DATA SCIENCE COURSEWORK NUMBER 2

by Hermine Tranié, CID: 01400919

TASK 1: NEURAL NETWORKS

1.0 Load data

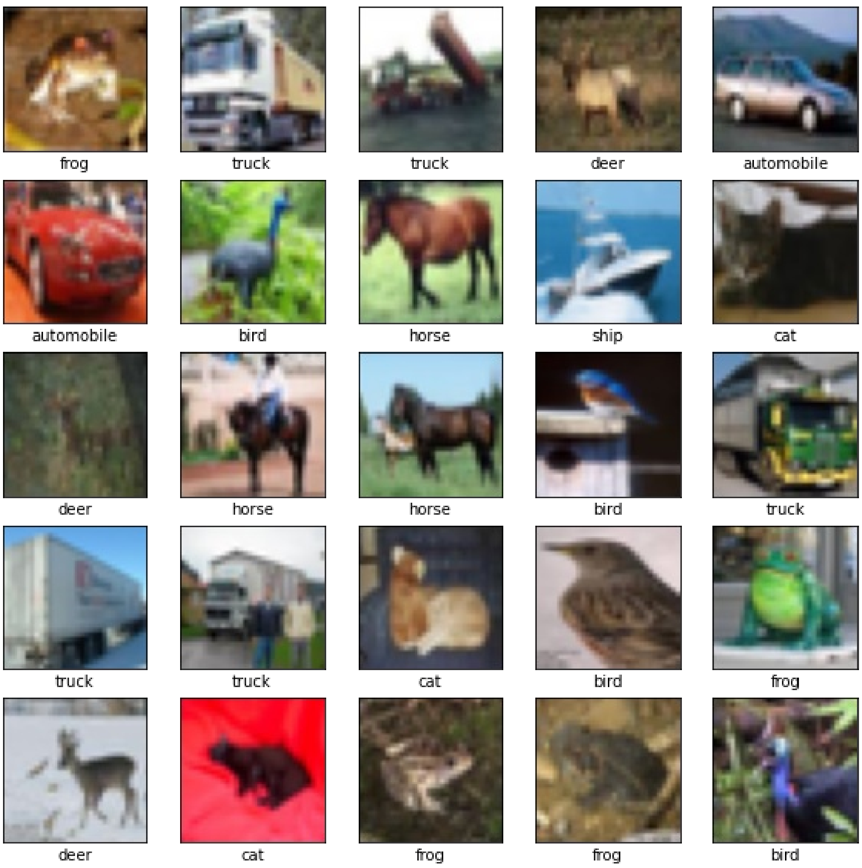
First, we start by loading the necessary packages

In [28]:
```
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
```

Preview data samples

In [29]:
```
(x_train, y_train), (x_val, y_val) = tf.keras.datasets.cifar10.load_data()
x_train = x_train.astype('float32') / 255
x_val = x_val.astype('float32') / 255
```

In [30]:
```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_train[i], cmap=plt.cm.binary)
    # The CIFAR labels happen to be arrays, which is why you need the extra index
    plt.xlabel(class_names[y_train[i][0]])
plt.show()
```



Thanks to the sample of our data, we can see that the dataset is comprised of 10 different classes to identify, namely airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. Our goal throughout this exercise is to implement two different methods of neural networks to maximise the accuracy of the detection of these objects. We will be implementing:

1. a Multilayer Perceptron with 5 hidden layers coded using NumPy only
2. a Convolutional Neural Network with 4 hidden layers coded using Tensorflow

Data exploration

First of all, we load the data, as instructed and we resize if needed.

In [31]:
```
def load_data():
    (x_train, y_train), (x_val, y_val) = tf.keras.datasets.cifar10.load_data()
    x_train = x_train.astype('float32') / 255
    x_val = x_val.astype('float32') / 255

    # convert labels to categorical samples
    y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
    y_val = tf.keras.utils.to_categorical(y_val, num_classes=10)
    return ((x_train, y_train), (x_val, y_val))

(x_train, y_train), (x_val, y_val) = load_data()
```

In [32]:
```
print('The shape of training predictors is:', x_train.shape)
print('The shape of training predictors is:', y_train.shape)
print('The shape of validation predictors is:', x_val.shape)
print('The shape of validation predictors is:', y_val.shape)
```

```
The shape of training predictors is: (50000, 32, 32, 3)
The shape of training predictors is: (50000, 10)
The shape of validation predictors is: (10000, 32, 32, 3)
The shape of validation predictors is: (10000, 10)
```

In [33]:
```
# Resize the predictors shape in order to feed to the algorithm
x_train = x_train.reshape([x_train.shape[0],-1])
x_val = x_val.reshape([x_val.shape[0],-1])
```

