_relu(x, alpha=0.01):
    '''
    Return the LeakyReLu function evaluated for a matrix of inputs with a slope of 0.01.
    '''
    data = [np.maximum(alpha*x_i, x_i) for x_i in x]
    return np.array(data, dtype=float)
```

In [254]:

```
def diff_leaky_relu(x, alpha=0.01):
    '''
    Return the differential of the LeakyReLu function evaluated for a mtrix of inputs with a slope of 0.01.
    '''
    data = [[alpha if x_i < 0 else 1 for x_i in x[row]] for row in range(x.shape[0])]
    return np.array(data, dtype=float)
```

In [255]:

```
def softmax(x):
    '''
    Return the Softmax function evaluated for a matrix of inputs with additional stabalisation.
    '''
    stable_x = x - np.max(x, axis = 0)  # We stabalise the inputs since otherwise the exponential causes our func
tion to 'blow up'.
    return np.exp(stable_x) / np.sum(np.exp(stable_x), axis = 0)
```

We use activation functions to assist in deciding whether a neuron should be activated of not: they help decide whether the neuron's input to the network is important or not in the process of prediction. Activation functions must be non-linear to offer any insight beyond a single linear neuron and their main function is to provide the non-linearity neaural networks need to implement complex mapping functions. The activation functions we have used are LeakyReLU, which is an improvement on the more conventional ReLU function which suffers from the 'dead neuron' problem, and softmax which returns a probability vector, which in this case represents the probability that the input image belongs to one of ten image classes. We also create a dictionary for the network's architecture that can be easily called upon throughout this question:

In [240]:

```
architecture = [
    {'units': 784, 'activation': 'none'},  # 784 = 28*28 since the images are 28x28 pixels.
    {'units': 400, 'activation': 'leaky relu'},
    {'units': 400, 'activation': 'leaky relu'},
    {'units': 400, 'activation': 'leaky relu'},
    {'units': 400, 'activation': 'leaky relu'},
    {'units': 400, 'activation': 'leaky relu'},
    {'units': 10, 'activation': 'softmax'},
]
```

We will create the parameters for our multi-layer perceptron as NumPy arrays and initialise the kernel values with samples from a zero-mean gaussian distribution with variance $\frac{2}{(n_{in}+n_{out})}$, where n_{in} and n_{out} are the number of neurons going in and out of the dense layer respectively. The bias parameters will be initialised to zeros. This initialisation strategy is known as Glorot initialisation.

In [246]:

```
def parameters():
    '''
    Return a list of the weights and biases using Glorot initialisation.
    '''
    params = []

    for i in range(len(architecture) - 1):
        var = 2 / (architecture[i+1]['units'] + architecture[i]['units'])
        W = np.random.randn(architecture[i+1]['units'], architecture[i]['units']) * np.sqrt(var)  # Initialise th
e weights using standard normal distribution.
        b = np.zeros((architecture[i+1]['units'], 1))  # Initialise the biases to zero.
        params.append([W, b])

    return params
```

We also define a `dense` function to be called in our `forward_pass` function below:

In [262]:

```
def dense(x, W, b):
    '''
    Return the layer's pre-activations given wieght and bias parameters.
    '''
    h = b + W @ x
    return h
```

In addition, we define our loss function and accuracy. The loss is categorical cross-entropy and the accuracy is simply the proportion of time our model is correct in its prediction. Categorical cross-entropy is defined as follows:

$$\text{Catergorical cross-entropy} = -\sum_{j}^{\text{output size}} y_j \log(\hat{y}_j)$$

where y is the true label and $\hat{y}$ is our prediction probability. The minus sign ensures that the loss gets smaller when the distributions get closer to each other. Without loss of generality, we will return the mean categorical cross-entropy to avoid our loss exploding.

In [251]:

```
def cross_entropy(y, y_hat):
    '''
    Return the mean categorical cross-entropy between the predicted and true values.
    '''
    y_hat = y_hat + 1e-15  # Offset each predicted value by a negligible amount to prevent any log(0) errors occu
ring.
    return (-1 * ((y * np.log(y_hat))[y != 0]).mean())  # Only calculate the values where y = 1 to reduce computa
tional time.
```

In [252]:

```
def accuracy(y, y_hat):
    '''
    Return the mean accuracy of the y_hat predictions for y.
    '''
    pred = (y_hat == y_hat.max(axis=0)[None,:]).astype(float)
    return (pred == y).all(axis = 0).mean()
```

We now initialise our forward and backward pass functions. The forward pass function returns a probability distribution for the taget data point being in each respective class (i.e. the type of clothing in the image). Ultimately, we will take the class with the highest probability to be the model's prediction. The forward pass function will also return a memory of the pre-activations which can be utilised in the backward pass function. The backward pass function takes the error in the prediction and propogates it back through the the network to obtain an additional error at each layer. These errors are then utilised in computing the gradients for stochastic gradient descent to update the initial parameters in a way that minimises our loss function.

In [256]:

```
def forward_pass(x, params):
    '''
    Return the prediction for a single run through the multi-layer perceptron and return a list of the pre and po
st
    activations to be used in backpropogation.
    '''
    memory = [[x, x]]  # Intitialise input data as the pre and post activations.
    h = x  # Input to the first layer is just our data.

    for i in range(len(params)):  # Recall len(params) = len(architecture) - 1
        a = dense(h, params[i][0], params[i][1])

        if architecture[i+1]['activation'] == 'leaky relu':
            h = leaky_relu(a, alpha=0.01)

        if architecture[i+1]['activation'] == 'softmax':
            h = softmax(a)
            preds = h

        memory.append([a, h])  # Append the pre and post activations to the memory.

    return preds, memory
```

In [274]:

```
def backward_pass_with_sgd(y, preds, memory, params, batch_size, learning_rate):
    '''
    Return the updated weights according to stochastic gradient descent and backwards propogation having complete
d a
    forward pass through the network.
    '''
    grads = [preds - y]  # Initialise grads with the output layer error.

    for i in reversed(range(len(params))):
        [a, h] = memory[i]
        [W, b] = params[i]

        if architecture[i]['activation'] == 'leaky relu':
            delta_i = diff_leaky_relu(a) * (W.T @ grads[-1])  # The error in layer i.

        dW_i = grads[-1] @ h.T  # The weights at layer i.
        db_i = grads[-1].mean(axis = 1)  # The bias at layer i.
        db_i = db_i.reshape((db_i.size, 1))  # Reshape to match the shape of b.

        sgd_W = learning_rate * dW_i / batch_size  # Stochastic gradient descent for the weights.
        sgd_b = learning_rate * db_i / batch_size  # Stochastic gradient descent for the biases.

        params[i] = [W - sgd_W, b - sgd_b]  # Update the weights and biases.
        grads.append(delta_i)  # Append delta_i to grads to be used in the next layer.

    return params
```

We can now complile our multi-layer perceptron model by calling the functions defined above in a pythonic way. The model will work as follows:

1. Batch the training data into equal sized batches of 256.
2. Propogate the batches of data through the `foward_pass` function having initilised the weights according the a standard random normal distribution with Glorot variance and zero bias.
3. Back propogate the error of each prediction through the network using `backward_pass_with_sgd` using stochastic gradient descent to minimise the cost of each batch.
4. Having updated the weights and biases using each batch, feed through the entire training dataset and compute the categorical cross-entropy loss.
5. Repeat for as many epochs as required, with the aim of decreasing the loss and increasing the accuracy on the test dataset.

In [275]:

```
def mlp_model(x_train, y_train, x_test, y_test, batch_size, learning_rate, epochs):
    '''
    Return the model's categorical cross-entropy loss and accuracy using a multi-layer perceptron and a specified
learning rate.
    '''
    np.random.seed(0)  # Set the seed for all random aspects in this cell to make results reproducible.

    # Flatten the input data to allow matrix operations (it is currently a matrix of matrices).
    x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1] * x_train.shape[2]))
    x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1] * x_test.shape[2]))

    # Transpose the input and output data to make them correct shape for the functions called below.
    x_train = x_train.T
    y_train = y_train.T
    x_test = x_test.T
    y_test = y_test.T

    # Initialise the batches.
    num_iterations = round(x_train.shape[1] / batch_size)
    batches = np.array_split(np.random.permutation(x_train.shape[1]), num_iterations)

    # Initialise the weights and biases.
    params = parameters()

    # Initialise the loss and accuracy lists for the training and test datasets.
    train_losses = []
    train_accs = []
    test_losses = []
    test_accs = []

    # Initial pass of the data, which will result in random predictions.
    preds_train = forward_pass(x_train, params)[0]
    preds_test = forward_pass(x_test, params)[0]

    # Add the initial losses and accuracies to their respective lists.
    train_losses.append(cross_entropy(y_train, preds_train))
    train_accs.append(accuracy(y_train, preds_train))
    test_losses.append(cross_entropy(y_test, preds_test))
    test_accs.append(accuracy(y_test, preds_test))

    # Complete for number of epochs.
    for epoch in range(epochs):

        # Complete for each batch.
        for i in range(num_iterations):
            batch = batches[i]
            x_batch = x_train[:, batch]
            y_batch = y_train[:, batch]

            batch_preds, memory = forward_pass(x_batch, params)
            params = backward_pass_with_sgd(y_batch, batch_preds, memory, params, batch_size, learning_rate)

        # Predict on the entire dataset having optimised the parameters using batches.
        preds_train = forward_pass(x_train, params)[0]
        preds_test = forward_pass(x_test, params)[0]

        # Add this epoch's losses and accuracies to their respective lists.
        train_losses.append(cross_entropy(y_train, preds_train))
        train_accs.append(accuracy(y_train, preds_train))
        test_losses.append(cross_entropy(y_test, preds_test))
        test_accs.append(accuracy(y_test, preds_test))

    return train_losses, train_accs, test_losses, test_accs
```

Having implemented the architecture of our multi-layer perceptron, we are now ready to train the network over various learning rates and number of epochs and observe the results.

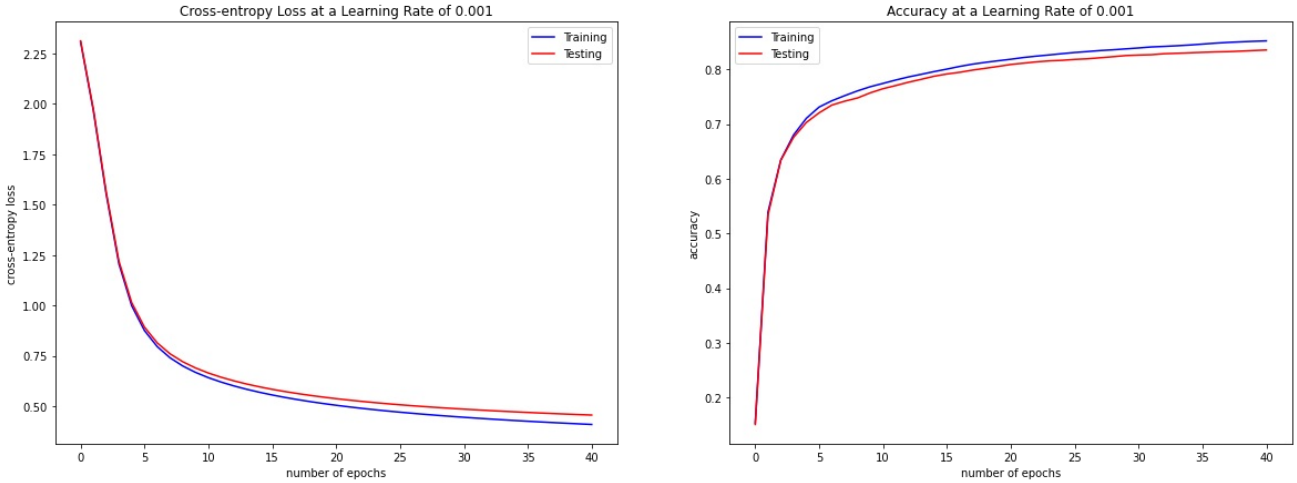
1.1.1

In [276]:

```
mlp_model_1 = mlp_model(x_train, y_train, x_test, y_test, 256, 1e-3, 40)
```

In [277]:

```
fig1, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 7))

ax1.plot([i for i in range(41)], mlp_model_1[0], color='blue', label='Training')
ax1.plot([i for i in range(41)], mlp_model_1[2], color='red', label='Testing')
ax1.set_xlabel('number of epochs')
ax1.set_ylabel('cross-entropy loss')
ax1.set_title('Cross-entropy Loss at a Learning Rate of 0.001')
ax1.legend()

ax2.plot([i for i in range(41)], mlp_model_1[1], color='blue', label='Training')
ax2.plot([i for i in range(41)], mlp_model_1[3], color='red', label='Testing')
ax2.set_xlabel('number of epochs')
ax2.set_ylabel('accuracy')
ax2.set_title('Accuracy at a Learning Rate of 0.001')
ax2.legend()
plt.show()
```



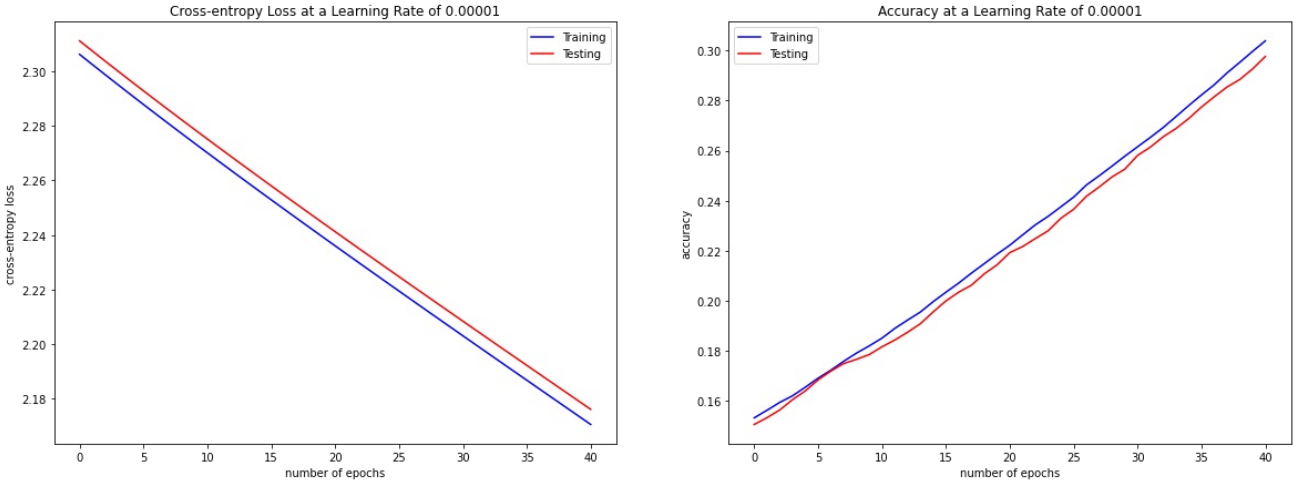
1.1.2

In [278]:

```
mlp_model_2 = mlp_model(x_train, y_train, x_test, y_test, 256, 1e-5, 40)
mlp_model_3 = mlp_model(x_train, y_train, x_test, y_test, 256, 1e-1, 40)
```

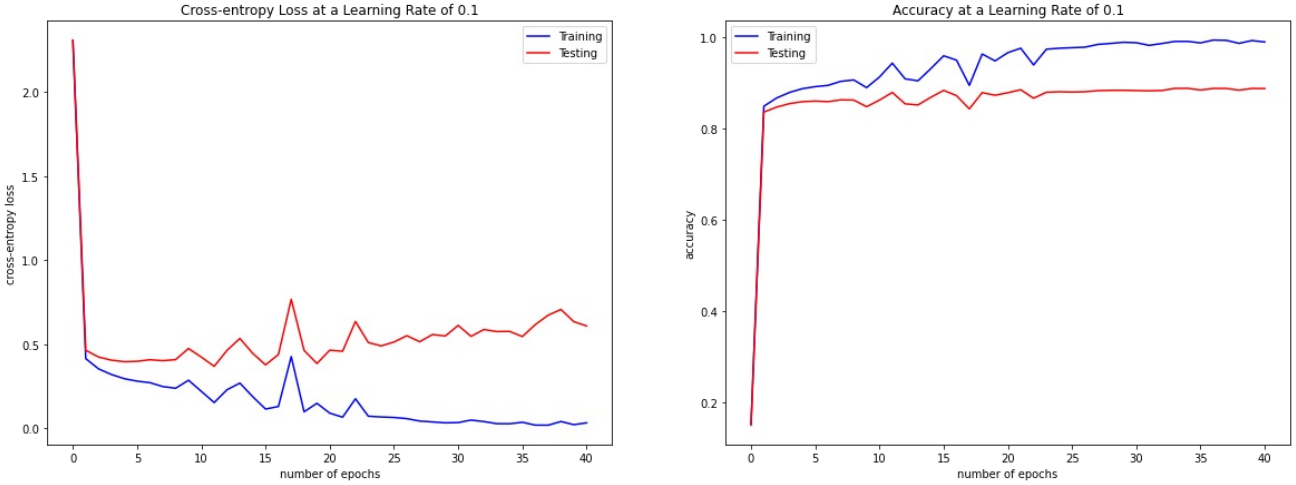
In [279]:

```
fig1, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 7))

ax1.plot([i for i in range(41)], mlp_model_2[0], color='blue', label='Training')
ax1.plot([i for i in range(41)], mlp_model_2[2], color='red', label='Testing')
ax1.legend()
ax1.set_xlabel('number of epochs')
ax1.set_ylabel('cross-entropy loss')
ax1.set_title('Cross-entropy Loss at a Learning Rate of 0.00001')

ax2.plot([i for i in range(41)], mlp_model_2[1], color='blue', label='Training')
ax2.plot([i for i in range(41)], mlp_model_2[3], color='red', label='Testing')
ax2.legend()
ax2.set_xlabel('number of epochs')
ax2.set_ylabel('accuracy')
ax2.set_title('Accuracy at a Learning Rate of 0.00001')
plt.show()
```



In [280]:

```
fig1, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 7))

ax1.plot([i for i in range(41)], mlp_model_3[0], color='blue', label='Training')
ax1.plot([i for i in range(41)], mlp_model_3[2], color='red', label='Testing')
ax1.legend()
ax1.set_xlabel('number of epochs')
ax1.set_ylabel('cross-entropy loss')
ax1.set_title('Cross-entropy Loss at a Learning Rate of 0.1')

ax2.plot([i for i in range(41)], mlp_model_3[1], color='blue', label='Training')
ax2.plot([i for i in range(41)], mlp_model_3[3], color='red', label='Testing')
ax2.legend()
ax2.set_xlabel('number of epochs')
ax2.set_ylabel('accuracy')
ax2.set_title('Accuracy at a Learning Rate of 0.1')
plt.show()
```



It is clear from the above plots that varying the learning rate leads to distinct differences in the model's performance. We begin by discussing `mlp_model_2` which has a learning rate of 0.00001. It is clear that this learning rate is too small: after 40 epochs there is no indication of the loss converging or the accuracy begining to plateau for either dataset, and each respective graph presents a roughly linear line. This suggests the model has not yet reached an optimal minimum and would benefit from more epochs. Additionally, whilst we do not plot the model with this learning rate and a higher number of epochs, we would expect the model to underfit the data and miss out on some of the image's more intricate features. In essence, this small learning rate leads to the model not developing a deep enough understanding of the image data, and would most likely plateau at a sub-optimal accuracy (i.e. less than the 80% acheived elsewhere).

Given this underfitting, we can observe the graph for a learning rate of 0.1 in `mlp_model_3` , which is an exagerated attempt to prevent the model underfitting. Whilst this model does acheive a high accuracy and low loss in a small number of epochs (after just one epoch the model has over 80% accuracy on the test dataset) it is clear the model is now overfitting. This can be seen by the training data set's convergence to an accuracy of 1 whilst the test data set's accuracy remains at around 0.8, indicating that whilst the model fits the data it was trained on very well, it is not generalisable to unseen datasets of the same variety. Whilst this high learning rate does in fact produce a higher accuracy than the other two learning rates (0.00001 and 0.001) this overfitting is a cause for concern and a smaller learning rate should be preferred. It is interesting to note that for a learning rate of 0.1, the accuracy remains relatively constant whilst the loss function is increasing. This is due to the loss function we have implemented, and its dependance on the logarithm. The logarithm is non-linear and two values that are symmetric about 0.5 will result in non-correlated contributions to the overall loss function: the cross-entropy loss function is skewed to penalise bad predictions more than it rewards good predictions. This explains why the loss function can increase whilst the model maintains a constant but sound understanding of 80% of the data.

Finally, we come to `mlp_model_1` which has a learning rate of 0.001. Whilst this model's accuracy remains lower than that of `mlp_model_3` it is clear that this is our optimal learning rate, since it suggests the model is neither underfitting or overfitting the data, and could therefore be optimised to achieve the highest accuracy and lowest loss. Such optimisations could be a strategic initial choice of weights and biases or via implementing a decreasing learning rate as the number of epochs increases. We explore this learning rate over 80 epochs below.

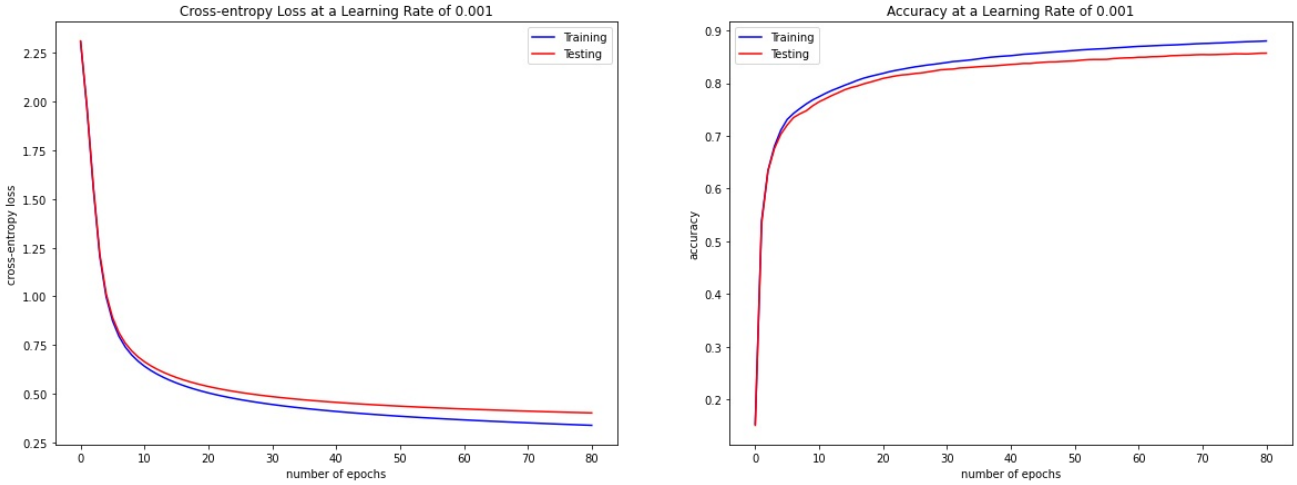
1.1.3

In [299]:

```
mlp_model_4 = mlp_model(x_train, y_train, x_test, y_test, 256, 1e-3, 80)
```

In [302]:

```
fig1, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 7))

ax1.plot([i for i in range(81)], mlp_model_4[0], color='blue', label='Training')
ax1.plot([i for i in range(81)], mlp_model_4[2], color='red', label='Testing')
ax1.legend()
ax1.set_xlabel('number of epochs')
ax1.set_ylabel('cross-entropy loss')
ax1.set_title('Cross-entropy Loss at a Learning Rate of 0.001')

ax2.plot([i for i in range(81)], mlp_model_4[1], color='blue', label='Training')
ax2.plot([i for i in range(81)], mlp_model_4[3], color='red', label='Testing')
ax2.legend()
ax2.set_xlabel('number of epochs')
ax2.set_ylabel('accuracy')
ax2.set_title('Accuracy at a Learning Rate of 0.001')
plt.show()
```



It is clear from the above plot why 0.001 should be considered our optimal learning rate out of the three presented: as the number of epochs increases, the test dataset's accuracy continues to increase and its loss continues to decrease. This is in contrast to the other learning rates which either learnt about the dataset too slowly, presumably plateauing at a sub-optimal accuracy, or simply maximised its model's accuracy after a small number of epochs. Since we have our optimal learning rate, when we increase the number of epochs the model can continue to learn new, more intricate details about the dataset and therefore improve its accuracy and reduce its cross-entropy loss. However, despite this monotonic improvement in both accuracy and loss, the model still seems to be slightly overfitting the data it has been trained on, due to the divergence of the training and testing lines: we would expect the testing accuracy and loss of a perfectly optimised learning rate to converge with the accuracy and loss of the training dataset. We can tackle these issues via methods such as dropout, as will be seen in the following section.

1.2 Convolutional neural network (CNN) ^

In [282]:

```
from tensorflow.keras import Sequential, regularizers
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPool2D, LeakyReLU, Dropout
from tensorflow.keras.models import Model
```

In [283]:

```
train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
test_dataset = tf.data.Dataset.from_tensor_slices((x_test, y_test))

train_dataset = train_dataset.shuffle(buffer_size = 60000)  # Shuffle the entire training dataset.

train_dataset = train_dataset.batch(batch_size = 256)  # Batch the datasets, as before.
test_dataset = test_dataset.batch(batch_size = 256)
```

As before, we print the key features of the `train_dataset` to aid understanding in the folowing section:

In [288]:

```
print(train_dataset)
```

```
<BatchDataset element_spec=(TensorSpec(shape=(None, 28, 28), dtype=tf.float32, name=None), TensorSpe
c(shape=(None, 10), dtype=tf.float32, name=None))>
```

1.2.1

We now implement a convolutional neural network using TensorFlow. A convolutional neural network can be thought of as a multi-layer perceptron with a special structure that allows the model to exploit spatial or temporal invariance in recognition. The network has the following architecture: an input layer, 5 hidden layers (of which the first 4 are convolutional layers and the last is a fully-connected layer), followed by the output layer with 10 neurons (one for each respective class). With regard to the hidden layers, all of the convolutional layers apply 3×3 filters, but the first two use 8 feature maps and the last two use 16 feature maps. The last convolutional layer is followed by a 2×2 maximum pooling layer and the fully-connected layer has 64 neurons. Once again we use LeakyReLU(x) and Softmax(x) as our activation functions and stochastic gradient descent with categorical cross-entropy as our loss function. Our input shape will be `(28, 28, 1)` since each image consists of 28×28 pixels and is greyscale.

In [289]:

```
model_121 = Sequential([
    Conv2D(8, (3, 3), activation = LeakyReLU(alpha=0.01), input_shape = (28, 28, 1)),
    Conv2D(8, (3, 3), activation = LeakyReLU(alpha=0.01)),
    Conv2D(16, (3, 3), activation = LeakyReLU(alpha=0.01)),
    Conv2D(16, (3, 3), activation = LeakyReLU(alpha=0.01)),
    MaxPool2D((2, 2)),
    Flatten(),
    Dense(64, activation = LeakyReLU(alpha=0.01)),
    Dense(10, activation = 'softmax')
    ])
```

In [290]:

```
model_121.summary()  # Print a summary of the model to easily visualise what is happening at each layer.
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 8)         80

 conv2d_1 (Conv2D)           (None, 24, 24, 8)         584

 conv2d_2 (Conv2D)           (None, 22, 22, 16)        1168

 conv2d_3 (Conv2D)           (None, 20, 20, 16)        2320

 max_pooling2d (MaxPooling2D  (None, 10, 10, 16)       0
 )

 flatten (Flatten)           (None, 1600)              0

 dense (Dense)               (None, 64)                102464

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 107,266
Trainable params: 107,266
Non-trainable params: 0
_________________________________________________________________
```

In [291]:

```
sgd = tf.keras.optimizers.SGD(learning_rate = 1e-3)  # Initialise the optimiser to be stochastic gradient descent
.
loss_fn = tf.keras.losses.CategoricalCrossentropy()  # Initialise the loss function to be categorical cross-entro
py.
```

We now train our model on the batches of 256 data points with a learning rate of 0.001 for 40 epochs. During this initial training procedure we will print the losses and accuracies after each epoch, however we will remove this aspect in subsequent models in the interest of space and readability.

In [292]:

```
model_121.compile(optimizer = sgd, loss = loss_fn, metrics = ['accuracy']) # Tracks accuracy
history_121 = model_121.fit(train_dataset, validation_data = test_dataset, epochs = 40)
```

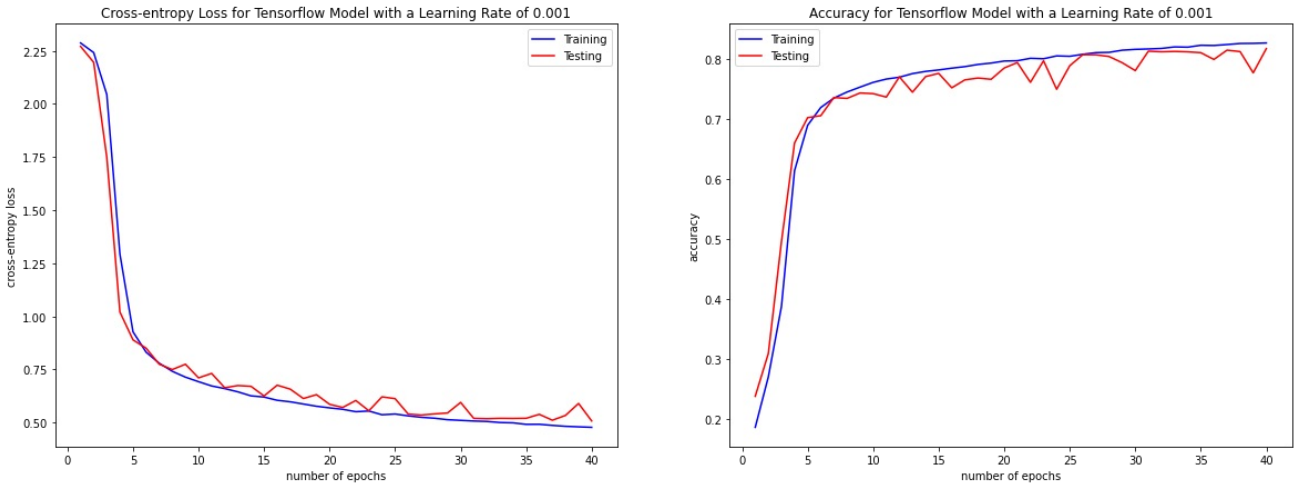
```
Epoch 1/40
235/235 [==============================] - 26s 106ms/step - loss: 2.2867 - accuracy: 0.1862 - val_lo
ss: 2.2705 - val_accuracy: 0.2380
Epoch 2/40
235/235 [==============================] - 30s 127ms/step - loss: 2.2413 - accuracy: 0.2703 - val_lo
ss: 2.1955 - val_accuracy: 0.3094
Epoch 3/40
235/235 [==============================] - 27s 114ms/step - loss: 2.0441 - accuracy: 0.3875 - val_lo
ss: 1.7488 - val_accuracy: 0.4957
Epoch 4/40
235/235 [==============================] - 27s 115ms/step - loss: 1.2929 - accuracy: 0.6142 - val_lo
ss: 1.0201 - val_accuracy: 0.6604
Epoch 5/40
235/235 [==============================] - 28s 119ms/step - loss: 0.9271 - accuracy: 0.6899 - val_lo
ss: 0.8892 - val_accuracy: 0.7027
Epoch 6/40
235/235 [==============================] - 27s 115ms/step - loss: 0.8312 - accuracy: 0.7196 - val_lo
ss: 0.8500 - val_accuracy: 0.7059
Epoch 7/40
235/235 [==============================] - 26s 112ms/step - loss: 0.7793 - accuracy: 0.7352 - val_lo
ss: 0.7748 - val_accuracy: 0.7362
Epoch 8/40
235/235 [==============================] - 29s 123ms/step - loss: 0.7412 - accuracy: 0.7454 - val_lo
ss: 0.7495 - val_accuracy: 0.7347
Epoch 9/40
235/235 [==============================] - 29s 121ms/step - loss: 0.7137 - accuracy: 0.7536 - val_lo
ss: 0.7745 - val_accuracy: 0.7438
Epoch 10/40
235/235 [==============================] - 28s 118ms/step - loss: 0.6924 - accuracy: 0.7615 - val_lo
ss: 0.7102 - val_accuracy: 0.7427
Epoch 11/40
235/235 [==============================] - 27s 116ms/step - loss: 0.6723 - accuracy: 0.7670 - val_lo
ss: 0.7313 - val_accuracy: 0.7369
Epoch 12/40
235/235 [==============================] - 30s 127ms/step - loss: 0.6598 - accuracy: 0.7701 - val_lo
ss: 0.6637 - val_accuracy: 0.7707
```

```
Epoch 13/40
235/235 [==============================] - 28s 120ms/step - loss: 0.6446 - accuracy: 0.7761 - val_lo
ss: 0.6742 - val_accuracy: 0.7451
Epoch 14/40
235/235 [==============================] - 28s 117ms/step - loss: 0.6257 - accuracy: 0.7798 - val_lo
ss: 0.6707 - val_accuracy: 0.7711
Epoch 15/40
235/235 [==============================] - 28s 120ms/step - loss: 0.6200 - accuracy: 0.7822 - val_lo
ss: 0.6251 - val_accuracy: 0.7765
Epoch 16/40
235/235 [==============================] - 31s 133ms/step - loss: 0.6049 - accuracy: 0.7852 - val_lo
ss: 0.6760 - val_accuracy: 0.7524
Epoch 17/40
235/235 [==============================] - 31s 131ms/step - loss: 0.5977 - accuracy: 0.7878 - val_lo
ss: 0.6573 - val_accuracy: 0.7657
Epoch 18/40
235/235 [==============================] - 31s 130ms/step - loss: 0.5873 - accuracy: 0.7914 - val_lo
ss: 0.6131 - val_accuracy: 0.7687
Epoch 19/40
235/235 [==============================] - 30s 127ms/step - loss: 0.5768 - accuracy: 0.7936 - val_lo
ss: 0.6314 - val_accuracy: 0.7666
Epoch 20/40
235/235 [==============================] - 30s 126ms/step - loss: 0.5691 - accuracy: 0.7971 - val_lo
ss: 0.5859 - val_accuracy: 0.7854
Epoch 21/40
235/235 [==============================] - 28s 119ms/step - loss: 0.5624 - accuracy: 0.7976 - val_lo
ss: 0.5710 - val_accuracy: 0.7945
Epoch 22/40
235/235 [==============================] - 27s 115ms/step - loss: 0.5511 - accuracy: 0.8016 - val_lo
ss: 0.6039 - val_accuracy: 0.7616
Epoch 23/40
235/235 [==============================] - 27s 116ms/step - loss: 0.5543 - accuracy: 0.8008 - val_lo
ss: 0.5555 - val_accuracy: 0.7975
Epoch 24/40
235/235 [==============================] - 30s 128ms/step - loss: 0.5367 - accuracy: 0.8056 - val_lo
ss: 0.6211 - val_accuracy: 0.7500
Epoch 25/40
235/235 [==============================] - 32s 137ms/step - loss: 0.5402 - accuracy: 0.8049 - val_lo
ss: 0.6124 - val_accuracy: 0.7892
Epoch 26/40
235/235 [==============================] - 29s 123ms/step - loss: 0.5315 - accuracy: 0.8083 - val_lo
ss: 0.5400 - val_accuracy: 0.8077
Epoch 27/40
235/235 [==============================] - 29s 123ms/step - loss: 0.5248 - accuracy: 0.8110 - val_lo
ss: 0.5353 - val_accuracy: 0.8075
Epoch 28/40
235/235 [==============================] - 29s 122ms/step - loss: 0.5205 - accuracy: 0.8115 - val_lo
ss: 0.5407 - val_accuracy: 0.8046
Epoch 29/40
235/235 [==============================] - 29s 123ms/step - loss: 0.5135 - accuracy: 0.8151 - val_lo
ss: 0.5451 - val_accuracy: 0.7943
Epoch 30/40
235/235 [==============================] - 30s 129ms/step - loss: 0.5103 - accuracy: 0.8165 - val_lo
ss: 0.5949 - val_accuracy: 0.7811
Epoch 31/40
235/235 [==============================] - 28s 120ms/step - loss: 0.5072 - accuracy: 0.8172 - val_lo
ss: 0.5196 - val_accuracy: 0.8135
Epoch 32/40
235/235 [==============================] - 28s 116ms/step - loss: 0.5052 - accuracy: 0.8181 - val_lo
ss: 0.5180 - val_accuracy: 0.8125
Epoch 33/40
235/235 [==============================] - 27s 116ms/step - loss: 0.5003 - accuracy: 0.8206 - val_lo
ss: 0.5197 - val_accuracy: 0.8131
Epoch 34/40
235/235 [==============================] - 27s 115ms/step - loss: 0.4987 - accuracy: 0.8202 - val_lo
ss: 0.5192 - val_accuracy: 0.8124
Epoch 35/40
235/235 [==============================] - 27s 115ms/step - loss: 0.4916 - accuracy: 0.8233 - val_lo
ss: 0.5200 - val_accuracy: 0.8110
Epoch 36/40
235/235 [==============================] - 28s 117ms/step - loss: 0.4919 - accuracy: 0.8230 - val_lo
ss: 0.5388 - val_accuracy: 0.7997
Epoch 37/40
235/235 [==============================] - 30s 125ms/step - loss: 0.4867 - accuracy: 0.8246 - val_lo
ss: 0.5110 - val_accuracy: 0.8152
Epoch 38/40
235/235 [==============================] - 28s 119ms/step - loss: 0.4823 - accuracy: 0.8263 - val_lo
ss: 0.5336 - val_accuracy: 0.8128
Epoch 39/40
235/235 [==============================] - 27s 115ms/step - loss: 0.4799 - accuracy: 0.8265 - val_lo
ss: 0.5899 - val_accuracy: 0.7774
Epoch 40/40
235/235 [==============================] - 27s 116ms/step - loss: 0.4775 - accuracy: 0.8272 - val_lo
```

```
ss: 0.5075 - val_accuracy: 0.8178
```

In [293]:

```
fig1, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 7))

ax1.plot([i for i in range(1, 41)], history_121.history['loss'], color='blue', label='Training')
ax1.plot([i for i in range(1, 41)], history_121.history['val_loss'], color='red', label='Testing')
ax1.legend()
ax1.set_xlabel('number of epochs')
ax1.set_ylabel('cross-entropy loss')
ax1.set_title('Cross-entropy Loss for Tensorflow Model with a Learning Rate of 0.001')

ax2.plot([i for i in range(1, 41)], history_121.history['accuracy'], color='blue', label='Training')
ax2.plot([i for i in range(1, 41)], history_121.history['val_accuracy'], color='red', label='Testing')
ax2.legend()
ax2.set_xlabel('number of epochs')
ax2.set_ylabel('accuracy')
ax2.set_title('Accuracy for Tensorflow Model with a Learning Rate of 0.001')
plt.show()
```



This convolutional neural network with the same learning rate, optimsation technique and activation functions expectedly yields very similar loss and accuracy graphs to the one shown for the `mlp_model_1` neural network: their loss and accuracy plots are almost identical in both value and shape. The slight difference between the two is that the curves for the testing dataset in the above plots are distinctly less smooth than their equivalent in the `mlp_model_1` plot. `model_121` adds more noise to its loss and accuracies to prevent overfitting, as shown by the lack of clear divergence of the testing curve from the training curve in the above plots. This is due to the `MaxPool2D((2, 2))` layer in the convolutional neural network, which is used to extract the most important features of the image.

1.2.2

Now we incorporate Dropou in the fully connected layer of the convolutional neural network. We will attempt to find the optimal dropout probability for our dataset and observe how this affects the overall model performance. Dropout is a technique to help prevent the model overfitting. It acheives this by randomly (with a probability = dropout rate) zeroing certain neurons. This makes the model more generalisable by impeding the neurons from co-adapting and reducing the degree of the model.

In [294]:

```
def model_122_fn(rate):

    return Sequential([
        Conv2D(8, (3, 3), activation = LeakyReLU(alpha=0.01), input_shape = (28, 28, 1), name='conv_layer1'),
        Conv2D(8, (3, 3), activation = LeakyReLU(alpha=0.01), name='conv_layer2'),
        Conv2D(16, (3, 3), activation = LeakyReLU(alpha=0.01), name='conv_layer3'),
        Conv2D(16, (3, 3), activation = LeakyReLU(alpha=0.01), name='conv_layer4'),
        MaxPool2D((2, 2)),
        Flatten(),
        Dense(64, activation = LeakyReLU(alpha=0.01), name='dense_layer'),
        Dropout(rate),
        Dense(10, activation = 'softmax')
        ])
```

In [295]:

```
model_122_fn(0.5).summary()  # Summary of the model for a dropout rate of 0.5.
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv_layer1 (Conv2D)         (None, 26, 26, 8)         80

 conv_layer2 (Conv2D)         (None, 24, 24, 8)         584

 conv_layer3 (Conv2D)         (None, 22, 22, 16)        1168

 conv_layer4 (Conv2D)         (None, 20, 20, 16)        2320

 max_pooling2d_1 (MaxPooling  (None, 10, 10, 16)       0
 2D)

 flatten_1 (Flatten)          (None, 1600)              0

 dense_layer (Dense)          (None, 64)                102464

 dropout (Dropout)            (None, 64)                0

 dense_2 (Dense)              (None, 10)                650

=================================================================
Total params: 107,266
Trainable params: 107,266
Non-trainable params: 0
_________________________________________________________________
```

For this model we will attempt to find the optimal dropout rate by training the model on 80% of the training dataset and using the remaining 20% as a validation set. Using this optimal dropout rate we can then retrain the model on the full training dataset and evaluate its performance.

In [296]:

```
full_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))  # Reload the training dataset.
data_set_size = full_dataset.__len__().numpy()

train_size = int(0.8 * data_set_size)
val_size = int(0.2 * data_set_size)

train_dataset_122 = full_dataset.take(train_size)  # Initialise the 80% training set.
val_dataset_122 = full_dataset.skip(train_size)  # Initialise the 20% validation set.

train_dataset_122 = train_dataset_122.batch(batch_size = 256)  # Batch each the dataset as before.
val_dataset_122 = val_dataset_122.batch(batch_size = 256)
```

We will use stochastic gradient descent and cross-entropy loss as before:

In [297]:

```
sgd = tf.keras.optimizers.SGD(learning_rate = 1e-3)
loss_fn = tf.keras.losses.CategoricalCrossentropy()
```

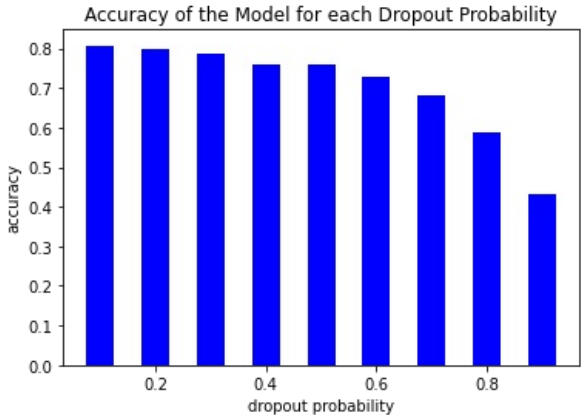
We now scan over a range of dropout probabilities (in steps of 0.1) to find the optimal dropout rate. We will use the maximum accuracy achieved by each dropout rate over all of the epochs it is run for as the criterion for deciding if each rate is optimal.

In [339]:

```
accs = []
dropouts = [rate/10 for rate in range(1, 10)]

for rate in dropouts:
    model_122 = model_122_fn(rate)
    model_122.compile(optimizer = sgd, loss = loss_fn, metrics = ['accuracy'])
    history_122 = model_122.fit(train_dataset_122, validation_data = val_dataset_122, epochs = 40, verbose = Fals
e)

    max_accuracy = max(history_122.history['accuracy'])
    accs.append(max_accuracy)
```

In [340]:

```
plt.bar(dropouts, accs, color='blue', width=0.05)
plt.xlabel('dropout probability')
plt.ylabel('accuracy')
plt.title('Accuracy of the Model for each Dropout Probability')
plt.show()
```



Hence, the optimal dropout rate according to this split of the dataset is:

In [341]:

```
optimal_dropout = accs.index(max(accs))
optimal_dropout = (optimal_dropout + 1) / 10  # Attain the probability from the respective index.
print(optimal_dropout)
```

```
0.1
```

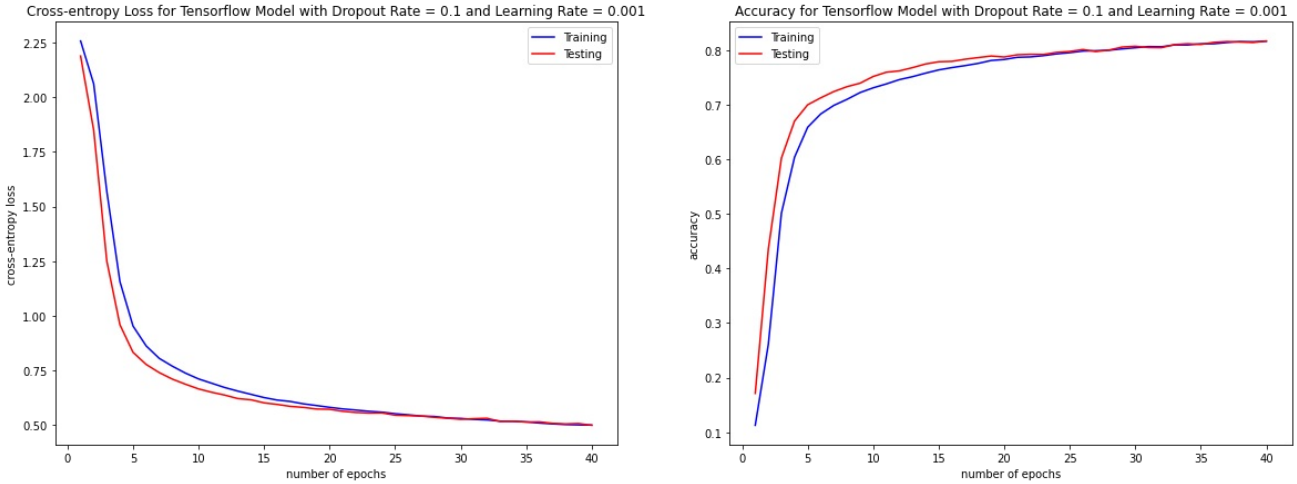
We now compile the model training on the entire dataset with this optimal dropout rate:

In [342]:

```
model_122 = model_122_fn(optimal_dropout)
model_122.compile(optimizer = sgd, loss = loss_fn, metrics = ['accuracy']) # Tracks accuracy, using same sgd and
cross entropy from before
history_122 = model_122.fit(train_dataset, validation_data = test_dataset, epochs = 40, verbose = False)
```

In [503]:

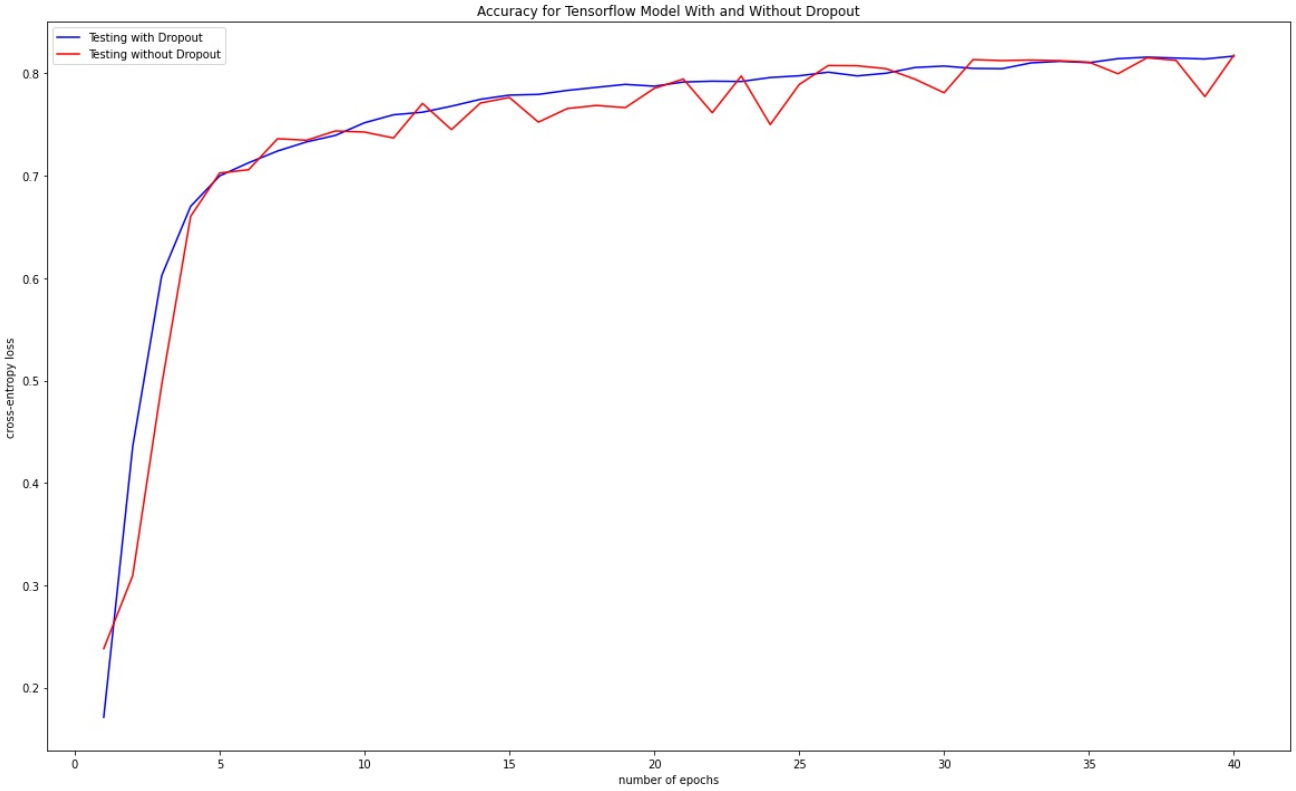
```
fig1, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 7))

ax1.plot([i for i in range(1, 41)], history_122.history['loss'], color='blue', label='Training')
ax1.plot([i for i in range(1, 41)], history_122.history['val_loss'], color='red', label='Testing')
ax1.legend()
ax1.set_xlabel('number of epochs')
ax1.set_ylabel('cross-entropy loss')
ax1.set_title(f'Cross-entropy Loss for Tensorflow Model with Dropout Rate = {optimal_dropout} and Learning Rate = 0.001')

ax2.plot([i for i in range(1, 41)], history_122.history['accuracy'], color='blue', label='Training')
ax2.plot([i for i in range(1, 41)], history_122.history['val_accuracy'], color='red', label='Testing')
ax2.legend()
ax2.set_xlabel('number of epochs')
ax2.set_ylabel('accuracy')
ax2.set_title(f'Accuracy for Tensorflow Model with Dropout Rate = {optimal_dropout} and Learning Rate = 0.001')
plt.show()
```



We can see from the above plots that dropout does improve the training procedure. Focussing on the accuracy plot, this is shown by the distinct lack of divergence from the training data accuracy as we encountered in the multi-layer perceptron. In essence, the dropout regularisation stops the model from overfitting, therefore making it more generalisable. In addition, as can be seen below, dropout makes the training procedure less noisy and the accuracy curve is therefore much smoother whilst maintaining the same level of accuracy as the equivalent model without dropout. The similarity between the training and testing dataset's accuracy in the above plot suggests that our model with dropout is a more effective model: this lack of divergence will translate to a higher accuracy over a larger number of epochs (say, 80), where the model that does not incorperate dropout will begin to overfit and subsequently diverge from the training data accuracy.

In [508]:

```
fig1, ax1 = plt.subplots(1, 1, figsize=(20, 12))

ax1.plot([i for i in range(1, 41)], history_122.history['val_accuracy'], color='blue', label='Testing with Dropou
t')
ax1.plot([i for i in range(1, 41)], history_121.history['val_accuracy'], color='red', label='Testing without Drop
out')
ax1.legend()
ax1.set_xlabel('number of epochs')
ax1.set_ylabel('cross-entropy loss')
ax1.set_title(f'Accuracy for Tensorflow Model With and Without Dropout')
plt.show()
```



We now plot the activations of the hidden units of the fully connected layer when the model is evaluated on the test set. This allows us to compare the features learnt by the fully connected layer in each respective model.

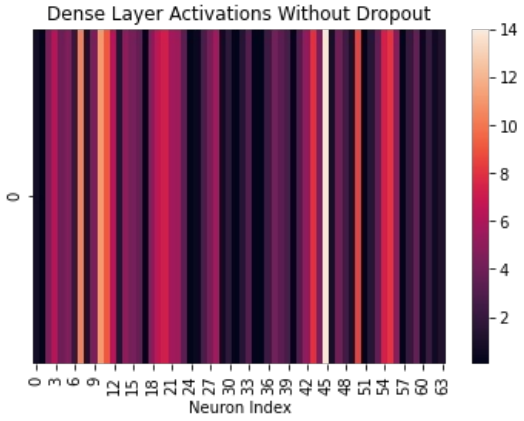
In [565]:

```
model_121_features = Model(inputs = model_121.inputs, outputs = model_121.get_layer('dense').output)
model_122_features = Model(inputs = model_122.inputs, outputs = model_122.get_layer('dense_layer').output)

model_121_activations = np.zeros((1, 64))
model_122_activations = np.zeros((1, 64))

for row in x_test:
        image = row[np.newaxis, :, :, np.newaxis]
        model_121_activations += np.abs(model_121_features(image))
        model_122_activations += np.abs(model_122_features(image))

model_121_activations = model_121_activations/len(x_test)
model_122_activations = model_122_activations/len(x_test)
```

We use `seaborn` 's heatmap to visually plot these activations:

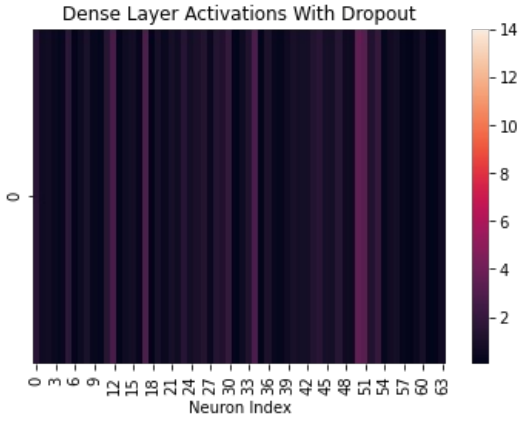
In [575]:

```
sns.heatmap(model_121_activations, vmax = 14)
plt.title('Dense Layer Activations Without Dropout')
plt.xlabel('Neuron Index')
plt.show()
```



In [576]:

```
sns.heatmap(model_122_activations, vmax = 14)
plt.title('Dense Layer Activations With Dropout')
plt.xlabel('Neuron Index')
plt.show()
```



As we can see from the above heatmaps, dropout causes the neurons to have much less variability in their activations and reduces the number of neurons with high activations (although these high activation neurons without dropout still remain the *relatively* higher activations in the model with dropout). This shows that the dropout encourages the network to learn distinct more general features of each image as opposed to the less general and more specific (to the training dataset) features picked up by the model without dropout.

1.2.3

To help compare the multi-layer perceptron model and the convolutional neural network with dropout we print a relevant table:

In [602]:

```
comparison_df = pd.DataFrame(
    data = {'Training Accuracy': [history_121.history['accuracy'][-1], history_122.history['accuracy'][-1]],
            'Testing Accuracy': [history_121.history['val_accuracy'][-1], history_122.history['val_accuracy'][-1]
],
            'Number of Parameters': [959610, 107266]},
    index = ['NumPy Multi-layer Perceptron', 'TensorFlow Convolutional Neural Network with Dropout'])

comparison_df
```

Out[602]:

	Training Accuracy	Testing Accuracy	Number of Parameters
NumPy Multi-layer Perceptron	0.827183	0.8178	959610
TensorFlow Convolutional Neural Network with Dropout	0.816483	0.8169	107266

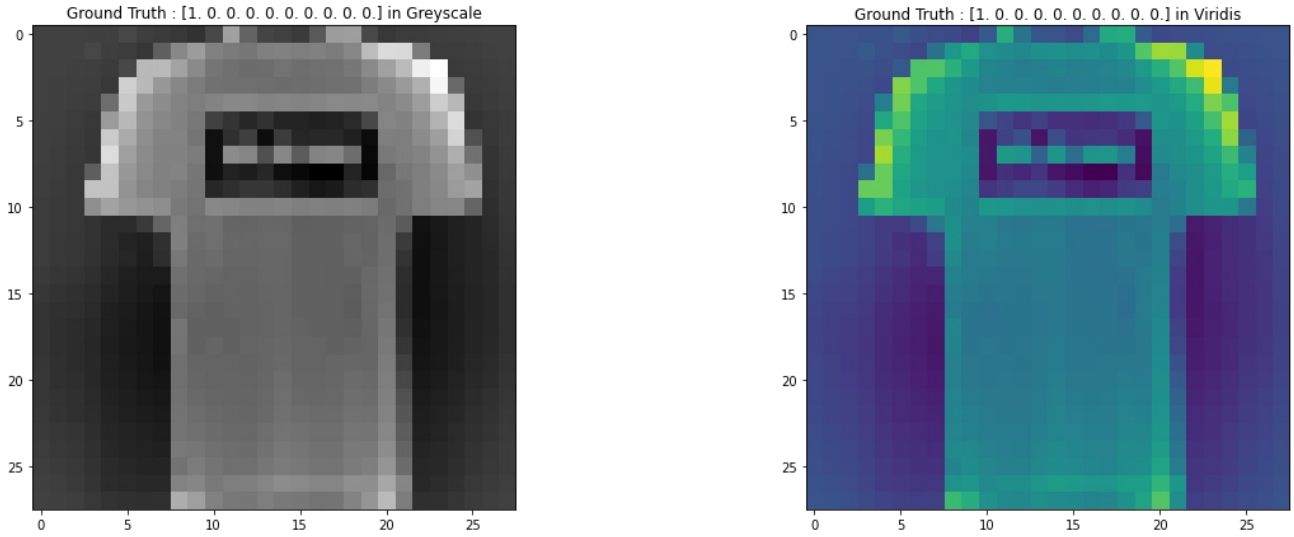
We see that whilst both model's final testing accuracies are similar, there is apparent (and previously identified) overfitting in the multi-layer perceptron model, since its training accuracy is relatively much higher than its testing accuracy. This implies this model may be memorising the training dataset as opposed to *learning* its distinctive features. This memorisation can be partially explained by the number of parameters in each model, where the multi-layer perceptron has almost 10 times the number of parameters. Generally, the more parameters a model has the more likely it is to overfit the data it is trained on, especially when a model does not implement any additional techniques to prevent overfitting. We would expect, for a larger number of epochs, the convolutional neural network to have a significantly higher testing accuracy than the multi-layer perceptron since it does not overfit the data, meaning the training and testing accuracies do not diverge, whilst still maintaining a relatively high testing accuracy.

1.2.4

We now randomly select an image from the test dataset and visualise one feature map from each convolutional layer in the architecture. This allows us to comment on the features extracted by each of the layers. We plot each greyscale image with the *viridis* colour palette for easier visualisation. We note that brighter $\implies$ more activated:

In [603]:

```
fig1, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 7))
ax1.imshow(x_train[1,:,:], cmap='gray')
ax1.set_title(f'Ground Truth : {y_train[1]} in Greyscale')
ax2.imshow(x_train[1,:,:])
ax2.set_title(f'Ground Truth : {y_train[1]} in Viridis')
plt.show()
```



In [604]:

```
image = tf.convert_to_tensor(x_train[1])
inputs = model_122.inputs
layer_names = ['conv_layer1', 'conv_layer2', 'conv_layer3', 'conv_layer4']
outputs = [model_122.get_layer(layer_name).output for layer_name in layer_names]
model_122_features = Model(inputs=inputs, outputs=outputs)
model_122_features.outputs
```

Out[604]:

```
[<KerasTensor: shape=(None, 26, 26, 8) dtype=float32 (created by layer 'conv_layer1')>,
 <KerasTensor: shape=(None, 24, 24, 8) dtype=float32 (created by layer 'conv_layer2')>,
 <KerasTensor: shape=(None, 22, 22, 16) dtype=float32 (created by layer 'conv_layer3')>,
 <KerasTensor: shape=(None, 20, 20, 16) dtype=float32 (created by layer 'conv_layer4')>]
```

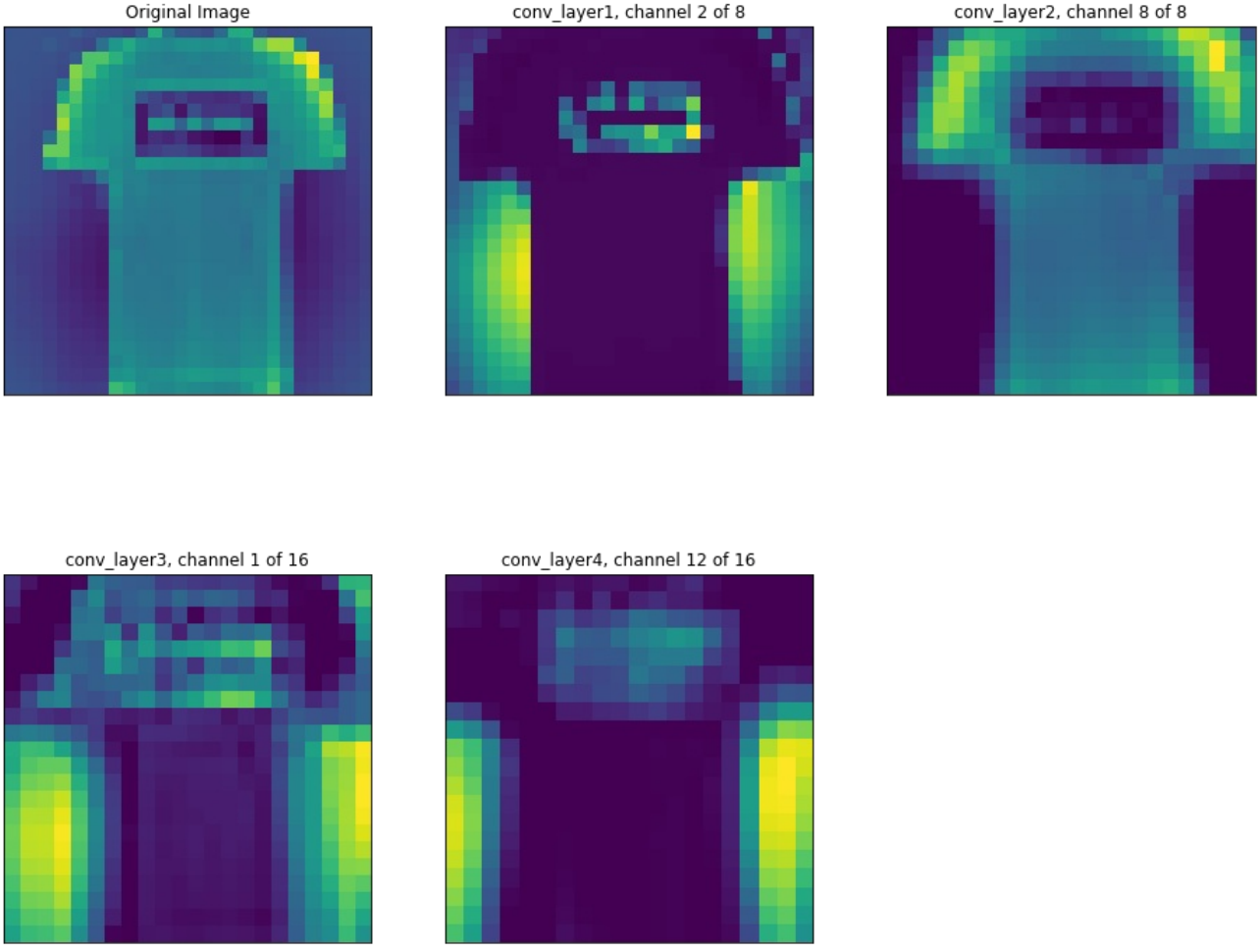
In [605]:

```
image_processed = image * 1.0
features = model_122_features(image_processed[tf.newaxis, ...]) # Extract the hierarchical features for this imag
e.
features = [image] + features
```

Having extracted the image's features at each layer, we can now visualise them:

In [610]:

```
np.random.seed(11)

n_rows, n_cols = 2, 3
fig, axes = plt.subplots(n_rows, n_cols, figsize=(16, 14))
fig.subplots_adjust(hspace=0.05, wspace=0.2)

for i in range(len(features)):
    feature_map = features[i]
    num_channels = feature_map.shape[-1]
    row = i // n_cols
    col = i % n_cols
    if i == 0:
        axes[row, col].imshow(image)
        axes[row, col].set_title('Original Image')
    else:
        random_feature = np.random.choice(num_channels)
        axes[row, col].imshow(feature_map[0, ..., random_feature])
        axes[row, col].set_title('{}, channel {} of {}'.format(layer_names[i-1], random_feature + 1, num_channels
))

    axes[row, col].get_xaxis().set_visible(False)
    axes[row, col].get_yaxis().set_visible(False)

fig.delaxes(axes[1][2])
plt.show()
```



We can see that the original image is clearly a t-shirt, surrounded by un-activated pixels containing some random noise (as not all of the surrounding pixels are completely un-activated). The first convolutional layer performs sharp, straight edge detection on both sides of the body of the t-shirt and also picks up the logo on the chest of the t-shirt. The second layer juxtaposes the first layer and is activated by the t-shirt itself, in particular the curved shoulder edges that were not previously activated. The third and fourth layers once again seem to pick up the pixels not containing the t-shirt either side of its body aswell as the logo on the chest. However, these activated regions are more general and sparse, containing more information about the class and layout of the image as opposed to specific shapes and features picked up in the initial layers.

Task 2: Unsupervised learning ^

2.1 Dimensionality reduction and clustering of a subset of the Fashion-MNIST image data

We now only consider the first N = 1000 images found in the Fashion-MNIST dataset.

In [303]:

```
(x_train_1000, y_train_1000), (x_test_1000, y_test_1000) = load_data()  # Re-initialise the training and testing datasets.
x_train_1000 = x_train_1000.reshape(-1, 28*28)[:1000]  # Reshape the training input data and select the first 1000 entries.
```

We also standardise this subset of the input data.

In [304]:

```
mu_1000 = x_train_1000.mean(axis = 0)
sigma_1000 = x_train_1000.std(axis = 0)
sigma_filled = sigma_1000.copy()
sigma_filled[sigma_1000==0] = 1.  # To prevent division by 0 in the normalisation of the data.

x_train_1000 = (x_train_1000 - mu_1000) / sigma_filled
```

We will initially implement principal component analysis which attempts to reduce the dimensionality of a given dataset whilst maintaining as much information as possible from the original data. This dimension reduction is acheived by ignoring redundant or highly correlated variables, with the aim of making models more effective and computationally feasible. Assuming we are performing principal component analysis on some dataset X for k principal components, the algorithm is as follows:

1. Compute the covariance matrix $C = \frac{X^{\top}X}{n-1}$
2. Find the eigenvalues and corresponding eigenvectors for the covariance matrix, $C = VLV^{\top}$
3. Sort the eigenvalues and corresponding eigenvectors by size.
4. Compute the projection onto the space spanned by the top k eigenvectors.

In [305]:

```
def pca_function(X, k):
    '''
    Return the PCA transformation of X and its eigenvectors and eigenvalues with respect to k.
    '''
    C = 1.0/(len(X)-1) * np.dot(X.T, X)  # Create the covariance matrix C.

    eigenvalues, eigenvectors = linalg.eigsh(C, k, which = "LM", return_eigenvectors = True)  # Compute the eigen values and eigenvectors using the SciPy function eigsh.

    sorted_index = np.argsort(eigenvalues)[::-1]  # Sort the eigenvalues and eigenvectors from largest to smallest eigenvalue.
    eigenvalues = eigenvalues[sorted_index]
    eigenvectors = eigenvectors[:, sorted_index]

    X_pca = X.dot(eigenvectors)  # Transform the data.