In [34]:
```
x_train
```

Out[34]:
```
array([[0.23137255, 0.24313726, 0.24705882, ..., 0.48235294, 0.36078432,
        0.28235295],
       [0.6039216 , 0.69411767, 0.73333335, ..., 0.56078434, 0.52156866,
        0.5647059 ],
       [1.        , 1.        , 1.        , ..., 0.3137255 , 0.3372549 ,
        0.32941177],
       ...,
       [0.13725491, 0.69803923, 0.92156863, ..., 0.04705882, 0.12156863,
        0.19607843],
       [0.7411765 , 0.827451  , 0.9411765 , ..., 0.7647059 , 0.74509805,
        0.67058825],
       [0.8980392 , 0.8980392 , 0.9372549 , ..., 0.6392157 , 0.6392157 ,
        0.6313726 ]], dtype=float32)
```

In [35]:
```
print('The shape of training predictors is:', x_train.shape)
print('The shape of validation predictors is:', x_val.shape)
```

```
The shape of training predictors is: (50000, 3072)
The shape of validation predictors is: (10000, 3072)
```

1.1 Multi-layer perceptron

1.1.0 Shuffle data

First, we want define a function to shuffle the data, to pick our batches selection as random as possible at the beginning of every epoch.

In [36]:
```
### SHUFFLE DATA ###

shuffle = np.random.permutation(range(x_train.shape[0]))
x_train = x_train[shuffle]
y_train = y_train[shuffle]
```

In [37]:
```
x_train
```

Out[37]:
```
array([[0.9411765 , 0.9372549 , 0.9529412 , ..., 0.95686275, 0.95686275,
         0.9529412 ],
        [0.4392157 , 0.40784314, 0.39607844, ..., 0.03137255, 0.02352941,
         0.01568628],
        [0.04705882, 0.05882353, 0.0627451 , ..., 0.08235294, 0.08235294,
         0.09019608],
        ...,
        [0.81960785, 0.9137255 , 0.9843137 , ..., 0.42352942, 0.54509807,
         0.6784314 ],
        [0.23137255, 0.22745098, 0.21176471, ..., 0.48235294, 0.32156864,
         0.23137255],
        [0.1764706 , 0.16078432, 0.12941177, ..., 0.31764707, 0.30980393,
         0.27058825]], dtype=float32)
```

1.1.1 Train multi-layer perceptron with LR=0.01 on 40 epochs

First let's randomly initialize our bias and our weights.

In [38]:
```
### Parameters initialization of MLP as in CT ###

var0 = 2. / (3072 + 400)
W0 = np.random.randn(3072, 400) * np.sqrt(var0) # initial size of image
b0 = np.zeros(400)

var1 = 2. / (400 + 400)
W1 = np.random.randn(400, 400) * np.sqrt(var1)
b1 = np.zeros(400)

var2 = 2. / (400 + 400)
W2 = np.random.randn(400,400) * np.sqrt(var2)
b2 = np.zeros(400)

var3 = 2. / (400 + 400)
W3 = np.random.randn(400,400) * np.sqrt(var3)
b3 = np.zeros(400)

var4 = 2. / (400 + 400)
W4 = np.random.randn(400,400) * np.sqrt(var4)
b4 = np.zeros(400)

var5 = 2. / (10 + 400)
W5 = np.random.randn(400,10) * np.sqrt(var5) # output value of 10 neurons
b5 = np.zeros(10)
```

Note that by initializing our paremeters, we can compute the number of parameters that our model is going to deal with. We have the weights W_i and bias b_i. The total number of parameters of our model is: $\sum_{i=0}^5 size(W_i) + size(b_i) = 400(3072 + 4*400 + 10) + 5*400 + 10 = 1,874,810$. So we have around 1.8 million parameters in our Multilayer perceptron.

Now let's define functions in order to implement our Multilayer Perceptron with Stochastic Gradient Descent (SGD) and cross-entropy loss as follows:

- 5 hidden layers (400 neurons each)\ --> tanh activation function
- 1 output layer (10 neurons)\ --> softmax activation function

We will train our model on batches of 128 points with a learning rate of 0.01 for 40 epochs.

Our main functions include:

1. the function *dense* which implements the dense layer transformation to obtain the pre-activation values (from CT)
2. the activation functions *tanh* and *softmax* and the derivative of tanh: *tanh_derivative*
3. the function *cross_entropy* which computes the cross entropy loss of the function
4. the function *output_error* which compute the error [8]
5. the function *backpropagate* which implement the backpropagation algorithm (from CT)
6. the function *MLP* which computes the output predicted vector of probabilities after having gone through the whole network
7. the function *grads* which computes the gradients for each weights and bias

This whole set up of functions will then allow us to build our model.

In [39]:
```
### Define dense function for dense layer transformation to obtain the pre-activation values ###

def dense(a, W, b):

    # a: K x a_in array of inputs
    # W: a_in x a_out array for kernel matrix parameters
    # b: Length a_out 1-D array for bias parameters
```

```
    # returns: K x a_out output array

    h = b + a @ W
    return h
```

In [40]:

```
### Define activation functions: tanh and softmax ###

def tanh(x): # define tanh activation function

    return np.tanh(x)

def tanh_derivative(x): # define tanh derivative

    return 1 - tanh(x)**2

def softmax(x): # define output layer activation function

    return (np.exp(x)) / (np.exp(x).sum(axis = 1)[:, None])
```

In order to put some background explanation for our activation functions (especially softmax, because tanh derivative is quite straight-forward):

We won't be computing the derivative of softmax directly, but instead, we'll combine it with the loss function: cross-entropy, as suggested by [8]:

First, we define the cross entropy $CE=-\sum_j t_j \log(o_j)$ with t_j is the target output for neuron j (y_val) and o_j is the output of the neuron (y_pred).

We get that $\frac{\partial CE}{\partial o_j} = -\frac{t_j}{o_j}$

Note that for our final loss is computed with the softmax activation function defined as follows: $ o_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$, where z is the set of inputs to all neurons in the softmax layer.

So we get that $\frac{\partial CE}{\partial z_j} = \sum_i \frac{\partial CE}{\partial o_i} \frac{\partial o_i}{\partial z_j}$. We can compute this, by splitting into 2 cases:

1. $i=j$: $\frac{\partial CE}{\partial o_i} \frac{\partial o_i}{\partial z_j} = -\frac{t_j}{o_j} o_j(1-o_j) = t_j o_j - t_j$
2. $i\neq j$: $\frac{\partial o_i}{\partial z_j} = \frac{\partial \frac{e^{z_i}}{\sum_k e^{z_k}}}{\partial z_j}= -\frac{e^{z_i}}{(\sum_k e^{z_k})^2}e^{z_j}=-o_io_j$ Combining these two, it yields: $\frac{\partial CE}{\partial z_j}=\sum_i t_io_j - t_j$

But we have that y_val (or t) is a one-hot encoded vector so $\sum_i t_i =1$. Hence: $\frac{\partial CE}{\partial z_j} = o_j -t_j$. (hence the definition of output error function below)

In [41]:

```
### Define loss function

def cross_entropy(y, x):

    ce = - x*np.log(y)

    return np.sum(ce)/y.shape[0]
```

In [42]:

```
### Error from softmax ### [8]

def output_error(y_true, a):

    # y_true: K x 1 array of data outputs
    # a: K x 1 array of output pre-activations
    # returns: K x 1 array of output errors

    return softmax(a) - y_true
```

Now we can implement the backpropagation algorithm as defined in the CT.

In [43]:

```
### Define backpropagate errors to find the previous delta ###

def backpropagate(delta, W, a):

    wt = delta @ W.T
    out = tanh_derivative(a) * wt

    return out
```

In [44]:

```
### Define MLP that does through all layers and outputs predicted y ###

def mlp(x, W0, b0, W1, b1, W2, b2, W3, b3, W4, b4, W5, b5):

    h = dense(x, W0, b0) # initial layer: hidden
    h = tanh(h) # tanh activation

    h = dense(h, W1, b1) # hidden layer
```

```python
    h = tanh(h) # tanh activation

    h = dense(h, W2, b2) # hidden layer
    h = tanh(h) # tanh activation

    h = dense(h, W3, b3) # hidden layer
    h = tanh(h) # tanh activation

    h = dense(h, W4, b4) # hidden layer
    h = tanh(h) # tanh activation

    h = dense(h, W5, b5) # output layer
    y = softmax(h) # softmax activation

    return np.array(y)
```