    return X_pca, eigenvectors, eigenvalues
```

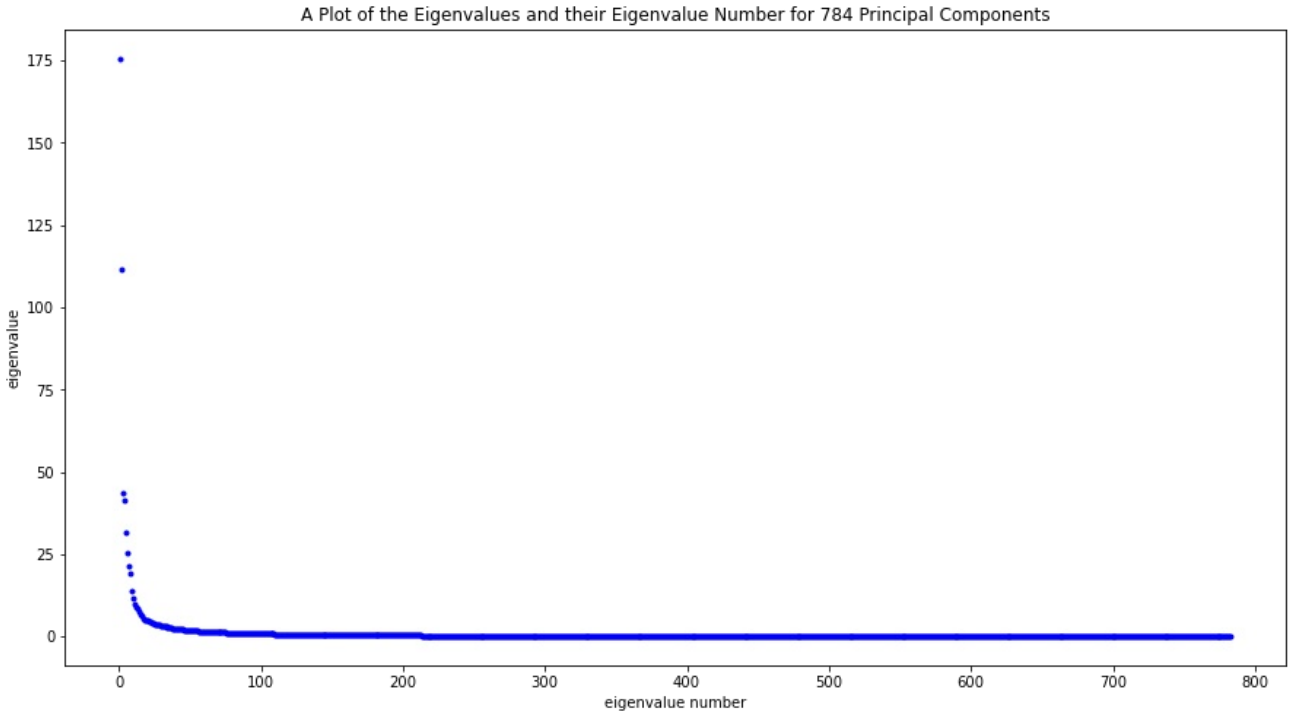
We intially set there to be a maximal 783 principal components, and plot the eigenvalues with respect to their eigenvalue numbers.

In [309]:

```
pca_784 = pca_function(x_train_1000, 783)
```

In [310]:

```
plt.figure(figsize=(15, 8))
plt.plot([i for i in range(1, 784)], pca_784[2], '.', color='blue')
plt.xlabel('eigenvalue number')
plt.ylabel('eigenvalue')
plt.title('A Plot of the Eigenvalues and their Eigenvalue Number for 784 Principal Components')
plt.show()
```



It is clear that the majority of the eigenvalues and therefore principal components are insignificant. We reduce k to visualise these more significant principal components more clearly.

2.1.2

We perform principal component analysis on the top $k = 25$ components and observe the fraction of variance explained as k is increased. Whilst not strictly necessary, we re-initialise our `pca_function` with this reduced k for readibility.

In [311]:

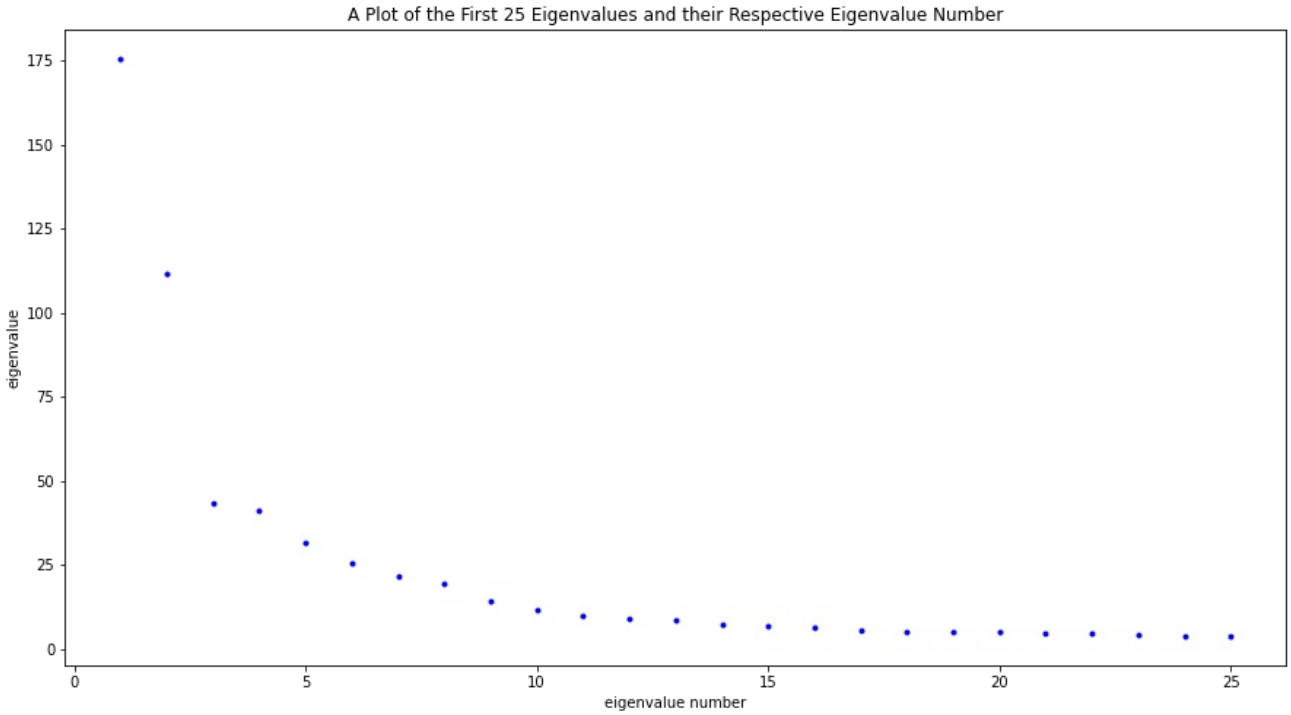
```
pca_25 = pca_function(x_train_1000, 25)
```

In [314]:

```
total_variance = abs(np.sum(pca_784[2]))
explained_variances = pca_784[2] / total_variance
cumsum_explained_variances = np.cumsum(explained_variances)
```

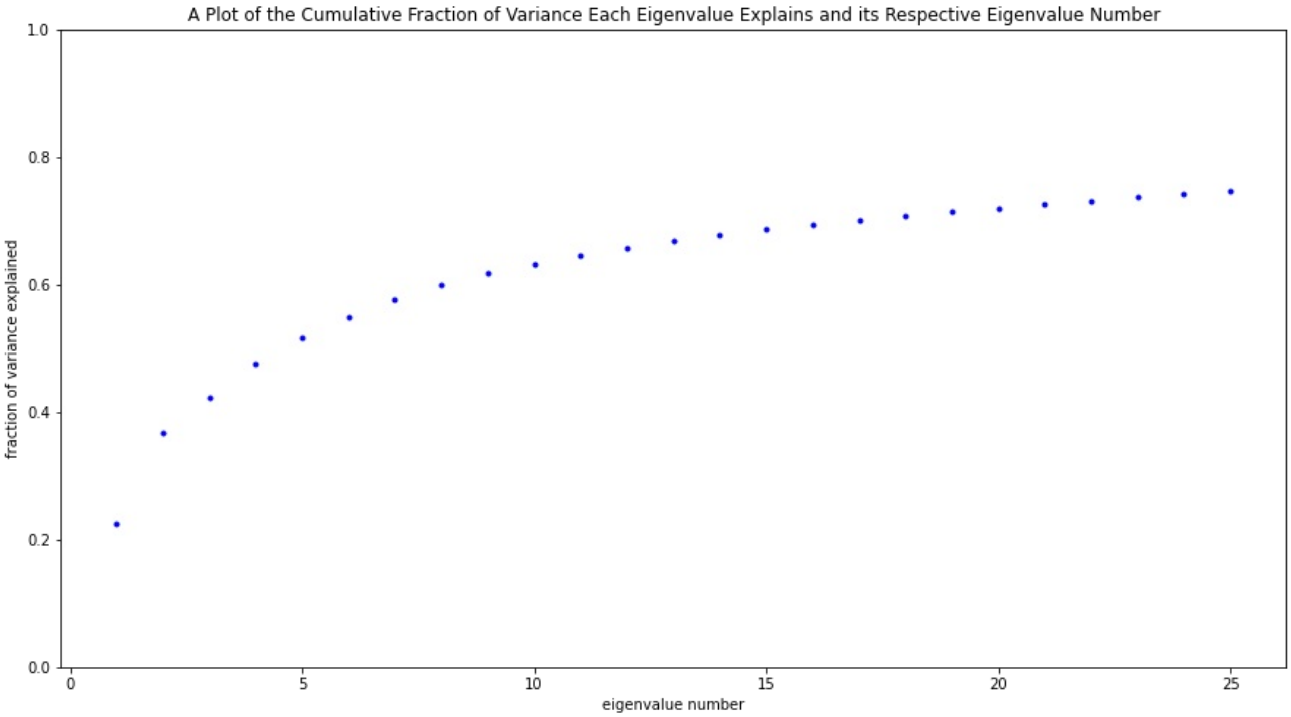
In [315]:

```
plt.figure(figsize=(15, 8))
plt.plot([i for i in range(1, 26)], pca_25[2], '.', color='blue')
plt.xlabel('eigenvalue number')
plt.ylabel('eigenvalue')
plt.title('A Plot of the First 25 Eigenvalues and their Respective Eigenvalue Number')
plt.show()
```



In [316]:

```
plt.figure(figsize=(15, 8))
plt.plot([i for i in range(1, 26)], cumsum_explained_variances[:25], '.', color='blue')
plt.xlabel('eigenvalue number')
plt.ylabel('fraction of variance explained')
plt.ylim(0, 1)
plt.title('A Plot of the Cumulative Fraction of Variance Each Eigenvalue Explains and its Respective Eigenvalue N
umber')
plt.show()
```



We can obtain our optimal value for the reduced dimension k using the visual 'elbow method', which suggests looking for the distinct 'elbow' of our curve. When looking at the plot of each eigenvalue's value, the obvious elbow would be $k = 2$, since the curve flattens out after this point. However, these two principal components only account for roughly 40% of the model's variance. The data is problematic because there are hundreds of principal components (i.e. $25 \leq k \leq 784$) where each fraction of variance is individually insignificant but when considered cumulatively, accounts for over 1/4 of the total variance in the model. Despite this, we consider $k = 2$ to be the optimal value for the reduced dimension according to principal component analysis since this is where the distinct elbow is on our plot and as humans we can easily visualise plots in 2-dimensions, despite the diminishing returns of additional dimensions probably being worth their additional cost.

2.1.3

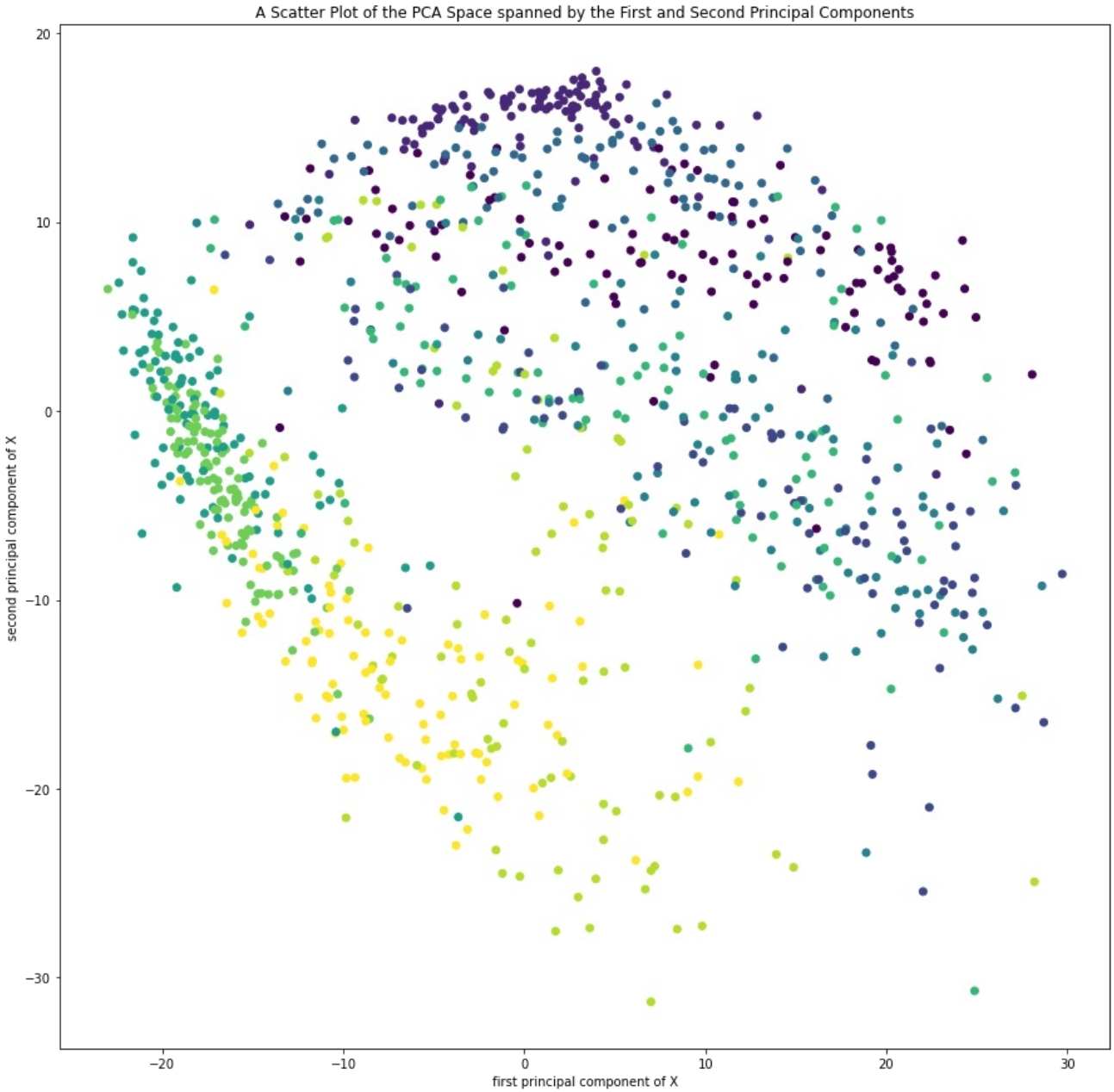
Given our optimal value for the reduced dimension according to principal component analysis, we plot the clustering of our data with respect these principal components. For visualisation purposes, we also colour each point according to its class image.

In [317]:

```
pca_2 = pca_function(x_train_1000, 2)
```

In [323]:

```
plt.figure(figsize=(15, 15))
plt.scatter(pca_2[0][:,0], pca_2[0][:,1], c=np.where(y_train[:1000]==1)[1])
plt.xlabel('first principal component of X')
plt.ylabel('second principal component of X')
plt.title('A Scatter Plot of the PCA Space spanned by the First and Second Principal Components')
plt.show()
```



Whilst the above plot does contain clear clusterings of colours, and therefore classes of images, these clusters are not distinct or seperable. This is due to the fact that the $k = 2$ principal components only account for $\approx$ 40% of the model's variance: each image contains features that are more nuanced than the high level features picked up by the top two dimensions. An example of this is the two shades of green points prevelant around (-20, 0) on the above plot. This similarity in 2-dimensions suggests they share many of the same high level features and we therefore need to extract some more low level features to successfully seperate and identify each respective image.

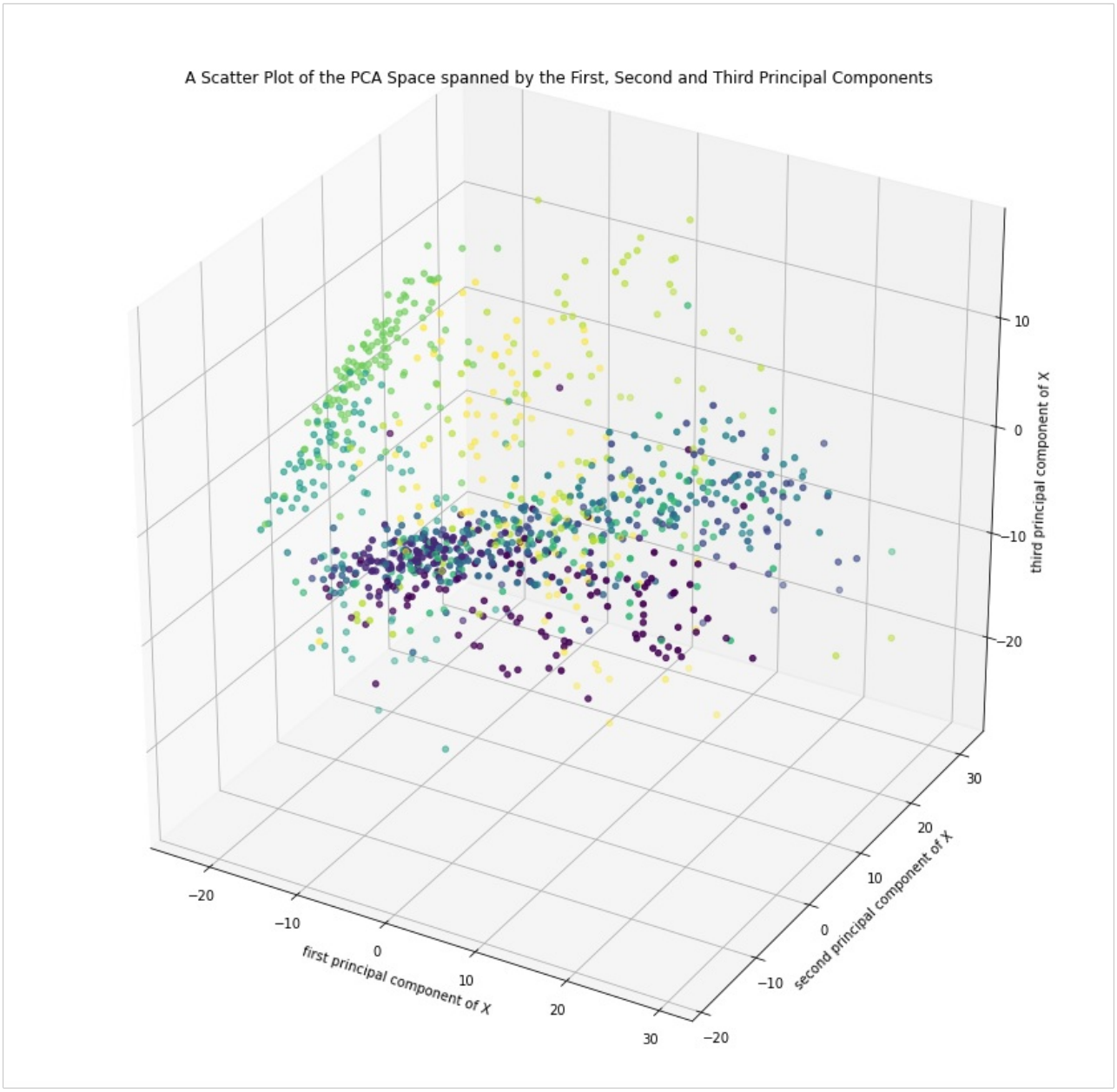
Whilst the third dimension only leads to a $\approx$ 5% increase in explained variance, in plotting this additional dimension we can begin to observe the improvement of clusterings additional dimensions will lead to.

In [325]:

```
pca_3 = pca_function(x_train_1000, 3)
```

In [326]:

```
fig = plt.figure(figsize=(15, 15))
ax = fig.add_subplot(projection='3d')
ax.scatter(pca_3[0][:,0], pca_3[0][:,1], pca_3[0][:,2], c=np.where(y_train[:1000]==1)[1])
ax.set_xlabel('first principal component of X')
ax.set_ylabel('second principal component of X')
ax.set_zlabel('third principal component of X')
ax.set_title('A Scatter Plot of the PCA Space spanned by the First, Second and Third Principal Components')
plt.show()
```



For example, in this 3-dimensional plot we see that the lighter green points (with respect to the two shades of green discussed previously) are generally higher in the third principcal component dimension than the darker green points. Given additional dimensions the seperability of these two classes of data, alongside all of the other classes, would become more obvious. Whilst not easily visualisable, it would seem that a larger number of principal components would be worth their additional cost to improve the seperability of the classes.

2.1.4

We now implement k-means to cluster the data points described by the top two principal components, varying the number of clusters k to obtain the optimal number of clusters. The function we write will also allow us to set the number of random intialisations for the classes we implement, since the initial configuration of the data points can have a notable influence on their final clustering. We will quantify the quality of our clusterings using the within-cluster distance for a balanced dataset, defined as follows:

$$W(C) = \sum_{l=1}^{k} \sum_{i,j \in c_l} \|x^{(i)} - x^{(j)}\|^2,$$

for a given clustering $C = \{c_l\}_{l=1}^{k}$.

It is also important to note that, since we standardised our data:

In [327]:

```
pca_2[0].mean()
```

Out[327]:

```
5.203247e-06
```

Additionally, we note that we are using the within-cluster distance for a *balanced* dataset since our subset of the training data is still relatively well balanced:

In [329]:

```
np.bincount(np.where(y_train[:1000]==1)[1])
```

Out[329]:

```
array([107, 104,  86,  92,  95, 100, 100, 115, 102,  99])
```

Utilising these two attributes of the data, we can now implement our k-means function:

In [ ]:

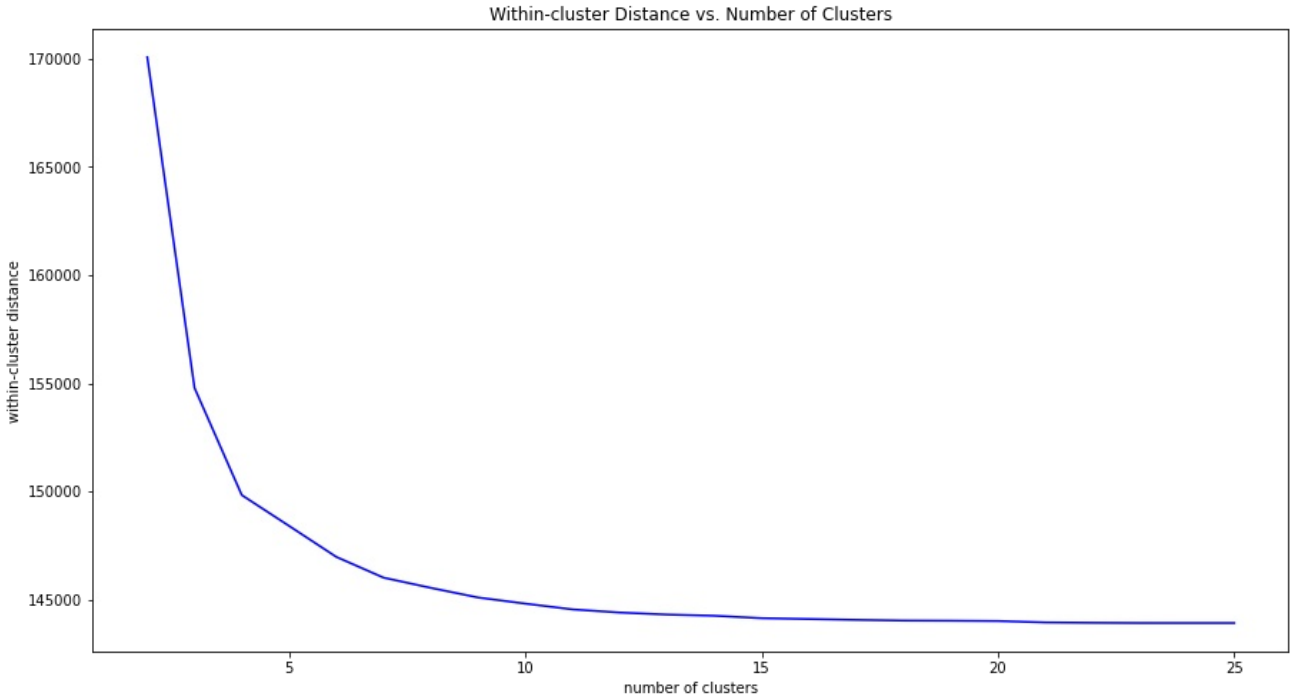
```
def k_means(X, k_min, k_max, num_init):
    '''
    Return the within-cluster distance between each cluster and the labels for which class each data point belong
s to.
    '''
    N, p = X.shape
    within_cluster_distance = np.zeros(k_max - k_min + 1)  # Initialise the average within-cluster distance for e
ach k to zeros.

    for k in range(k_min, k_max + 1):
        initial_labels = np.random.randint(k, size = (num_init, N))  # Randomly assign each sample to one of the
clusters.
        labels = np.zeros_like(initial_labels)

        for initialisation in range(num_init):
            current_labels = initial_labels[initialisation]
            previous_labels = np.zeros_like(current_labels)
            distances = np.zeros((k, N))  # Within-distances to each centroid.
            centroids = np.zeros((k, p))  # Coordinates of each centroid.