In [45]:

```python
### Compute gradients averaged over batch size ###

def grads(delta1, delta2, delta3, delta4, delta5, delta6, h0, h1, h2, h3, h4, h5):

    # gradients computation

    grad_W0 = h0.T@delta1 # gradient pour all variable W0 (weight matrix for layer 0) (3072x400)
    grad_b0 = delta1 # gradient of the bias (cross entropy loss function)
    grad_W1 = h1.T@delta2 # 400 x 400
    grad_b1 = delta2
    grad_W2 = h2.T@delta3
    grad_b2 = delta3
    grad_W3 = h3.T@delta4
    grad_b3 = delta4
    grad_W4 = h4.T@delta5
    grad_b4 = delta5
    grad_W5 = h5.T@delta6 # 10 x 400
    grad_b5 = delta6

    # gradients averaged over batch size

    grad_W0 /= h0.shape[0]
    grad_b0 = np.mean(grad_b0, axis=0)
    grad_W1 /= h1.shape[0]
    grad_b1 = np.mean(grad_b1, axis=0)
    grad_W2 /= h2.shape[0]
    grad_b2 = np.mean(grad_b2, axis=0)
    grad_W3 /= h3.shape[0]
    grad_b3 = np.mean(grad_b3, axis=0)
    grad_W4 /= h4.shape[0]
    grad_b4 = np.mean(grad_b4, axis=0)
    grad_W5 /= h5.shape[0]
    grad_b5 = np.mean(grad_b5, axis=0)

    return grad_W0, grad_b0, grad_W1, grad_b1, grad_W2, grad_b2, grad_W3, grad_b3, grad_W4, grad_b4, grad_W5, gra
```

Now that we have defined all the necessary functions, we can train our multilayer perceptron as follows: (40 epochs with 0.01 learning rate)

In [48]:

```python
### FUNCTION TO TRAIN MLP THROUGH ALL THE EPOCHS ###

def train_MLP(epochs, lr, batches, x_train, y_train):

  # initialize random weights and bias
  var0 = 2. / (3072 + 400) # hidden layer n1
  W0 = np.random.randn(3072, 400) * np.sqrt(var0) # initial size of image
  b0 = np.zeros(400)

  var1 = 2. / (400 + 400) # hidden layer n2
  W1 = np.random.randn(400, 400) * np.sqrt(var1)
  b1 = np.zeros(400)

  var2 = 2. / (400 + 400) # hidden layer n3
  W2 = np.random.randn(400,400) * np.sqrt(var2)
  b2 = np.zeros(400)

  var3 = 2. / (400 + 400) # hidden layer n4
  W3 = np.random.randn(400,400) * np.sqrt(var3)
  b3 = np.zeros(400)

  var4 = 2. / (400 + 400) # hidden layer n5
  W4 = np.random.randn(400,400) * np.sqrt(var4)
  b4 = np.zeros(400)

  var5 = 2. / (10 + 400) # output layer
  W5 = np.random.randn(400,10) * np.sqrt(var5) # output value of 10 neurons
  b5 = np.zeros(10)

  # initialize cross-entropy loss and accuracy
  L = []
  L_val = []
  A = []
```

```
    A_val = []

    # define range of looping
    lp = np.linspace(0, 50000-1, batches, dtype = int)

    for epoch in range(epochs): # loop over number of epochs

      # print what epoch we're at
      print("epoch {}/{}" .format(epoch+1,epochs))

      # shuffle data for stochastic aspect of SGD
      shuffle = np.random.permutation(range(x_train.shape[0]))
      x_train = x_train[shuffle]
      y_train = y_train[shuffle]

      for j in range(lp.size-2): # loop over number of iterations

          if j == int(1/4*(lp.size-3)): # print when we've gone through 1/4 of one epoch
              print ('[25%]', end='')
          if j == int(2/4*(lp.size-3)): # print when we've gone through 1/2 of one epoch
              print ('[50%]', end='')
          if j == int(3/4*(lp.size-3)): # print when we've gone through 3/4 of one epoch
              print ('[75%]', end='')
          if j == lp.size-3: # print when we've gone through the whole of one epoch
              print ('[100%]')

          # first, select batch of 128 images from predictors and response variable
          x_batch = x_train[lp[j]:lp[j+1]] # select batched predictors: first 128, then next 128 etc (no problem as
          y_batch = y_train[lp[j]:lp[j+1]] # select batched response variable

          # then, update layers in forward way
          h0 = x_batch
          a1 = dense(x_batch, W0, b0)
          h1 = tanh(a1)
          a2 = dense(h1, W1, b1)
          h2 = tanh(a2)
          a3 = dense(h2, W2, b2)
          h3 = tanh(a3)
          a4 = dense(h3, W3, b3)
          h4 = tanh(a4)
          a5 = dense(h4, W4, b4)
          h5 = tanh(a5)
          a6 = dense(h5, W5, b5)

          # update deltas: backpropagate
          delta6 = output_error(y_batch, a6)
          delta5 = backpropagate(delta6, W5, a5)
          delta4 = backpropagate(delta5, W4, a4)
          delta3 = backpropagate(delta4, W3, a3)
          delta2 = backpropagate(delta3, W2, a2)
          delta1 = backpropagate(delta2, W1, a1)

          # compute gradients
          grad_W0, grad_b0, grad_W1, grad_b1, grad_W2, grad_b2, grad_W3, grad_b3, grad_W4, grad_b4, grad_W5, grad_b

          # update parameters with gradient: SGD
          W0 -= lr*grad_W0
          b0 -= lr*grad_b0
          W1 -= lr*grad_W1
          b1 -= lr*grad_b1
          W2 -= lr*grad_W2
          b2 -= lr*grad_b2
          W3 -= lr*grad_W3
          b3 -= lr*grad_b3
          W4 -= lr*grad_W4
          b4 -= lr*grad_b4
          W5 -= lr*grad_W5
          b5 -= lr*grad_b5

      # compute cross-entropy loss
      Loss = cross_entropy(mlp(x_train, W0, b0, W1, b1, W2, b2, W3, b3, W4, b4, W5, b5), y_train)
      Loss_val = cross_entropy(mlp(x_val, W0, b0, W1, b1, W2, b2, W3, b3, W4, b4, W5, b5), y_val)

      # compute accuracy
      Acc = 100 * np.sum(np.argmax(mlp(x_train, W0, b0, W1, b1, W2, b2, W3, b3, W4, b4, W5, b5), axis=1) == np.argm
      Acc_val = 100 * np.sum(np.argmax(mlp(x_val, W0, b0, W1, b1, W2, b2, W3, b3, W4, b4, W5, b5), axis=1) == np.ar

      print("The cross-entropy loss on the train set {} " .format(Loss))
      print("The cross-entropy loss on the validation set {} " .format(Loss_val))
      print("The accuracy on the train set {} " .format(Acc))
      print("The accuracy on the validation set {} " .format(Acc_val))

      # update cross-entropy loss
      L.append(Loss) # train
      L_val.append(Loss_val) # val

      # update accuracy
      A.append(Acc) # train
      A_val.append(Acc_val) # val
```

```python
    return L, L_val, A, A_val
```

In [50]:

```python
### TRAIN MLP ON 40 EPOCHS WITH LEARNING RATE 0.01 ###

# begin timing
import time
start = time.time()

# define model training parameters
epochs = 40
lr = 0.01
batches = 390

L_1, L_val_1, A_1, A_val_1 = train_MLP(epochs, lr, batches, x_train, y_train)

# end timing
end = time.time()
time1 = int((end-start)//60)
print("The ellapsed time for 40 epochs with lr=0.01 is:", time1)
```

```
epoch 1/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.8168576256812559
The cross-entropy loss on the validation set 1.8196304317996794
The accuracy on the train set 35.332
The accuracy on the validation set 35.74
epoch 2/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.7354872570076554
The cross-entropy loss on the validation set 1.740361003290435
The accuracy on the train set 38.832
The accuracy on the validation set 37.9
epoch 3/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.7392545002111715
The cross-entropy loss on the validation set 1.7425731739623396
The accuracy on the train set 38.322
The accuracy on the validation set 38.3
epoch 4/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.648427572046266
The cross-entropy loss on the validation set 1.6587540667128353
The accuracy on the train set 41.992
The accuracy on the validation set 41.53
epoch 5/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.629877110231575
The cross-entropy loss on the validation set 1.6375125199937994
The accuracy on the train set 42.372
The accuracy on the validation set 41.98
epoch 6/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.6047837979027633
The cross-entropy loss on the validation set 1.6179873575798949
The accuracy on the train set 43.806
The accuracy on the validation set 43.03
epoch 7/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.5623227120700458
The cross-entropy loss on the validation set 1.5760744659137067
The accuracy on the train set 44.914
The accuracy on the validation set 44.06
epoch 8/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.5383370187889926
The cross-entropy loss on the validation set 1.5594323847043425
The accuracy on the train set 46.026
The accuracy on the validation set 45.12
epoch 9/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.5368282197394725
The cross-entropy loss on the validation set 1.5657854766531514
The accuracy on the train set 45.322
The accuracy on the validation set 44.06
epoch 10/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.4984287901808953
The cross-entropy loss on the validation set 1.5271002782534482
The accuracy on the train set 46.896
The accuracy on the validation set 46.14
epoch 11/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.5154089518567786
```