            # Stop the process when the labels are no longer changing.
            while np.count_nonzero(current_labels - previous_labels) > 0:
                previous_labels = current_labels  # Update the previous labels.

                for cluster in range(k):
                    cluster_k_indices = [index for index, value in enumerate(current_labels) if value == cluster]
# Take the index of each data point that belongs to the current cluster.

                    if not len(cluster_k_indices):
                        cluster_k_indices = np.random.randint(N)  # If the current cluster is empty assign a rand
om data point to it. Otherwise, we will encounter an error.

                    centroids[cluster] = X[cluster_k_indices].mean()  # Find the centroid of the current cluster.
                    distances[cluster] = np.linalg.norm(X - centroids[cluster], axis = 1)  # Compute the average
distance to the centroid for all data points in the cluster.

                current_labels = np.argmin(distances, axis = 0)  # Reassign each data point to the cluster with t
he nearest centroid and update its label.

            labels[initialisation] = current_labels  # Store an array of labels for each initialisation.

            for cluster in range(k):
                cluster_k_indices = [index for index, label in enumerate(current_labels) if label == cluster]  #
Indices of the in-cluster data points.
                centroids[cluster] = X[cluster_k_indices].mean()  # Centroid of current cluster.

                within_cluster_distance[k - k_min] += (np.linalg.norm(X[cluster_k_indices] - centroids[cluster]))
** 2

    within_cluster_distance = within_cluster_distance / num_init  # Average within-cluster distance over all init
ialisations.
    return within_cluster_distance, labels
```

In [369]:

```
pca2_k_means_25 = k_means(pca_2[0], 2, 25, 100)
```

In [370]:

```
plt.figure(figsize=(15, 8))
plt.plot([i for i in range(2, 26)], pca2_k_means_25[0], color='blue')
plt.xlabel('number of clusters')
plt.ylabel('within-cluster distance')
plt.title('Within-cluster Distance vs. Number of Clusters')
plt.show()
```



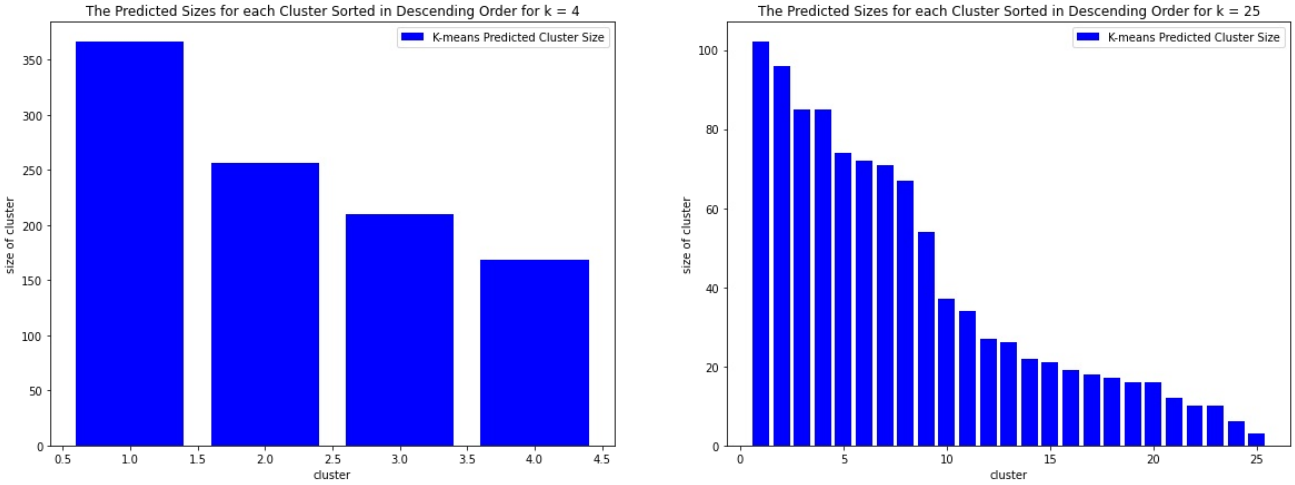
As before, we heuristically select our optimal k according to the elbow method. Whilst possible elbows in the above graph are $k = 4$ or $k = 7$, it seems the optimal elbow is $k = 10$, since after this point the graph is effectively horizontal having been strictly decreasing previously. We can observe this further via the bar plots below. Initially, we plot the predicted cluster sizes in descending order for $k = 4$ (a possible optimal cluster size) and $k = 25$ (clearly to large of a cluster size) as has been shown above.

In [379]:

```
pca2_k_means_4 = k_means(pca_2[0], 2, 4, 100)
pca2_k_means_10 = k_means(pca_2[0], 2, 10, 100)
```

In [388]:

```
fig1, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 7))

ax1.bar([i for i in range(1, 5)], np.sort(np.bincount(pca2_k_means_4[1][-1]))[::-1], color='blue', label = 'K-mea
ns Predicted Cluster Size')
ax1.legend()
ax1.set_xlabel('cluster')
ax1.set_ylabel('size of cluster')
ax1.set_title('The Predicted Sizes for each Cluster Sorted in Descending Order for k = 4')

ax2.bar([i for i in range(1, 26)], np.sort(np.bincount(pca2_k_means_25[1][-1]))[::-1], color='blue', label = 'K-m
eans Predicted Cluster Size')
ax2.legend()
ax2.set_xlabel('cluster')
ax2.set_ylabel('size of cluster')
ax2.set_title('The Predicted Sizes for each Cluster Sorted in Descending Order for k = 25')
plt.show()
```

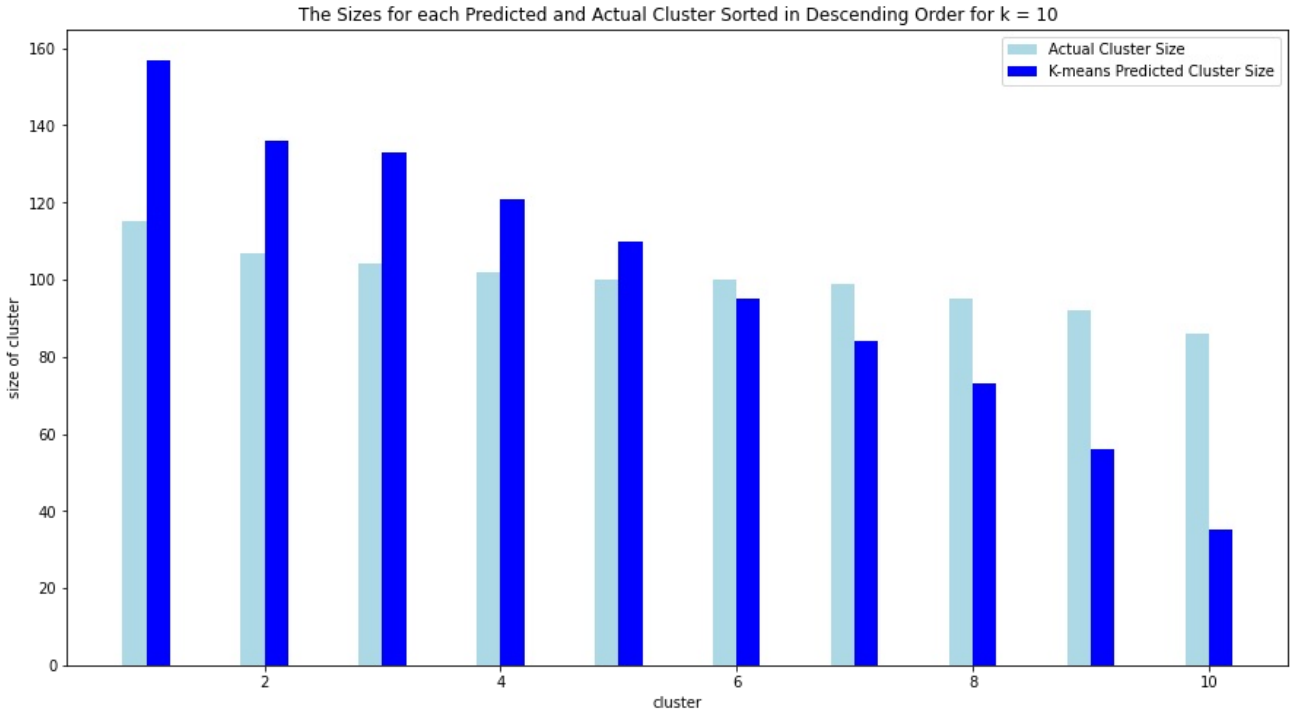


The plot on the left is clearly sub-optimal since the cluster sizes are so un-balanced: we are looking the number of clusters suggests each cluster is roughly the same size, snice the we know the data we were given was a balanced dataset. The plot on the right is similarly sub-optimal due to its descending nature. Interstingly, there is a notable jump in the size of each cluster between cluster 8 and cluster 10. This suggests that the $\approx$ 10 largest clusters are dominant, with the remaining clusters becoming redundant with respect to the overall clustering. This ties in nicely with our optimal $k = 10$ choice, since the graph on the right implies there are 10 main clusterings of points.

We now plot the predicted cluster sizes for $k = 10$ alongside the actual cluster sizes:

In [394]:

```
sorted_actual_cluster_sizes = np.sort(np.bincount(np.where(y_train[:1000]==1)[1]))[::-1]
sorted_predicted_cluster_sizes = np.sort(np.bincount(pca2_k_means_10[1][-1]))[::-1]

plt.figure(figsize=(15, 8))
plt.bar([i-0.1 for i in range(1, 11)], sorted_actual_cluster_sizes, 0.2, label = 'Actual Cluster Size', color='lightblue')
plt.bar([i+0.1 for i in range(1, 11)], sorted_predicted_cluster_sizes, 0.2, label = 'K-means Predicted Cluster Size', color='blue')

plt.xlabel("cluster")
plt.ylabel("size of cluster")
plt.title("The Sizes for each Predicted and Actual Cluster Sorted in Descending Order for k = 10")
plt.legend()
plt.show()
```



Whilst the predicted cluster sizes are still un-balanced, $k = 10$ seems to be a good suggestion for the optimal number of clusters since the bar plot contains no stark jumps and the sizes are relatively well balanced in comparison to $k = 4$ or $k = 25$. Since a dataset of $N = 1000$ is quite small, there is a higher chance for discrepencies than if we had used the entire dataset which is also *perfectly* balanced. This would be an interesting extension to the problem but will not be explored in this notebook.

2.2 Clustering of the feature matrix ^

We now work with a new dataset from a social network of 62 bottlenose dolphins collected in a study of the social behaviour and interactions off the coast of New Zealand. We attempt to study the data in an unsupervised way.

Firstly, we initialise the data and standardise it, since we are going to be working with distances:

In [5]:

```
F = np.load('dolphins_F_62x32(1).npy')  # Set of features characterising each individual.
A = np.load('dolphins_A_62x62(1).npy')  # Social network of associations (this is a binary matrix).
names = pd.read_csv('dolphin_names.csv', index_col = 0).to_numpy()  # Indentifiable names of each indexed dolphin.
```

In [6]:

```
mu = F.mean(axis = 0)
sigma = F.std(axis = 0)

F = (F - mu) / sigma
```

2.2.1

We will now work with the feature matrix `F` and employ heirarchical clustering using average linkage with Euclidean distance. We begin by defining necessary functions to be called on in our `hierarchical_clustering` function.

In [7]:

```
def pairwise_distances(points):
    '''
    Return the Euclidean distance matrix between two data points, such that the element (i, j) is the computed di
stance between the i-th point and j-th point in points, over all attributes.
    '''
    N, D = points.shape
    distance = np.empty((N, N))

    for i in range(N):
        distance[i, i] = 0
        for j in range(i + 1, N):
            d = np.sqrt(np.power(points[i, :] - points[j, :], 2).sum())  # The Euclidean distance norm.
            distance[i, j] = d
            distance[j, i] = d

    return distance
```

In [8]:

```
distances = pairwise_distances(F)  # Initialise the matrix of distances.
```

In [9]:

```
def average_linkage(distances, cluster_assignment, i, j):
    '''
    Return the average distance between the two given clusters, given a numpy array of pair-wise distances and cl
uster assignments for each data point.
    '''
    points_i = np.argwhere(cluster_assignment == i) # Select the data point indices of the first cluster.
    points_j = np.argwhere(cluster_assignment == j) # Select the data point indices of the second cluster.
    pairs = np.array([[element_i.item(), element_j.item()] for element_i in points_i for element_j in points_j])
# Form a cartesian product between the indices in i and indices in j.
    pairs_distance = distances[pairs[:, 0], pairs[:, 1]]  # Select the pair distances between the points in the t
wo clusters from the distances matrix.
    return pairs_distance.mean()  # Return the average.
```

We are now ready to define our `hierarchical_clustering` function, with a specified `linkage`. We will of course use `average_linkage` as our linkage.

In [10]:

```
def hierarchical_clustering(points, distances, linkage):
    """
    Return an array of shape (N, N), where row k represents the points assigned to k unique clusters. The hierarc
hical clustering algorithm start with every point as a single cluster and each iteration merges two clusters into
one.
    """
    N, D = points.shape
    assignments = np.zeros((N, N))
    current_assignment = np.arange(N)  # Begin with every point as its own cluster.
    next_cluster_id = N  # The id to be assigned for the next merged cluster.

    # Begin from level (N - 1) to level 1
    for level in range(N - 1, 0, -1):
        cluster_ids = np.unique(current_assignment)
        min_d = np.inf
        cluster_a, cluster_b = (-1, -1)  # Initialise the cluster ids to be merged in this iteration.

        # Find the two clusters that have the minimum distance in between.
        for i in range(cluster_ids.size):
            for j in range(i + 1, cluster_ids.size):
                cluster_i = cluster_ids[i]
                cluster_j = cluster_ids[j]
                d = linkage(distances, current_assignment, cluster_i, cluster_j)
                if d < min_d:
                    min_d = d
                    cluster_a, cluster_b = (cluster_i, cluster_j)

        current_assignment[(current_assignment == cluster_a) | (current_assignment == cluster_b)] = next_cluster_
id  # Merge the two clusters.

        next_cluster_id += 1
        assignments[level, :] = current_assignment  # Store the current cluster assignment into the assignments a
rray.

    return assignments
```

In [11]:

```
a = hierarchical_clustering(F, distances, average_linkage)
```

We can now report the entire sequence of clusters from finest to coarsest:

In [31]:

```
a.shape
```

Out[31]:

```
(62, 62)
```

In [39]:

```
for i in range(a.shape[0]-1, 0, -1):
    print(f'Number of clusters = {i+1}, cluster assignments = {a[i]}')
```

```
Number of clusters = 62, cluster assignments = [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12.
13. 14. 15. 16. 17.
 18. 19. 20. 21. 62. 23. 24. 25. 26. 27. 28. 29. 30. 62. 32. 33. 34. 35.
 36. 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53.
 54. 55. 56. 57. 58. 59. 60. 61.]
Number of clusters = 61, cluster assignments = [ 0.  1.  2.  3. 63.  5.  6.  7.  8.  9. 10. 63. 12.
13. 14. 15. 16. 17.
 18. 19. 20. 21. 62. 23. 24. 25. 26. 27. 28. 29. 30. 62. 32. 33. 34. 35.
 36. 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53.
 54. 55. 56. 57. 58. 59. 60. 61.]
Number of clusters = 60, cluster assignments = [ 0.  1.  2.  3. 63.  5.  6.  7.  8. 64. 10. 63. 12.
64. 14. 15. 16. 17.
 18. 19. 20. 21. 62. 23. 24. 25. 26. 27. 28. 29. 30. 62. 32. 33. 34. 35.
 36. 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53.
 54. 55. 56. 57. 58. 59. 60. 61.]
Number of clusters = 59, cluster assignments = [ 0.  1.  2.  3. 63. 65. 65.  7.  8. 64. 10. 63. 12.
64. 14. 15. 16. 17.
 18. 19. 20. 21. 62. 23. 24. 25. 26. 27. 28. 29. 30. 62. 32. 33. 34. 35.
 36. 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53.
 54. 55. 56. 57. 58. 59. 60. 61.]
Number of clusters = 58, cluster assignments = [ 0.  1.  2.  3. 63. 65. 65.  7.  8. 66. 10. 63. 12.
66. 14. 15. 16. 17.
 18. 19. 20. 21. 62. 23. 24. 25. 26. 27. 28. 29. 30. 62. 32. 33. 34. 35.
 36. 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53.
 54. 55. 56. 66. 58. 59. 60. 61.]
```