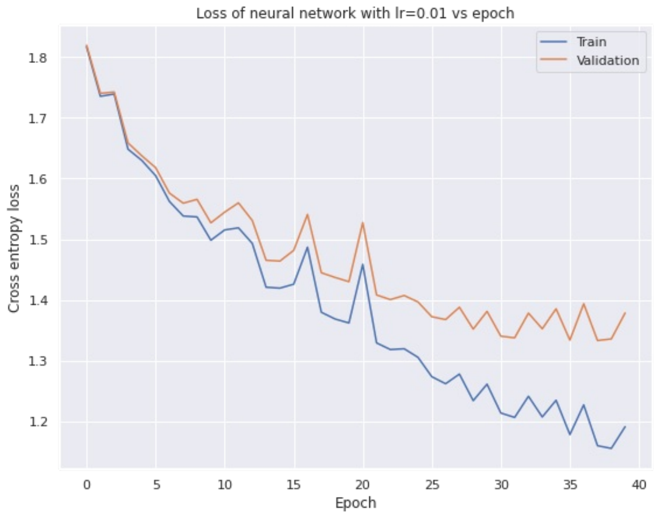
```
The cross-entropy loss on the validation set 1.5446675491509108
The accuracy on the train set 46.67
The accuracy on the validation set 45.49
epoch 12/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.518866593484001
The cross-entropy loss on the validation set 1.5600404276367004
The accuracy on the train set 46.074
The accuracy on the validation set 44.68
epoch 13/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.4932346682062676
The cross-entropy loss on the validation set 1.5310221447810617
The accuracy on the train set 46.44
The accuracy on the validation set 45.05
epoch 14/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.421062842804738
The cross-entropy loss on the validation set 1.465486094844629
The accuracy on the train set 49.998
The accuracy on the validation set 47.96
epoch 15/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.41954726797015
The cross-entropy loss on the validation set 1.4640978878282165
The accuracy on the train set 49.68
The accuracy on the validation set 48.2
epoch 16/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.4260863119572447
The cross-entropy loss on the validation set 1.481788862326981
The accuracy on the train set 48.994
The accuracy on the validation set 47.35
epoch 17/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.487047209654671
The cross-entropy loss on the validation set 1.5410229578004753
The accuracy on the train set 47.0
The accuracy on the validation set 44.81
epoch 18/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.3798084607415375
The cross-entropy loss on the validation set 1.4448526306546212
The accuracy on the train set 50.798
The accuracy on the validation set 48.33
epoch 19/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.368715548639018
The cross-entropy loss on the validation set 1.437189911712698
The accuracy on the train set 51.322
The accuracy on the validation set 48.95
epoch 20/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.3621746661986045
The cross-entropy loss on the validation set 1.4301877279055202
The accuracy on the train set 51.794
The accuracy on the validation set 49.36
epoch 21/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.4587498637832024
The cross-entropy loss on the validation set 1.5274976756883432
The accuracy on the train set 47.398
The accuracy on the validation set 44.99
epoch 22/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.3293827339006399
The cross-entropy loss on the validation set 1.4083219396219657
The accuracy on the train set 52.764
The accuracy on the validation set 49.48
epoch 23/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.3184294354836839
The cross-entropy loss on the validation set 1.400704258846461
The accuracy on the train set 53.268
The accuracy on the validation set 49.53
epoch 24/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.3197422621529735
The cross-entropy loss on the validation set 1.407415716649177
The accuracy on the train set 53.092
The accuracy on the validation set 49.96
epoch 25/40
[25%][50%][75%][100%]
```

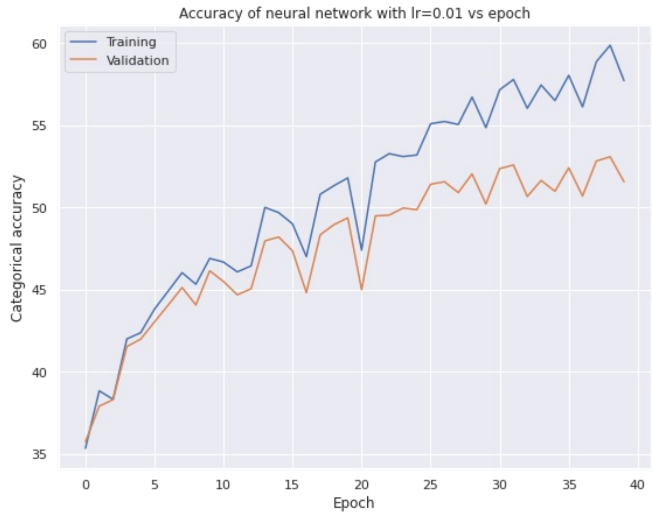
```
The cross-entropy loss on the train set 1.3056327700347974
The cross-entropy loss on the validation set 1.3968080158035812
The accuracy on the train set 53.188
The accuracy on the validation set 49.86
epoch 26/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.2737348574766163
The cross-entropy loss on the validation set 1.3725086067941548
The accuracy on the train set 55.09
The accuracy on the validation set 51.41
epoch 27/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.2621247251429528
The cross-entropy loss on the validation set 1.3677915943060859
The accuracy on the train set 55.226
The accuracy on the validation set 51.56
epoch 28/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.2780241739679448
The cross-entropy loss on the validation set 1.3881122309949667
The accuracy on the train set 55.046
The accuracy on the validation set 50.9
epoch 29/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.2342460526016952
The cross-entropy loss on the validation set 1.3520669750770824
The accuracy on the train set 56.71
The accuracy on the validation set 52.04
epoch 30/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.2614595881304818
The cross-entropy loss on the validation set 1.381335232051496
The accuracy on the train set 54.854
The accuracy on the validation set 50.21
epoch 31/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.213901847910029
The cross-entropy loss on the validation set 1.340428017845366
The accuracy on the train set 57.156
The accuracy on the validation set 52.36
epoch 32/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.206641275192012
The cross-entropy loss on the validation set 1.3377483095501046
The accuracy on the train set 57.784
The accuracy on the validation set 52.58
epoch 33/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.2415410466265877
The cross-entropy loss on the validation set 1.3784306474413863
The accuracy on the train set 56.03
The accuracy on the validation set 50.66
epoch 34/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.2073870463876937
The cross-entropy loss on the validation set 1.3526021837005051
The accuracy on the train set 57.45
The accuracy on the validation set 51.64
epoch 35/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.2349798590049401
The cross-entropy loss on the validation set 1.385560298983288
The accuracy on the train set 56.5
The accuracy on the validation set 50.98
epoch 36/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.1784115552182781
The cross-entropy loss on the validation set 1.3339619524162212
The accuracy on the train set 58.038
The accuracy on the validation set 52.41
epoch 37/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.2272974777228232
The cross-entropy loss on the validation set 1.3938560235612827
The accuracy on the train set 56.114
The accuracy on the validation set 50.69
epoch 38/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.1600874333078173
The cross-entropy loss on the validation set 1.3332779713952798
The accuracy on the train set 58.868
The accuracy on the validation set 52.82
epoch 39/40
```

```
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.1556330092785982
The cross-entropy loss on the validation set 1.3359074195904899
The accuracy on the train set 59.87
The accuracy on the validation set 53.08
epoch 40/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.1913785656138878
The cross-entropy loss on the validation set 1.3786912365482749
The accuracy on the train set 57.712
The accuracy on the validation set 51.56
The ellapsed time for 40 epochs with lr=0.01 is: 22
```

In [51]:

```
### PLOT LEARNING CURVES FOR MLP ON 40 EPOCHS FOR LEARNING RATE 0.01 ###
fig = plt.figure(figsize=(20, 7))

sns.set()

fig.add_subplot(121)
plt.plot(L_1, label="Train")
plt.plot(L_val_1, label="Validation")
plt.xlabel("Epoch")
plt.ylabel("Cross entropy loss")
plt.legend()
plt.title("Loss of neural network with lr=0.01 vs epoch")

fig.add_subplot(122)
plt.plot(A_1, label="Training")
plt.plot(A_val_1, label="Validation")
plt.xlabel("Epoch")
plt.ylabel("Categorical accuracy")
plt.legend()
plt.title("Accuracy of neural network with lr=0.01 vs epoch")

plt.show()
```





In [52]:

```
print("The best loss for learning rate 0.01 on 40 epochs is:", min(L_val_1))
print("The best validation accuracy for learning rate 0.01 on 40 epochs is:", max(A_val_1))
```

```
The best loss for learning rate 0.01 on 40 epochs is: 1.3332779713952798
The best validation accuracy for learning rate 0.01 on 40 epochs is: 53.08
```

Convergence of the model\ Clearly, this model with learning rate 0.01 on 40 epochs is quite interesting because it yields us a best validation accuracy of 53% which is already a bit more than 5 times better than randomness (because we have 10 categories).\ Regarding the convergence of the model, we can see that our model learns pretty fast as we can see a significant drop in the cross entropy loss over the training set, with some noise due to the stochastic aspect of our optimization (instead of computing the descent in the same direction at every epoch, we re-shuffle the data and find the best new direction: SGD). Regarding the validation set, the drop is a bit less significant but still quite satisfying. We can see that no overfitting is occurring on these 40 epochs, as the validation loss isn't going back up, however its drop is slowing down a lot which might mean that if we were to increase the number of epochs, it could overfit, we will explore this in 1.1.3.\ However one must note that we might want to try other parameters on the model, like increasing epochs or varying learning rate in order to see if we could improve the model accuracy.