```
Number of clusters = 57, cluster assignments = [ 0.  1.  2.  3. 63. 65. 65.  7.  8. 66. 10. 63. 12.
66. 14. 15. 16. 17.
 18. 19. 20. 21. 62. 23. 67. 25. 26. 27. 28. 29. 30. 62. 32. 33. 34. 35.
 36. 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53.
 54. 67. 56. 66. 58. 59. 60. 61.]
Number of clusters = 56, cluster assignments = [ 0.  1.  2.  3. 63. 65. 65.  7.  8. 66. 10. 63. 12.
66. 14. 15. 16. 17.
 68. 19. 20. 21. 62. 23. 67. 25. 26. 27. 28. 29. 30. 62. 32. 33. 34. 35.
 36. 37. 38. 39. 40. 41. 42. 43. 44. 68. 46. 47. 48. 49. 50. 51. 52. 53.
 54. 67. 56. 66. 58. 59. 60. 61.]
Number of clusters = 55, cluster assignments = [ 0.  1.  2.  3. 63. 65. 65.  7.  8. 66. 10. 63. 12.
66. 14. 15. 16. 17.
 68. 19. 20. 21. 62. 23. 67. 25. 26. 27. 28. 29. 69. 62. 32. 33. 34. 35.
 36. 37. 38. 39. 40. 41. 42. 43. 44. 68. 46. 69. 48. 49. 50. 51. 52. 53.
 54. 67. 56. 66. 58. 59. 60. 61.]
Number of clusters = 54, cluster assignments = [ 0.  1.  2.  3. 70. 65. 65.  7.  8. 66. 10. 70. 12.
66. 14. 15. 16. 17.
 68. 19. 20. 21. 62. 23. 67. 25. 26. 27. 28. 29. 69. 62. 32. 33. 34. 35.
 36. 37. 38. 39. 40. 41. 42. 43. 44. 68. 46. 69. 48. 49. 50. 70. 52. 53.
 54. 67. 56. 66. 58. 59. 60. 61.]
Number of clusters = 53, cluster assignments = [ 0.  1.  2.  3. 70. 65. 65.  7.  8. 66. 10. 70. 12.
66. 14. 15. 16. 17.
 68. 19. 20. 21. 62. 23. 67. 25. 26. 27. 28. 29. 69. 62. 71. 33. 34. 35.
 36. 37. 38. 39. 40. 41. 42. 43. 44. 68. 46. 69. 48. 49. 50. 70. 52. 53.
 54. 67. 56. 66. 58. 59. 71. 61.]
Number of clusters = 52, cluster assignments = [ 0.  1.  2.  3. 70. 72. 72.  7.  8. 72. 10. 70. 12.
72. 14. 15. 16. 17.
 68. 19. 20. 21. 62. 23. 67. 25. 26. 27. 28. 29. 69. 62. 71. 33. 34. 35.
 36. 37. 38. 39. 40. 41. 42. 43. 44. 68. 46. 69. 48. 49. 50. 70. 52. 53.
 54. 67. 56. 72. 58. 59. 71. 61.]
Number of clusters = 51, cluster assignments = [ 0.  1.  2.  3. 70. 72. 72.  7.  8. 72. 10. 70. 12.
72. 14. 15. 16. 17.
 68. 19. 20. 21. 62. 23. 67. 25. 26. 27. 28. 29. 69. 62. 71. 33. 34. 35.
 36. 37. 38. 39. 40. 73. 42. 43. 44. 68. 46. 69. 73. 49. 50. 70. 52. 53.
 54. 67. 56. 72. 58. 59. 71. 61.]
Number of clusters = 50, cluster assignments = [ 0.  1.  2.  3. 70. 72. 72.  7.  8. 72. 10. 70. 12.
72. 14. 15. 16. 74.
 68. 19. 20. 21. 74. 23. 67. 25. 26. 27. 28. 29. 69. 74. 71. 33. 34. 35.
 36. 37. 38. 39. 40. 73. 42. 43. 44. 68. 46. 69. 73. 49. 50. 70. 52. 53.
 54. 67. 56. 72. 58. 59. 71. 61.]
Number of clusters = 49, cluster assignments = [ 0.  1.  2.  3. 70. 72. 72. 75.  8. 72. 10. 70. 12.
72. 14. 15. 16. 74.
 68. 75. 20. 21. 74. 23. 67. 25. 26. 27. 28. 29. 69. 74. 71. 33. 34. 35.
 36. 37. 38. 39. 40. 73. 42. 43. 44. 68. 46. 69. 73. 49. 50. 70. 52. 53.
 54. 67. 56. 72. 58. 59. 71. 61.]
Number of clusters = 48, cluster assignments = [ 0.  1.  2.  3. 70. 72. 72. 75.  8. 72. 10. 70. 12.
72. 14. 15. 16. 74.
 68. 75. 20. 21. 74. 23. 67. 25. 26. 27. 28. 29. 69. 74. 71. 33. 34. 35.
 36. 37. 76. 39. 40. 73. 42. 43. 44. 68. 46. 69. 73. 49. 50. 70. 52. 53.
 54. 67. 56. 72. 76. 59. 71. 61.]
Number of clusters = 47, cluster assignments = [ 0.  1.  2.  3. 70. 72. 72. 75.  8. 72. 10. 70. 12.
72. 14. 15. 16. 74.
 68. 75. 20. 21. 74. 23. 67. 25. 26. 27. 28. 29. 69. 74. 71. 33. 34. 35.
 36. 37. 76. 39. 40. 73. 42. 43. 44. 68. 77. 69. 73. 77. 50. 70. 52. 53.
 54. 67. 56. 72. 76. 59. 71. 61.]
Number of clusters = 46, cluster assignments = [ 0.  1.  2.  3. 70. 72. 72. 75.  8. 72. 10. 70. 12.
72. 14. 15. 16. 74.
 68. 75. 20. 21. 74. 23. 67. 78. 26. 78. 28. 29. 69. 74. 71. 33. 34. 35.
 36. 37. 76. 39. 40. 73. 42. 43. 44. 68. 77. 69. 73. 77. 50. 70. 52. 53.
 54. 67. 56. 72. 76. 59. 71. 61.]
Number of clusters = 45, cluster assignments = [ 0.  1.  2.  3. 70. 72. 72. 75.  8. 72. 79. 70. 12.
72. 14. 15. 16. 74.
 68. 75. 20. 21. 74. 23. 67. 78. 26. 78. 28. 29. 69. 74. 71. 33. 34. 35.
 36. 37. 76. 39. 40. 73. 79. 43. 44. 68. 77. 69. 73. 77. 50. 70. 52. 53.
 54. 67. 56. 72. 76. 59. 71. 61.]
Number of clusters = 44, cluster assignments = [ 0.  1.  2.  3. 70. 72. 72. 75.  8. 72. 79. 70. 12.
72. 14. 15. 16. 74.
 68. 75. 20. 21. 74. 23. 67. 80. 80. 80. 28. 29. 69. 74. 71. 33. 34. 35.
 36. 37. 76. 39. 40. 73. 79. 43. 44. 68. 77. 69. 73. 77. 50. 70. 52. 53.
 54. 67. 56. 72. 76. 59. 71. 61.]
Number of clusters = 43, cluster assignments = [ 0.  1.  2.  3. 70. 81. 81. 75.  8. 81. 79. 70. 12.
81. 14. 15. 16. 74.
 68. 75. 20. 21. 74. 23. 67. 80. 80. 80. 28. 29. 69. 74. 71. 33. 34. 35.
 36. 37. 76. 39. 40. 81. 79. 43. 44. 68. 77. 69. 81. 77. 50. 70. 52. 53.
 54. 67. 56. 81. 76. 59. 71. 61.]
Number of clusters = 42, cluster assignments = [ 0.  1.  2.  3. 70. 81. 81. 75.  8. 81. 79. 70. 12.
81. 82. 15. 16. 74.
 68. 75. 20. 21. 74. 23. 67. 80. 80. 80. 28. 29. 69. 74. 71. 82. 34. 35.
 36. 37. 76. 39. 40. 81. 79. 43. 44. 68. 77. 69. 81. 77. 50. 70. 52. 53.
 54. 67. 56. 81. 76. 59. 71. 61.]
Number of clusters = 41, cluster assignments = [ 0.  1.  2.  3. 83. 81. 81. 75.  8. 81. 79. 83. 12.
81. 82. 15. 16. 74.
 68. 75. 20. 21. 74. 23. 83. 80. 80. 80. 28. 29. 69. 74. 71. 82. 34. 35.
```

```
 36. 37. 76. 39. 40. 81. 79. 43. 44. 68. 77. 69. 81. 77. 50. 83. 52. 53.
 54. 83. 56. 81. 76. 59. 71. 61.]
Number of clusters = 40, cluster assignments = [ 0.  1.  2.  3. 83. 81. 81. 75.  8. 81. 79. 83. 12.
81. 84. 15. 84. 74.
 68. 75. 20. 21. 74. 23. 83. 80. 80. 80. 28. 29. 69. 74. 71. 84. 34. 35.
 36. 37. 76. 39. 40. 81. 79. 43. 44. 68. 77. 69. 81. 77. 50. 83. 52. 53.
 54. 83. 56. 81. 76. 59. 71. 61.]
Number of clusters = 39, cluster assignments = [ 0.  1.  2.  3. 83. 81. 81. 75.  8. 81. 79. 83. 12.
81. 84. 15. 84. 74.
 68. 75. 20. 21. 74. 23. 83. 80. 80. 80. 28. 85. 69. 74. 71. 84. 34. 85.
 36. 37. 76. 39. 40. 81. 79. 43. 44. 68. 77. 69. 81. 77. 50. 83. 52. 53.
 54. 83. 56. 81. 76. 59. 71. 61.]
Number of clusters = 38, cluster assignments = [ 0.  1.  2.  3. 83. 81. 81. 75.  8. 81. 79. 83. 12.
81. 84. 15. 84. 74.
 86. 75. 20. 86. 74. 23. 83. 80. 80. 80. 28. 85. 69. 74. 71. 84. 34. 85.
 36. 37. 76. 39. 40. 81. 79. 43. 44. 86. 77. 69. 81. 77. 50. 83. 52. 53.
 54. 83. 56. 81. 76. 59. 71. 61.]
Number of clusters = 37, cluster assignments = [ 0.  1.  2.  3. 83. 81. 81. 75.  8. 81. 79. 83. 12.
81. 84. 15. 84. 74.
 86. 75. 20. 86. 74. 23. 83. 80. 80. 80. 28. 85. 69. 74. 71. 84. 34. 85.
 87. 37. 76. 87. 40. 81. 79. 43. 44. 86. 77. 69. 81. 77. 50. 83. 52. 53.
 54. 83. 56. 81. 76. 59. 71. 61.]
Number of clusters = 36, cluster assignments = [ 0.  1.  2.  3. 88. 81. 81. 75.  8. 81. 79. 88. 12.
81. 84. 15. 84. 74.
 88. 75. 20. 88. 74. 23. 88. 80. 80. 80. 28. 85. 69. 74. 71. 84. 34. 85.
 87. 37. 76. 87. 40. 81. 79. 43. 44. 88. 77. 69. 81. 77. 50. 88. 52. 53.
 54. 88. 56. 81. 76. 59. 71. 61.]
Number of clusters = 35, cluster assignments = [ 0.  1.  2.  3. 88. 81. 81. 75.  8. 81. 89. 88. 12.
81. 84. 15. 84. 74.
 88. 75. 20. 88. 74. 23. 88. 80. 80. 80. 28. 85. 89. 74. 71. 84. 34. 85.
 87. 37. 76. 87. 40. 81. 89. 43. 44. 88. 77. 89. 81. 77. 50. 88. 52. 53.
 54. 88. 56. 81. 76. 59. 71. 61.]
Number of clusters = 34, cluster assignments = [ 0.  1.  2.  3. 88. 90. 90. 75.  8. 90. 89. 88. 12.
90. 84. 15. 84. 74.
 88. 75. 20. 88. 74. 23. 88. 80. 80. 80. 28. 85. 89. 74. 71. 84. 34. 85.
 87. 37. 76. 87. 40. 90. 89. 43. 44. 88. 77. 89. 90. 77. 50. 88. 52. 53.
 54. 88. 90. 90. 76. 59. 71. 61.]
Number of clusters = 33, cluster assignments = [ 0.  1.  2.  3. 88. 90. 90. 91.  8. 90. 89. 88. 12.
90. 84. 15. 84. 74.
 88. 91. 20. 88. 74. 23. 88. 80. 80. 80. 28. 85. 89. 74. 71. 84. 34. 85.
 87. 37. 76. 87. 40. 90. 89. 43. 44. 88. 77. 89. 90. 77. 50. 88. 52. 53.
 91. 88. 90. 90. 76. 59. 71. 61.]
Number of clusters = 32, cluster assignments = [ 0.  1.  2.  3. 88. 90. 90. 91.  8. 90. 89. 88. 12.
90. 84. 15. 84. 74.
 88. 91. 20. 88. 74. 23. 88. 80. 80. 80. 28. 85. 89. 74. 71. 84. 34. 85.
 87. 37. 92. 87. 40. 90. 89. 43. 44. 88. 77. 89. 90. 77. 50. 88. 92. 53.
 91. 88. 90. 90. 92. 59. 71. 61.]
Number of clusters = 31, cluster assignments = [ 0.  1.  2.  3. 88. 90. 90. 91.  8. 90. 89. 88. 12.
90. 93. 15. 93. 74.
 88. 91. 20. 88. 74. 23. 88. 80. 80. 80. 28. 85. 89. 74. 71. 93. 34. 85.
 87. 37. 92. 87. 93. 90. 89. 43. 44. 88. 77. 89. 90. 77. 50. 88. 92. 53.
 91. 88. 90. 90. 92. 59. 71. 61.]
Number of clusters = 30, cluster assignments = [ 0.  1.  2. 94. 88. 90. 90. 91.  8. 90. 89. 88. 12.
90. 93. 15. 93. 74.
 88. 91. 20. 88. 74. 23. 88. 80. 80. 80. 28. 85. 89. 74. 71. 93. 34. 85.
 87. 37. 92. 87. 93. 90. 89. 43. 44. 88. 77. 89. 90. 77. 50. 88. 92. 53.
 91. 88. 90. 90. 92. 94. 71. 61.]
Number of clusters = 29, cluster assignments = [ 0.  1.  2. 94. 88. 90. 90. 91.  8. 90. 89. 88. 12.
90. 93. 15. 93. 95.
 88. 91. 20. 88. 95. 23. 88. 95. 95. 95. 28. 85. 89. 95. 71. 93. 34. 85.
 87. 37. 92. 87. 93. 90. 89. 43. 44. 88. 77. 89. 90. 77. 50. 88. 92. 53.
 91. 88. 90. 90. 92. 94. 71. 61.]
Number of clusters = 28, cluster assignments = [ 0.  1.  2. 94. 88. 90. 90. 91.  8. 90. 89. 88. 12.
90. 93. 15. 93. 95.
 88. 91. 20. 88. 95. 23. 88. 95. 95. 95. 28. 85. 89. 95. 71. 93. 34. 85.
 87. 37. 92. 87. 93. 90. 89. 43. 44. 88. 77. 89. 90. 77. 50. 88. 92. 96.
 91. 88. 90. 90. 92. 94. 71. 96.]
Number of clusters = 27, cluster assignments = [ 0.  1.  2. 94. 88. 90. 90. 91.  8. 90. 89. 88. 97.
90. 93. 15. 93. 95.
 88. 91. 20. 88. 95. 23. 88. 95. 95. 95. 28. 85. 89. 95. 71. 93. 34. 85.
 87. 37. 92. 87. 93. 90. 89. 43. 44. 88. 77. 89. 90. 77. 97. 88. 92. 96.
 91. 88. 90. 90. 92. 94. 71. 96.]
Number of clusters = 26, cluster assignments = [ 0. 98.  2. 94. 88. 90. 90. 98.  8. 90. 89. 88. 97.
90. 93. 15. 93. 95.
 88. 98. 20. 88. 95. 23. 88. 95. 95. 95. 28. 85. 89. 95. 71. 93. 34. 85.
 87. 37. 92. 87. 93. 90. 89. 43. 44. 88. 77. 89. 90. 77. 97. 88. 92. 96.
 98. 88. 90. 90. 92. 94. 71. 96.]
Number of clusters = 25, cluster assignments = [ 0. 98.  2. 94. 88. 90. 90. 98.  8. 90. 89. 88. 97.
90. 93. 15. 93. 95.
 88. 98. 20. 88. 95. 99. 88. 95. 95. 95. 28. 85. 89. 95. 71. 93. 34. 85.
 99. 37. 92. 99. 93. 90. 89. 43. 44. 88. 77. 89. 90. 77. 97. 88. 92. 96.
 98. 88. 90. 90. 92. 94. 71. 96.]
Number of clusters = 24, cluster assignments = [  0.  98.   2.  94.  88.  90.  90.  98.   8.  90.  8
```

```
9.  88.  97.  90.
  93.  15.  93.  95.  88.  98.  20.  88.  95.  99.  88.  95.  95.  95.
  28.  85.  89.  95.  71.  93. 100.  85.  99.  37.  92.  99.  93.  90.
  89.  43.  44.  88. 100.  89.  90. 100.  97.  88.  92.  96.  98.  88.
  90.  90.  92.  94.  71.  96.]
Number of clusters = 23, cluster assignments = [  0.  98.   2.  94.  88.  90.  90.  98.   8.  90.  8
9.  88.  97.  90.
 101.  15. 101.  95.  88.  98.  20.  88.  95.  99.  88.  95.  95.  95.
  28.  85.  89.  95.  71. 101. 100.  85.  99. 101.  92.  99. 101.  90.
  89.  43.  44.  88. 100.  89.  90. 100.  97.  88.  92.  96.  98.  88.
  90.  90.  92.  94.  71.  96.]
Number of clusters = 22, cluster assignments = [  0.  98.   2. 102.  88.  90.  90.  98. 102.  90.  8
9.  88.  97.  90.
 101.  15. 101.  95.  88.  98.  20.  88.  95.  99.  88.  95.  95.  95.
  28.  85.  89.  95.  71. 101. 100.  85.  99. 101.  92.  99. 101.  90.
  89.  43.  44.  88. 100.  89.  90. 100.  97.  88.  92.  96.  98.  88.
  90.  90.  92. 102.  71.  96.]
Number of clusters = 21, cluster assignments = [  0.  98.   2. 102.  88.  90.  90.  98. 102.  90.  8
9.  88.  97.  90.
 101.  15. 101.  95.  88.  98. 103.  88.  95.  99.  88.  95.  95.  95.
 103.  85.  89.  95.  71. 101. 100.  85.  99. 101.  92.  99. 101.  90.
  89.  43.  44.  88. 100.  89.  90. 100.  97.  88.  92.  96.  98.  88.
  90.  90.  92. 102.  71.  96.]
Number of clusters = 20, cluster assignments = [  0.  98.   2. 102.  88.  90.  90.  98. 102.  90.  8
9.  88.  97.  90.
 101.  15. 101.  95.  88.  98. 103.  88.  95.  99.  88.  95.  95.  95.
 103.  85.  89.  95.  71. 101. 100.  85.  99. 101. 104.  99. 101.  90.
  89. 104.  44.  88. 100.  89.  90. 100.  97.  88. 104.  96.  98.  88.
  90.  90. 104. 102.  71.  96.]
Number of clusters = 19, cluster assignments = [  0.  98.   2. 102.  88.  90.  90.  98. 102.  90.  8
9.  88. 105.  90.
 105.  15. 105.  95.  88.  98. 103.  88.  95.  99.  88.  95.  95.  95.
 103.  85.  89.  95.  71. 105. 100.  85.  99. 105. 104.  99. 105.  90.
  89. 104.  44.  88. 100.  89.  90. 100. 105.  88. 104.  96.  98.  88.
  90.  90. 104. 102.  71.  96.]
Number of clusters = 18, cluster assignments = [  0.  98.   2. 102. 106.  90.  90.  98. 102.  90.  8
9. 106. 105.  90.
 105. 106. 105.  95. 106.  98. 103. 106.  95.  99. 106.  95.  95.  95.
 103.  85.  89.  95.  71. 105. 100.  85.  99. 105. 104.  99. 105.  90.
  89. 104.  44. 106. 100.  89.  90. 100. 105. 106. 104.  96.  98. 106.
  90.  90. 104. 102.  71.  96.]
Number of clusters = 17, cluster assignments = [107.  98.   2. 102. 106.  90.  90.  98. 102.  90. 10
7. 106. 105.  90.
 105. 106. 105.  95. 106.  98. 103. 106.  95.  99. 106.  95.  95.  95.
 103.  85. 107.  95.  71. 105. 100.  85.  99. 105. 104.  99. 105.  90.
 107. 104.  44. 106. 100. 107.  90. 100. 105. 106. 104.  96.  98. 106.
  90.  90. 104. 102.  71.  96.]
Number of clusters = 16, cluster assignments = [107.  98.   2. 102. 106.  90.  90.  98. 102.  90. 10
7. 106. 108.  90.
 108. 106. 108.  95. 106.  98. 103. 106.  95.  99. 106.  95.  95.  95.
 103.  85. 107.  95.  71. 108. 100.  85.  99. 108. 108.  99. 108.  90.
 107. 108.  44. 106. 100. 107.  90. 100. 108. 106. 108.  96.  98. 106.
  90.  90. 108. 102.  71.  96.]
Number of clusters = 15, cluster assignments = [107.  98.   2. 102. 109.  90.  90.  98. 102.  90. 10
7. 109. 108.  90.
 108. 109. 108.  95. 109.  98. 103. 109.  95.  99. 109.  95.  95.  95.
 103. 109. 107.  95.  71. 108. 100. 109.  99. 108. 108.  99. 108.  90.
 107. 108.  44. 109. 100. 107.  90. 100. 108. 109. 108.  96.  98. 109.
  90.  90. 108. 102.  71.  96.]
Number of clusters = 14, cluster assignments = [107.  98. 110. 102. 109.  90.  90.  98. 102.  90. 10
7. 109. 108.  90.
 108. 109. 108.  95. 109.  98. 103. 109.  95.  99. 109.  95.  95.  95.
 103. 109. 107.  95.  71. 108. 100. 109.  99. 108. 108.  99. 108.  90.
 107. 108.  44. 109. 100. 107.  90. 100. 108. 109. 108. 110.  98. 109.
  90.  90. 108. 102.  71. 110.]
Number of clusters = 13, cluster assignments = [107.  98. 110. 102. 109. 111. 111.  98. 102. 111. 10
7. 109. 108. 111.
 108. 109. 108.  95. 109.  98. 103. 109.  95.  99. 109.  95.  95.  95.
 103. 109. 107.  95. 111. 108. 100. 109.  99. 108. 108.  99. 108. 111.
 107. 108.  44. 109. 100. 107. 111. 100. 108. 109. 108. 110.  98. 109.
 111. 111. 108. 102. 111. 110.]
Number of clusters = 12, cluster assignments = [107. 112. 110. 102. 109. 111. 111. 112. 102. 111. 10
7. 109. 108. 111.
 108. 109. 108. 112. 109. 112. 103. 109. 112.  99. 109. 112. 112. 112.
 103. 109. 107. 112. 111. 108. 100. 109.  99. 108. 108.  99. 108. 111.
 107. 108.  44. 109. 100. 107. 111. 100. 108. 109. 108. 110. 112. 109.
 111. 111. 108. 102. 111. 110.]
Number of clusters = 11, cluster assignments = [107. 112. 110. 102. 109. 111. 111. 112. 102. 111. 10
7. 109. 113. 111.
 113. 109. 113. 112. 109. 112. 103. 109. 112.  99. 109. 112. 112. 112.
 103. 109. 107. 112. 111. 113. 100. 109.  99. 113. 113.  99. 113. 111.
 107. 113. 113. 109. 100. 107. 111. 100. 113. 109. 113. 110. 112. 109.
 111. 111. 113. 102. 111. 110.]
```

```
Number of clusters = 10, cluster assignments = [107. 112. 110. 114. 109. 111. 111. 112. 114. 111. 10
7. 109. 113. 111.
 113. 109. 113. 112. 109. 112. 114. 109. 112.  99. 109. 112. 112. 112.
 114. 109. 107. 112. 111. 113. 100. 109.  99. 113. 113.  99. 113. 111.
 107. 113. 113. 109. 100. 107. 111. 100. 113. 109. 113. 110. 112. 109.
 111. 111. 113. 114. 111. 110.]
Number of clusters = 9, cluster assignments = [107. 112. 110. 115. 109. 111. 111. 112. 115. 111. 107
. 109. 113. 111.
 113. 109. 113. 112. 109. 112. 115. 109. 112. 115. 109. 112. 112. 112.
 115. 109. 107. 112. 111. 113. 100. 109. 115. 113. 113. 115. 113. 111.
 107. 113. 113. 109. 100. 107. 111. 100. 113. 109. 113. 110. 112. 109.
 111. 111. 113. 115. 111. 110.]
Number of clusters = 8, cluster assignments = [107. 112. 110. 115. 109. 111. 111. 112. 115. 111. 107
. 109. 116. 111.
 116. 109. 116. 112. 109. 112. 115. 109. 112. 115. 109. 112. 112. 112.
 115. 109. 107. 112. 111. 116. 116. 109. 115. 116. 116. 115. 116. 111.
 107. 116. 116. 109. 116. 107. 111. 116. 116. 109. 116. 110. 112. 109.
 111. 111. 116. 115. 111. 110.]
Number of clusters = 7, cluster assignments = [107. 117. 110. 115. 109. 117. 117. 117. 115. 117. 107
. 109. 116. 117.
 116. 109. 116. 117. 109. 117. 115. 109. 117. 115. 109. 117. 117. 117.
 115. 109. 107. 117. 117. 116. 116. 109. 115. 116. 116. 115. 116. 117.
 107. 116. 116. 109. 116. 107. 117. 116. 116. 109. 116. 110. 117. 109.
 117. 117. 116. 115. 117. 110.]
Number of clusters = 6, cluster assignments = [107. 117. 110. 118. 118. 117. 117. 117. 118. 117. 107
. 118. 116. 117.
 116. 118. 116. 117. 118. 117. 118. 118. 117. 118. 118. 117. 117. 117.
 118. 118. 107. 117. 117. 116. 116. 118. 118. 116. 116. 118. 116. 117.
 107. 116. 116. 118. 116. 107. 117. 116. 116. 118. 116. 110. 117. 118.
 117. 117. 116. 118. 117. 110.]
Number of clusters = 5, cluster assignments = [119. 117. 119. 118. 118. 117. 117. 117. 118. 117. 119
. 118. 116. 117.
 116. 118. 116. 117. 118. 117. 118. 118. 117. 118. 118. 117. 117. 117.
 118. 118. 119. 117. 117. 116. 116. 118. 118. 116. 116. 118. 116. 117.
 119. 116. 116. 118. 116. 119. 117. 116. 116. 118. 116. 119. 117. 118.
 117. 117. 116. 118. 117. 119.]
Number of clusters = 4, cluster assignments = [119. 117. 119. 120. 120. 117. 117. 117. 120. 117. 119
. 120. 120. 117.
 120. 120. 120. 117. 120. 117. 120. 120. 117. 120. 120. 117. 117. 117.
 120. 120. 119. 117. 117. 120. 120. 120. 120. 120. 120. 120. 120. 117.
 119. 120. 120. 120. 120. 119. 117. 120. 120. 120. 120. 119. 117. 120.
 117. 117. 120. 120. 117. 119.]
Number of clusters = 3, cluster assignments = [121. 117. 121. 121. 121. 117. 117. 117. 121. 117. 121
. 121. 121. 117.
 121. 121. 121. 117. 121. 117. 121. 121. 117. 121. 121. 117. 117. 117.
 121. 121. 121. 117. 117. 121. 121. 121. 121. 121. 121. 121. 121. 117.
 121. 121. 121. 121. 121. 121. 117. 121. 121. 121. 121. 121. 117. 121.
 117. 117. 121. 121. 117. 121.]
Number of clusters = 2, cluster assignments = [122. 122. 122. 122. 122. 122. 122. 122. 122. 122. 122
. 122. 122. 122.
 122. 122. 122. 122. 122. 122. 122. 122. 122. 122. 122. 122. 122. 122.
 122. 122. 122. 122. 122. 122. 122. 122. 122. 122. 122. 122. 122. 122.
 122. 122. 122. 122. 122. 122. 122. 122. 122. 122. 122. 122. 122. 122.
 122. 122. 122. 122. 122. 122.]
```

Of course, data points (represented by their respective column in each row of `a`) with the same cluster assignment in `a` are said to be in the same cluster, despite the numbers not being sequential from 1. This is simply due to our construction of `heirarchical_clustering`.

A more insightful way of displaying the information contained in `a` is to observe the relative sizes of each cluster. We display this for the first 20 clusters below, since beyond this the clusters become too small (i.e. contain 1 or 2 data points) to offer any useful information:

In [427]:

```
print(f'For {1} cluster the respective cluster size is: {sorted(list(np.bincount(a[1].astype(int))[np.nonzero(np.
bincount(a[1].astype(int)))]), reverse=True)}')
for i in range(2, 21):
    print(f'For {i} clusters the respective cluster sizes are: {sorted(list(np.bincount(a[i].astype(int))[np.nonz
ero(np.bincount(a[i].astype(int)))]), reverse=True)}')
```

```
For 1 cluster the respective cluster size is: [62]
For 2 clusters the respective cluster sizes are: [42, 20]
For 3 clusters the respective cluster sizes are: [34, 20, 8]
For 4 clusters the respective cluster sizes are: [20, 19, 15, 8]
For 5 clusters the respective cluster sizes are: [20, 19, 15, 5, 3]
For 6 clusters the respective cluster sizes are: [20, 15, 11, 8, 5, 3]
For 7 clusters the respective cluster sizes are: [15, 11, 10, 10, 8, 5, 3]
For 8 clusters the respective cluster sizes are: [12, 11, 10, 10, 8, 5, 3, 3]
For 9 clusters the respective cluster sizes are: [12, 11, 10, 10, 5, 5, 3, 3, 3]
For 10 clusters the respective cluster sizes are: [12, 11, 10, 10, 5, 3, 3, 3, 3, 2]
For 11 clusters the respective cluster sizes are: [11, 11, 10, 10, 5, 3, 3, 3, 3, 2, 1]
For 12 clusters the respective cluster sizes are: [11, 11, 10, 6, 5, 4, 3, 3, 3, 3, 2, 1]
For 13 clusters the respective cluster sizes are: [11, 11, 8, 6, 5, 4, 3, 3, 3, 3, 2, 2, 1]
For 14 clusters the respective cluster sizes are: [11, 11, 8, 6, 5, 4, 3, 3, 3, 2, 2, 2, 1, 1]
For 15 clusters the respective cluster sizes are: [11, 9, 8, 6, 5, 4, 3, 3, 3, 2, 2, 2, 2, 1, 1]
For 16 clusters the respective cluster sizes are: [9, 8, 7, 6, 5, 4, 4, 3, 3, 3, 2, 2, 2, 2, 1, 1]
For 17 clusters the respective cluster sizes are: [9, 8, 7, 6, 4, 4, 4, 3, 3, 3, 2, 2, 2, 2, 1, 1, 1
]
For 18 clusters the respective cluster sizes are: [8, 8, 7, 6, 4, 4, 4, 3, 3, 3, 2, 2, 2, 2, 1, 1, 1
, 1]
For 19 clusters the respective cluster sizes are: [8, 8, 6, 5, 4, 4, 4, 3, 3, 3, 2, 2, 2, 2, 2, 1, 1
, 1, 1]
For 20 clusters the respective cluster sizes are: [8, 8, 6, 5, 4, 4, 3, 3, 3, 3, 2, 2, 2, 2, 2, 1, 1
, 1, 1, 1]
```

We can see that the cluster size begins to peter out after the $\approx$ 6 largest clusters (i.e. the smaller clusters are not as significant, since they only contain between 1 and 3 elements in each cluster). This would suggest that our data is clustered in to $\approx$ 6 dominant features, with the additional features being more nuanced.

2.2.2

We will now explore the silhouette Score for each clustering to help determine the optimal number of clusters for F . The Silhouette Score is a measure of the quality of clustering and is defined as the mean Silhouette Coefficient over all samples, where the Silhouette Coefficient of a given sample is given by:

$$\text{Silhouette Coefficient} = \frac{(d_2 - d_1)}{max(d_1, d_2)}$$

where d_1 is the average distance between the sample and the points in its cluster, and d_2 is the average distance between the sample and the points in the nearest cluster. We will initialise the silhouette coeficient for a singleton cluster to be 0. We begin by defining a function for find the average distance between a data point and a specified cluster:

In [428]:

```
def average_distance(point_index, distances, cluster_assignment, cluster_id):
    '''
    Return the average distance from a data point with index: point_index to the elements in cluster: cluster_id.
    '''
    cluster_indices = np.argwhere(cluster_assignment == cluster_id)  # Select the indices of the data points in c
luster_id.

    pairs = np.array([[cluster_index.item(), point_index] for cluster_index in cluster_indices])  # Form a cartes
ian product between the cluster indices and the point_index.
    pairs_distance = distances[pairs[:, 0], pairs[:,1]]  # Select the pair distances between the points in the cl
uster and the data point from the distances matrix.

    return pairs_distance.mean()  # Return the average distance.
```

Using the `average_distance` function, we can now define a function to find the silhouette coeficient:

In [438]:

```
def silhouette_coefficient(point_index, distances, cluster_assignment):
    '''
    Return the silhouette coefficient for a data point with index: point_index.
    '''
    point_cluster_id = cluster_assignment[point_index]  # The cluster assignment for the point with index: point_
index.
    previous_distance = np.inf
    closest_cluster_id = None
    unique_clusters = np.unique(cluster_assignment)

    if list(cluster_assignment).count(point_cluster_id) == 1:
        return 0.  # Return zero for a singleton cluster.

    for cluster_id in unique_clusters:
        if cluster_id != point_cluster_id:
            new_distance = average_distance(point_index, distances, cluster_assignment, cluster_id)
            if new_distance < previous_distance:
                closest_cluster_id = cluster_id

    d1 = average_distance(point_index, distances, cluster_assignment, point_cluster_id)
    d2 = average_distance(point_index, distances, cluster_assignment, closest_cluster_id)

    return (d2 - d1) / max(d1, d2)
```

And subsequently a function to find the silhouette score for a given clustering:

In [453]:

```
def silhouette_score(distances, cluster_assignment):
    '''
    Return the silhouette score, using the silhouette coeficients.
    '''
    silhouette_coefs = [silhouette_coefficient(point_index, distances, cluster_assignment) for point_index in ran
ge(distances.shape[0])]
    return np.mean(silhouette_coefs)
```

In [454]:

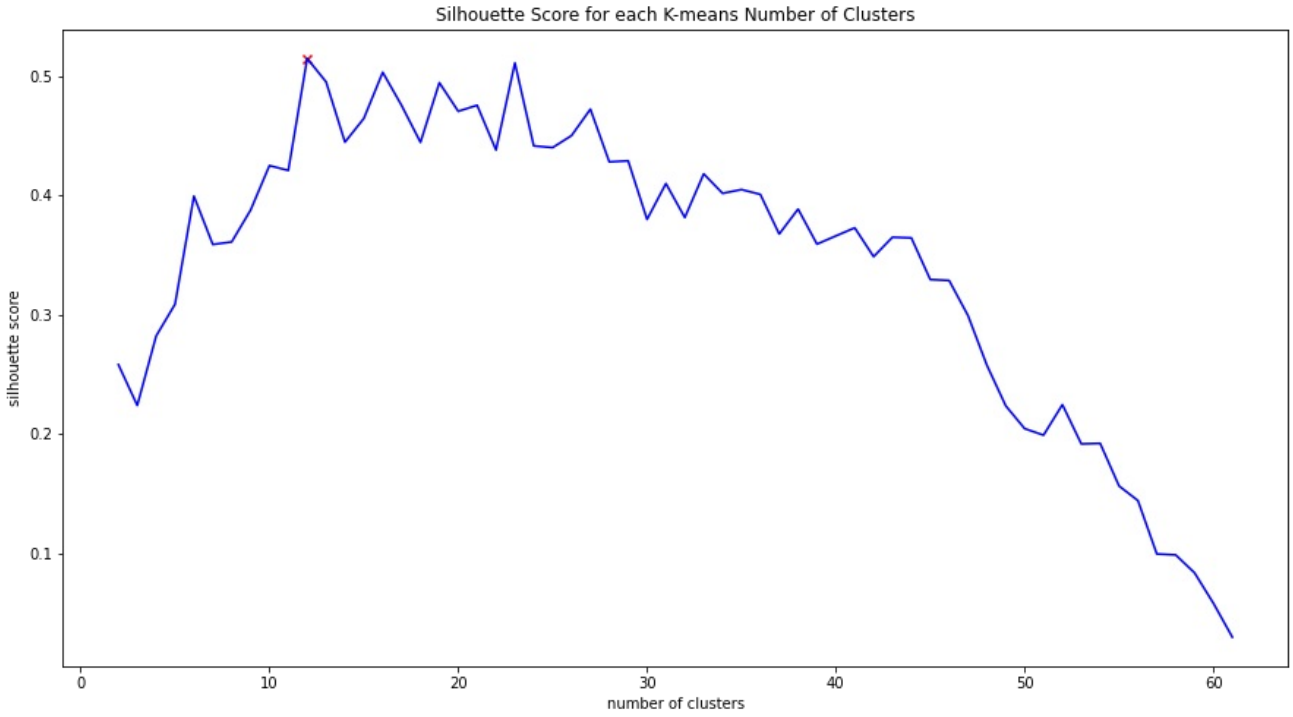
```
silhouette_scores = [silhouette_score(distances, a[i]) for i in range(2, 62)]  # Initialise the silhouette scores
.
```

In [455]:

```
optimal_number_of_clusters, optimal_silouette_score = (silhouette_scores.index(max(silhouette_scores)) + 2, max(s
ilhouette_scores))
```

In [456]:

```
plt.figure(figsize=(15, 8))
plt.plot([i for i in range(2, 62)], silhouette_scores, color='blue')
plt.scatter(optimal_number_of_clusters, optimal_silouette_score, color='red', marker='x')
plt.xlabel('number of clusters')
plt.ylabel('silhouette score')
plt.title('Silhouette Score for each K-means Number of Clusters')
plt.show()
print(f'The optimal number of clusters, according to the silhouette score, is {optimal_number_of_clusters} with a score of {optimal_silouette_score}')
```




```
The optimal number of clusters, according to the silhouette score, is 12 with a score of 0.514590115
1235512
```

Whilst this does not exactly agree with our suggestion of $\approx$ 6 key clusterings, it does reinforce the fact that F contains many dolphins with similar features and can be clustered optimally into a relatively small number of clusters.

2.3 Graph-based analysis ^

In this final section we will analyse the dolphin social network encoded by the adjacency matrix A. This matrix represents the frequent associations between each pair of dolphins.

2.3.1

Firstly, we obtain a spectral partition of the graph into two sets of nodes, which can be found via the normalised laplacian matrix and its two smallest eigenvalues (and corresponding eigenvectors).

The symmetric normalized Laplacian matrix can be constructed as follows:

$$\boldsymbol{L} = \boldsymbol{I} - \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$$

where D is the diagonal degree matrix for the graph.

In [458]:

```
D_inv_sqrt = np.diag(1.0 / np.sqrt(A.sum(axis=1)))  # Initialise the inverse square root of the degree matrix.
L_norm = np.eye(A.shape[0]) - D_inv_sqrt.dot(A.dot(D_inv_sqrt))  # Define the normalised Laplacian matrix.
```

In [459]:

```
eigenvalues, eigenvectors = linalg.eigsh(L_norm, 2, which="SM", return_eigenvectors=True)  # Return the 2 smalles
t eigenvalues and corresponding eigenvectors.
```

In [462]:

```
print('The first eigenalue is: {}'.format(eigenvalues[0]))
print('The second eigenalue is: {}'.format(eigenvalues[1]))
print('With corresponding eigenvectors (represented by each vertical column):')
print(eigenvectors)
```

```
The first eigenalue is: 2.8544163289522667e-16
The second eigenalue is: 0.039524553757434584
With corresponding eigenvectors (represented by each vertical column):
[[ 0.13736056  0.07851822]
 [ 0.15861032 -0.16036544]
 [ 0.11215443  0.07295835]
 [ 0.09712859  0.0618522 ]
 [ 0.05607722  0.04448139]
 [ 0.11215443 -0.20911166]
 [ 0.13736056 -0.24277991]
 [ 0.12539247 -0.08127465]
 [ 0.13736056  0.07633475]
 [ 0.14836637 -0.27090905]
 [ 0.12539247  0.07404428]
 [ 0.05607722  0.04448139]
 [ 0.05607722  0.04272592]
 [ 0.15861032 -0.2828202 ]
 [ 0.19425717  0.13802011]
 [ 0.14836637  0.10061878]
 [ 0.13736056  0.09725818]
 [ 0.16823165 -0.27600033]
 [ 0.14836637  0.10975751]
 [ 0.11215443 -0.09208262]
 [ 0.16823165  0.09191852]
 [ 0.13736056  0.10422357]
 [ 0.05607722 -0.09578601]
 [ 0.09712859  0.05468197]
 [ 0.13736056  0.10396969]
 [ 0.09712859 -0.14133741]
 [ 0.09712859 -0.12558405]
 [ 0.12539247 -0.15791498]
 [ 0.12539247  0.0091502 ]
 [ 0.16823165  0.12810224]
 [ 0.12539247 -0.01420465]
 [ 0.05607722 -0.09578601]
 [ 0.09712859 -0.19048394]
 [ 0.17733173  0.129771  ]
 [ 0.12539247  0.09502782]
 [ 0.05607722  0.04445793]
 [ 0.14836637  0.02227752]
 [ 0.18598708  0.12415257]
 [ 0.15861032  0.11623065]
 [ 0.07930516 -0.06247532]
 [ 0.15861032  0.0734026 ]
 [ 0.12539247 -0.20014729]
 [ 0.13736056  0.06586726]
 [ 0.14836637  0.11530721]
 [ 0.11215443  0.07845602]
 [ 0.18598708  0.13208491]
 [ 0.07930516  0.06635459]
 [ 0.13736056  0.05119187]
 [ 0.05607722 -0.09712005]
 [ 0.07930516  0.06582962]
 [ 0.14836637  0.1025759 ]
 [ 0.17733173  0.13510289]
 [ 0.11215443  0.07787247]
 [ 0.07930516  0.06220904]
 [ 0.14836637 -0.20501613]
 [ 0.07930516  0.05945121]
 [ 0.07930516 -0.14994309]
 [ 0.16823165 -0.27984426]
 [ 0.05607722  0.04278479]
 [ 0.12539247  0.07130897]
 [ 0.05607722 -0.11450158]
 [ 0.09712859  0.07087134]]
```

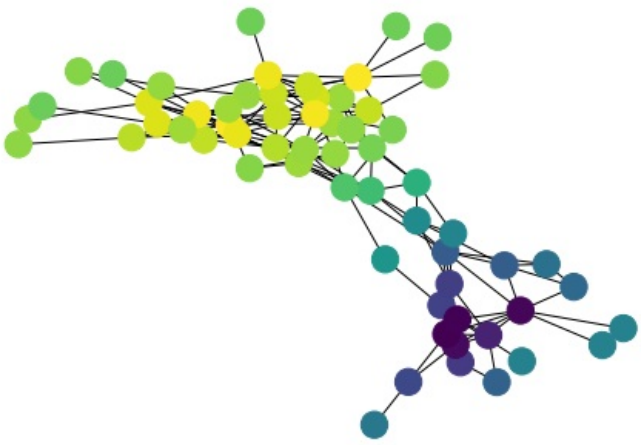
There are two things to note here:

1. The smallest eigenvalue is 0, as expected and required.
2. The second smallest eigenvalue is very small which implies there exists a good bipartition of the graph given by v_2, the corresponding *Fiedler* eigenvector. This eigenvalue is called the *algabraic connectivity* of the graph.

We can observe this bipartition by coloring each node according to its corresponding value in v_2:

In [465]:

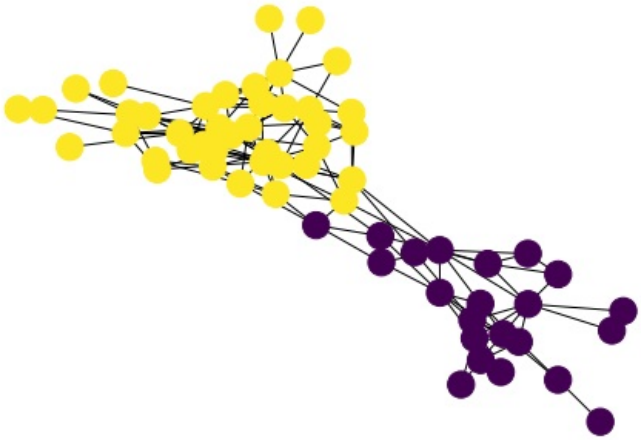
```
g = nx.Graph(A)
nx.draw(g, node_color=eigenvectors[:,1])
```



It does not make sense to list each node's cluster affiliation, but we can binarise v_2 to obtain a more visual plot. This binarisation is done according to the sign of the value in v_2 corresponding to each node.

In [469]:

```
spectral_partition = eigenvectors[:,1]
spectral_partition[spectral_partition<0] = 0
spectral_partition[spectral_partition>0] = 1

nx.draw(g, node_color=spectral_partition)  # Draw the binarised spectral partition.
```



Visually, this graph confirms that we have indeed obtained a good spectral partition of the graph.

2.3.2

We now analyse the network according to three measures of centrality: PageRank, Degree centrality and Eigenvector centrality. We will use the measures to study which nodes are highly central and why different centrality measures lead to different centrality rankings. We begin by defining functions for each of the three measures of centrality:

In [470]:

```
def page_rank(A):
    '''
    Return the PageRank centrality for each node.
    '''
    alpha = 0.85  # Conventionally the teleportation parameter is set to 0.85.
    D = np.diag(A.sum(axis = 1))
    I = np.identity(A.shape[0])

    X = I - alpha * A @ np.linalg.inv(D)
    b = (1 - alpha) / A.shape[0] * np.ones(A.shape[0])

    return np.linalg.solve(X, b)  # Solve Xc = b for c.
```

In [471]:

```
def degree_centrality(A):
    '''
    Return the degree centrality for each node.
    '''
    d = A.sum(axis = 1)  # Degree vector.
    return A.sum(axis=1) / d.sum()  # Degree / (total network degree) for each node.
```

In [472]:

```
def eigenvector_centrality(A):
    '''
    Return the eigenvector centrality for each node. The eigenvector centrality of each node is given by its corr
esponding component in the leading eigenvector of the adjacency matrix.
    '''
    return np.ndarray.flatten(np.abs(linalg.eigsh(A, 1)[1])) # Leading eigenvector of A. We take the absolute val
ue so the vector is always positive.
```

We report the above centralities for each dolphin in the following table:

In [483]:

```
data1 = {'Page Rank Centrality': list(page_rank(A)), 'Degree Centrality': list(degree_centrality(A)), 'Eigenvecto
r Centrality': list(eigenvector_centrality(A))}
df1 = pd.DataFrame(data, index=list(names.flatten()))
pd.set_option('display.max_rows', 1000)
df1
```

Out[483]:

	Page Rank	Degree	Eigenvector
Beak	0.016965	0.018868	0.128504
Beescratch	0.024651	0.025157	0.042076
Bumper	0.013338	0.012579	0.039757
CCL	0.009629	0.009434	0.079335
Cross	0.005080	0.003145	0.029287
DN16	0.014428	0.012579	0.006559
DN21	0.020054	0.018868	0.012191
DN63	0.015643	0.015723	0.042901
Double	0.017098	0.018868	0.143102
Feather	0.023458	0.022013	0.012197
Fish	0.015108	0.015723	0.075253
Five	0.005080	0.003145	0.029287
Fork	0.004835	0.003145	0.039076
Gallatin	0.026157	0.025157	0.015005
Grin	0.032144	0.037736	0.315783
Haecksel	0.019883	0.022013	0.164176
Hook	0.016627	0.018868	0.207994
Jet	0.031728	0.028302	0.017512
Jonah	0.019396	0.022013	0.202495
Knit	0.012928	0.012579	0.020675
Kringel	0.024641	0.028302	0.184477
MN105	0.016939	0.018868	0.207351
MN23	0.005416	0.003145	0.002434
MN60	0.009863	0.009434	0.087362
MN83	0.016906	0.018868	0.193214
Mus	0.011504	0.009434	0.005946
Notch	0.011210	0.009434	0.008944
Number1	0.017130	0.015723	0.016317
Oscar	0.014846	0.015723	0.068224
Patchback	0.026459	0.028302	0.211763
PL	0.015302	0.015723	0.040747
Quasi	0.005416	0.003145	0.002434
Ripplefluke	0.013209	0.009434	0.003856

Ripplefluke	0.013309	0.009434	0.003856
Scabs	0.028423	0.031447	0.281099
Shmuddel	0.015920	0.015723	0.138828
SMN5	0.004918	0.003145	0.029438
SN100	0.020613	0.022013	0.132762
SN4	0.029875	0.034591	0.300562
SN63	0.023939	0.025157	0.196618
SN89	0.007765	0.006289	0.020871
SN9	0.021966	0.025157	0.207872
SN90	0.016138	0.015723	0.015244
SN96	0.017619	0.018868	0.080950
Stripes	0.021691	0.022013	0.190339
Thumper	0.012831	0.012579	0.077802
Topless	0.029514	0.034591	0.285005
TR120	0.008826	0.006289	0.029717
TR77	0.017340	0.018868	0.080371
TR82	0.005262	0.003145	0.002415
TR88	0.008877	0.006289	0.023430
TR99	0.019232	0.022013	0.217692
Trigger	0.031299	0.031447	0.210682
TSN103	0.012073	0.012579	0.129564
TSN83	0.008181	0.006289	0.033687
Upbang	0.021651	0.022013	0.023002
Vau	0.007494	0.006289	0.052110
Wave	0.008326	0.006289	0.002607
Web	0.030095	0.028302	0.017376
Whitetip	0.004963	0.003145	0.027332
Zap	0.014768	0.015723	0.111819
Zig	0.006190	0.003145	0.000536
Zipfel	0.011039	0.009434	0.051991

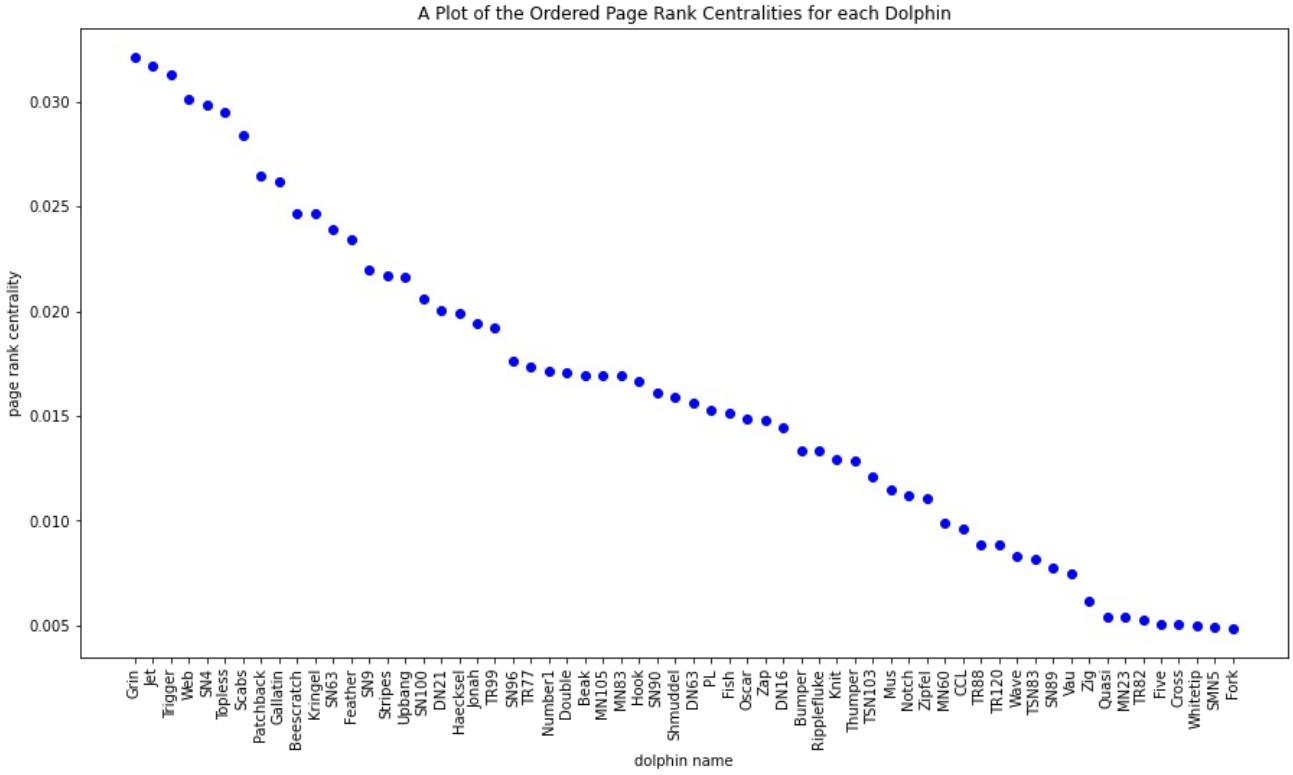
For a more intuitive understanding of these centralities, we can plot the dolphins in descending order according to each respective centrality measure:

In [479]:

```
page_rank_ordered_names = [[x for _, x in sorted(zip(page_rank(A), names), reverse=True)][i][0] for i in range(62
)]
degree_centrality_ordered_names = [[x for _, x in sorted(zip(degree_centrality(A), names), reverse=True)][i][0] f
or i in range(62)]
eigenvector_centrality_ordered_names = [[x for _, x in sorted(zip(eigenvector_centrality(A), names), reverse=True
)][i][0] for i in range(62)]

page_rank_ordered = sorted(page_rank(A), reverse=True)
degree_centrality_ordered = sorted(degree_centrality(A), reverse=True)
eigenvector_centrality_ordered = sorted(eigenvector_centrality(A), reverse=True)
```

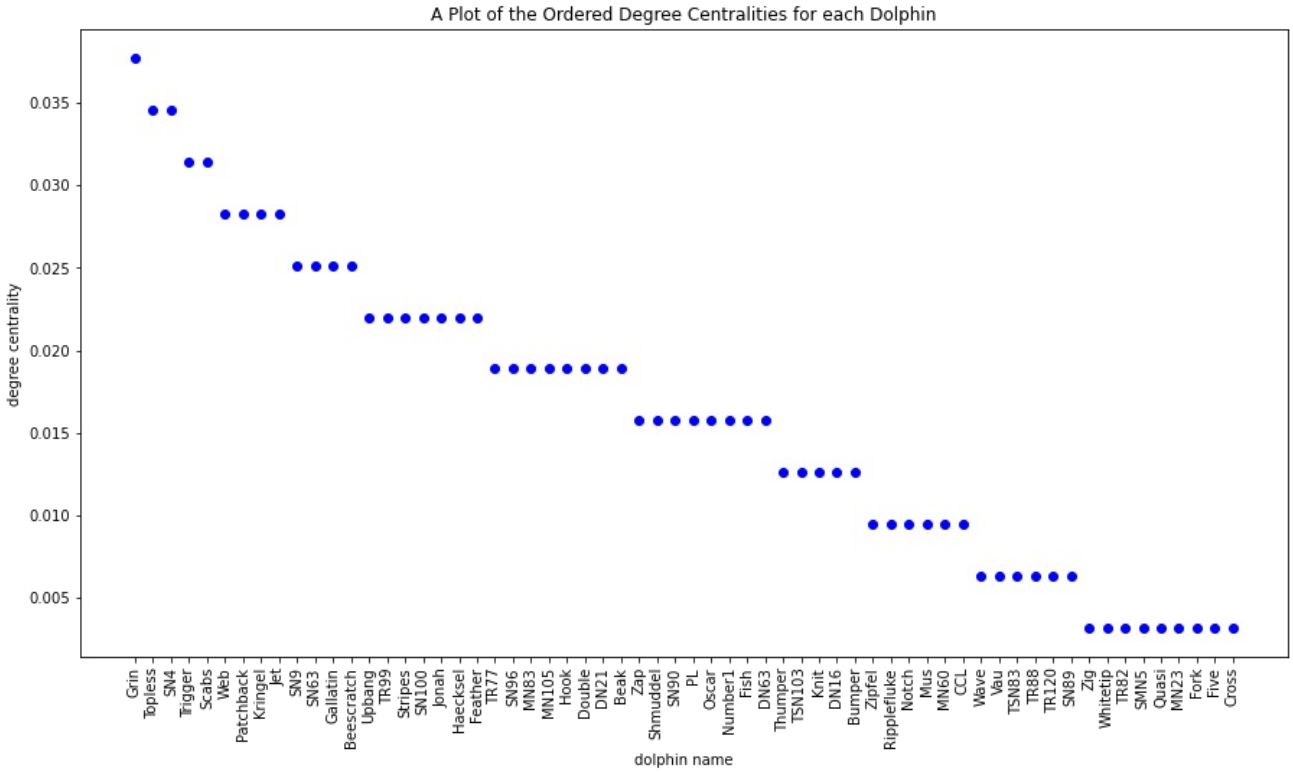
In [480]:

```
plt.figure(figsize=(15, 8))
plt.scatter(page_rank_ordered_names, page_rank_ordered, color='blue')
plt.xlabel('dolphin name')
plt.ylabel('page rank centrality')
plt.title('A Plot of the Ordered Page Rank Centralities for each Dolphin')
plt.xticks(rotation =90)
plt.show()
```



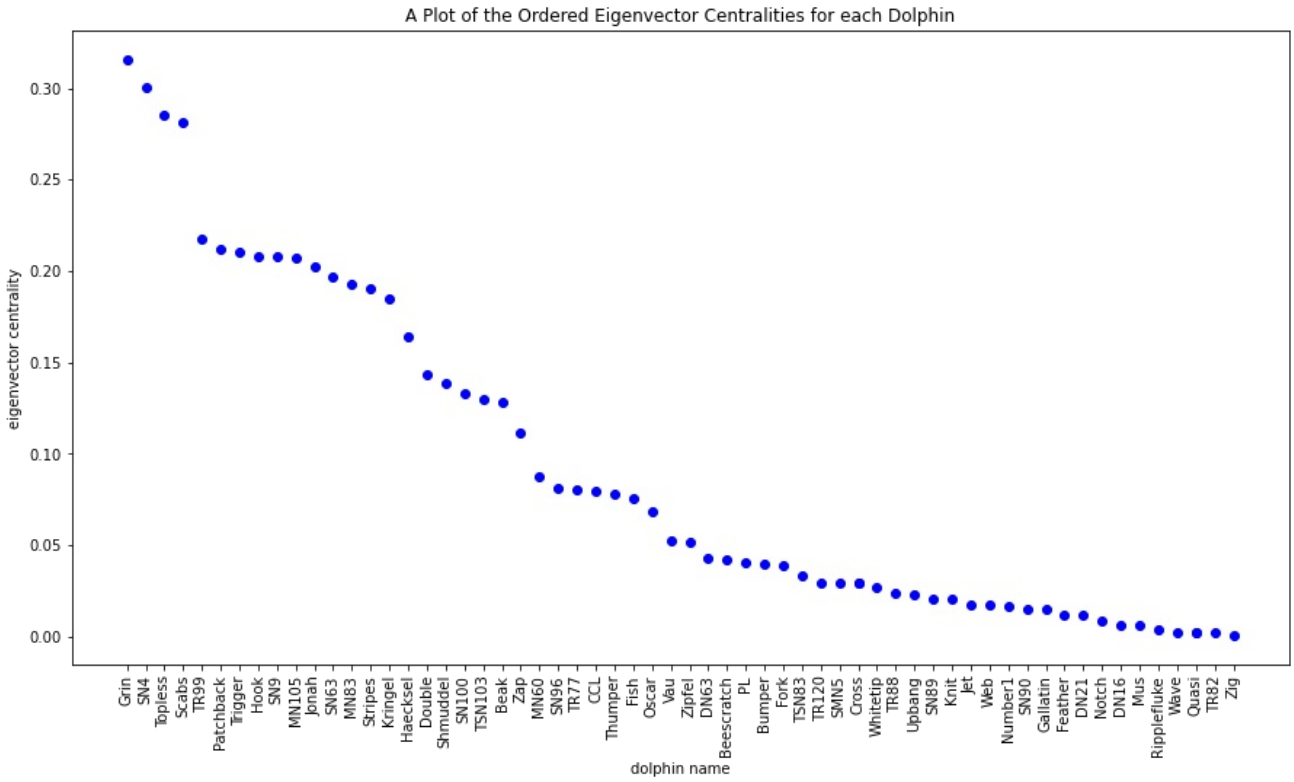
In [481]:

```
plt.figure(figsize=(15, 8))
plt.scatter(degree_centrality_ordered_names, degree_centrality_ordered, color='blue')
plt.xlabel('dolphin name')
plt.ylabel('degree centrality')
plt.title('A Plot of the Ordered Degree Centralities for each Dolphin')
plt.xticks(rotation =90)
plt.show()
```



In [482]:

```
plt.figure(figsize=(15, 8))
plt.scatter(eigenvector_centrality_ordered_names, eigenvector_centrality_ordered, color='blue')
plt.xlabel('dolphin name')
plt.ylabel('eigenvector centrality')
plt.title('A Plot of the Ordered Eigenvector Centralities for each Dolphin')
plt.xticks(rotation =90)
plt.show()
```



From the above plots, we can see that there is a small subset of the dolphins that are central to the network according to our three centrality measures. For example, 'Grin' has the highest centrality for all three measures and the top 10 dolphins for each measure are mostly a permutation of eachother. We can see this clearly in the following table:

In [484]:

```
data2 = {'Page Rank Centrality': page_rank_ordered_names[:10], 'Degree Centrality': degree_centrality_ordered_nam
es[:10], 'Eigenvector Centrality': eigenvector_centrality_ordered_names[:10]}
df2 = pd.DataFrame(data2, index=[i for i in range(1, 11)])
df2
```

Out[484]:

	Page Rank Centrality	Degree Centrality	Eigenvector Centrality
1	Grin	Grin	Grin
2	Jet	Topless	SN4
3	Trigger	SN4	Topless
4	Web	Trigger	Scabs
5	SN4	Scabs	TR99
6	Topless	Web	Patchback
7	Scabs	Patchback	Trigger
8	Patchback	Kringel	Hook
9	Gallatin	Jet	SN9
10	Beescratch	SN9	MN105

We see that 'Grin', 'Trigger', 'SN4', 'Topless', 'Scabs' and 'Patchback' are all in the top 10 most central nodes for each centrality measure. There is, however, some discrepencies between each measure's rankings due to the different aspects they attempt to measure. To complement this discussion, we plot the correlation matrix for the centralities and its associated scatter plots:

In [488]:

```
data3 = {'Page Rank': list(page_rank(A)), 'Degree': list(degree_centrality(A)), 'Eigenvector': list(eigenvector_c
entrality(A))}
corr = pd.DataFrame(data3, index=list(names.flatten())).corr()
print(corr)
```

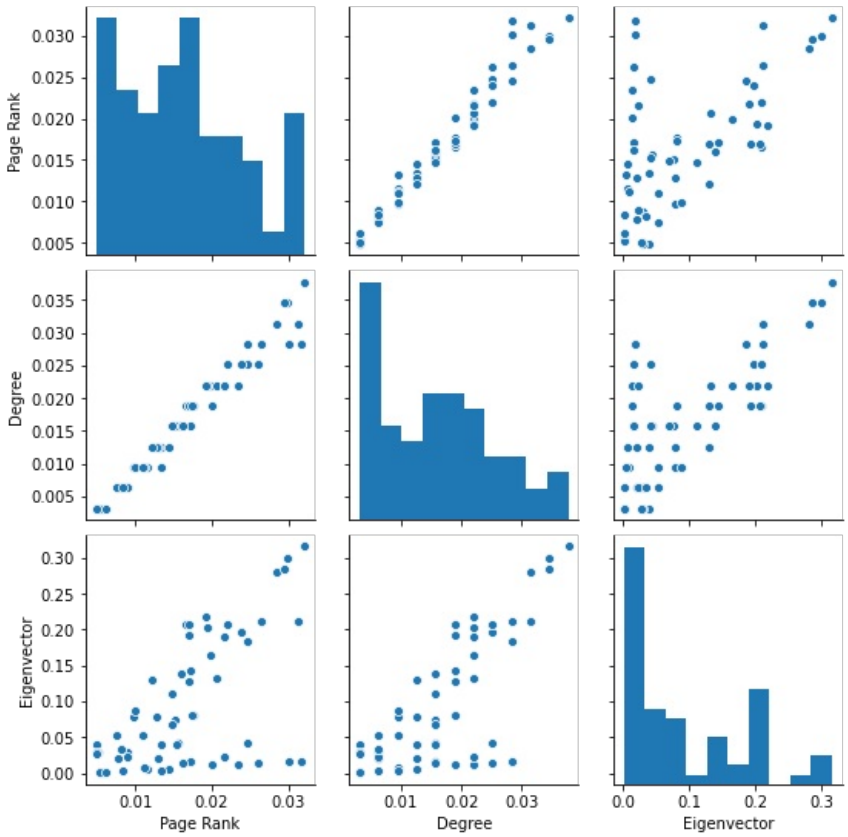
```
             Page Rank    Degree  Eigenvector
Page Rank     1.000000  0.983095     0.609417
Degree        0.983095  1.000000     0.719648
Eigenvector   0.609417  0.719648     1.000000
```

In [489]:

```
sns.pairplot(df1)
```

Out[489]:

```
<seaborn.axisgrid.PairGrid at 0x7f88509a3190>
```



We can see that the PageRank and Degree centralities are very highly correlated. This is expected, since the PageRank centrality is a measure of how important a node is according to its degree and the degrees of its neighbours. This correlation can be further seen via the similarity in each measures top 10 most central dolphins, as discussed earlier. Additionally, we see that the Eigenvector centrality is not especially correlated with the other two measures. Again, this is due to the particular features it is sensitive to: the Eigenvector centrality of a node considers its neighbour's connectivity, whilst the Degree centrality does not and the PageRank centrality dampens the influence of a neighbour's centrality. As such, the Eigenvector centrality considers a node with a single connection to a central node to be a relatively central node itself, despite visually and according to other measures not being central at all. This lack of correlation is also shown via the discrepencies in the top 10 rankings table discussed previously.

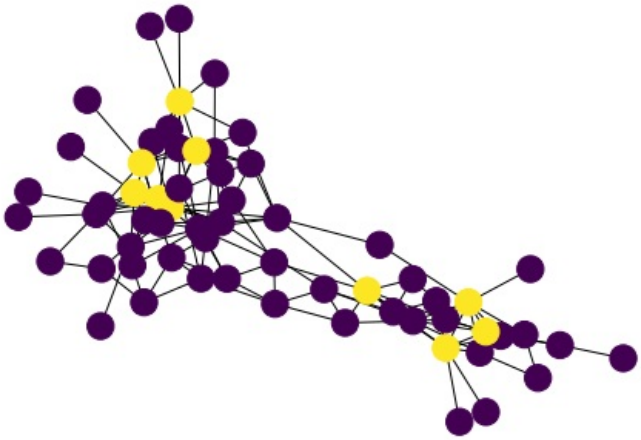
We can visualise the top 10 nodes, coloured yellow, according to each centrality measure graphically:

In [490]:

```
page_rank_indices = []

for name in page_rank_ordered_names[:10]:
    page_rank_indices.append(list(names).index(name))

degree_centrality_indices = []

for name in degree_centrality_ordered_names[:10]:
    degree_centrality_indices.append(list(names).index(name))

eigenvector_centrality_indices = []

for name in eigenvector_centrality_ordered_names[:10]:
    eigenvector_centrality_indices.append(list(names).index(name))


page_rank_binary = np.zeros(names.shape[0])
degree_centrality_binary = np.zeros(names.shape[0])
eigenvector_centrality_binary = np.zeros(names.shape[0])

page_rank_binary[page_rank_indices] = 1
degree_centrality_binary[degree_centrality_indices] = 1
eigenvector_centrality_binary[eigenvector_centrality_indices] = 1
```

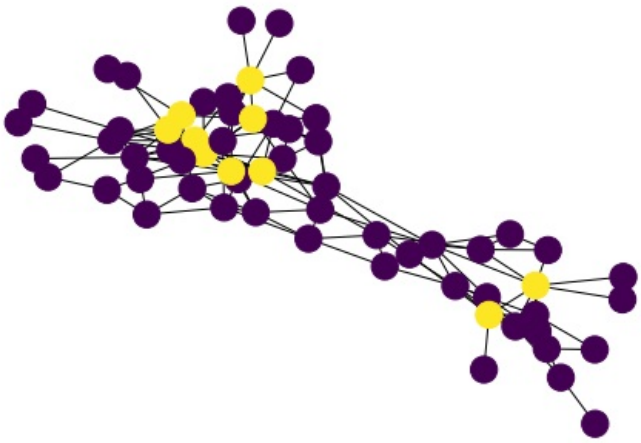
In [502]:

```
nx.draw(g, node_color=page_rank_binary)  # PageRank centrality.
```



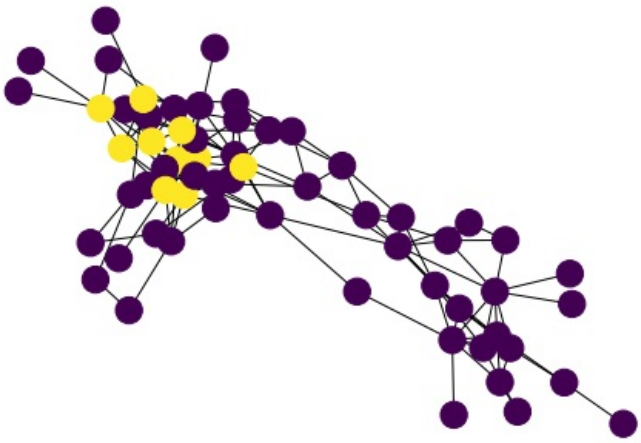
In [222]:

```
nx.draw(g, node_color=degree_centrality_binary)  # Degree centrality.
```



In [227]:

```
nx.draw(g, node_color=eigenvector_centrality_binary)  # Eigenvector centrality.
```



From these graphs we can further see how the PageRank and Degree centralities are correlated, due to their similarity in node selection. Whilst the Eigenvector centrality has some similarity, it is distinctly different to the other two.

We finish this discussion with reference to the original paper by Lusseau et al. The original paper suggests that there are 3 predominant groups of dolphins. Visually, this is in-keeping with our graphs where, although we only plotted a bipartition of the graph, it is easy to see how the graph could be split into 3 groups. According to the original paper, the most central dolphins in group 1 and 2 are:

1. Group 1: 'Jonah' and 'Topless'
2. Group 2: 'Gallatin'
3. Group 3: 'Scabs'

Whilst 'Jonah' does not appear in any of our top 10 most clustered nodes according to each measure and 'Gallatin' only appears in the PageRank centrality top 10, 'Topless' and 'Scabs' rank highly in all three measures: our analysis agrees with the original paper that 'Topless' and 'Scabs' are central dolphins. However, our analysis also differs from the original paper quite significantly. For all three centrality measures, we ranked 'Grin' as the most central Dolphin, but the original paper has no mention of 'Grin'. This seems counter-intuitive and is presumably due to the original paper using different centrality measures to us, which in turn have different sensitivities to the data. This would also explain why 'Jonah' and 'Gallatin' appear so infrequently in our top 10 ranking despite being two of the most central nodes according to the original paper.

This concludes the notebook. Thank you for your time in reading this project and we hope you found it insightful.