1.1.2 Train MLP with different learning rates

Now let's try implement our MLP with different learning rates to see the impact on the loss and the accuracy:

LEARNING RATE = 0.0001

In [54]:
```
### TRAIN MLP ON 40 EPOCHS WITH LEARNING RATE 0.0001 ###

# begin timing
import time
start = time.time()

# define model training parameters
epochs = 40
lr = 0.0001
batches = 390

L_2, L_val_2, A_2, A_val_2 = train_MLP(epochs, lr, batches, x_train, y_train)

# end timing
end = time.time()
time2 = int((end-start)//60)
print("The ellapsed time for 40 epochs with lr=0.0001 is:", time2)
```

```
epoch 1/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.2897608716922173
The cross-entropy loss on the validation set 2.289581183973729
The accuracy on the train set 12.814
The accuracy on the validation set 12.51
epoch 2/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.2659613322937857
The cross-entropy loss on the validation set 2.2659259305171076
The accuracy on the train set 15.72
The accuracy on the validation set 15.6
epoch 3/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.244822765913386
The cross-entropy loss on the validation set 2.2449170458917784
The accuracy on the train set 17.996
The accuracy on the validation set 17.97
epoch 4/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.224902084214757
The cross-entropy loss on the validation set 2.2251541365526295
The accuracy on the train set 20.08
The accuracy on the validation set 20.15
epoch 5/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.20578471002146
The cross-entropy loss on the validation set 2.2061707616279413
The accuracy on the train set 21.514
The accuracy on the validation set 21.74
epoch 6/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.187430708322737
The cross-entropy loss on the validation set 2.187927819430271
The accuracy on the train set 22.96
The accuracy on the validation set 23.31
epoch 7/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.169737159177325
The cross-entropy loss on the validation set 2.1703586043639365
The accuracy on the train set 24.11
The accuracy on the validation set 24.56
epoch 8/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.152662558895526
The cross-entropy loss on the validation set 2.153369954010073
The accuracy on the train set 24.938
The accuracy on the validation set 25.45
epoch 9/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.136261135784886
The cross-entropy loss on the validation set 2.1370492568317325
The accuracy on the train set 25.736
The accuracy on the validation set 26.16
epoch 10/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.1205834681181837
The cross-entropy loss on the validation set 2.1214404122193167
The accuracy on the train set 26.484
The accuracy on the validation set 26.69
epoch 11/40
```

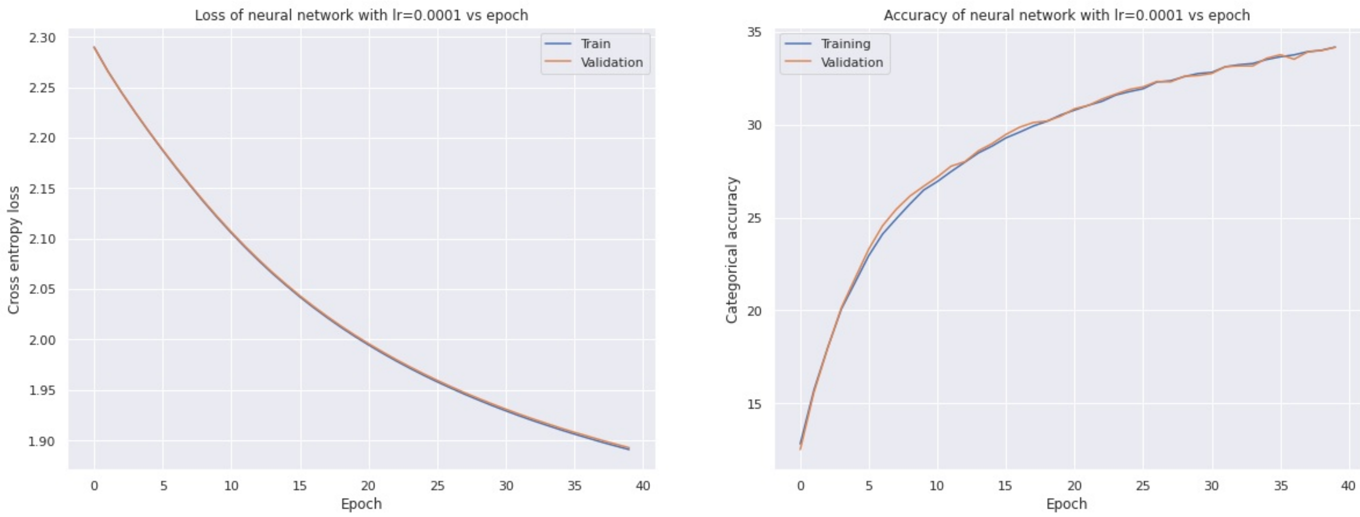
```
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.1056547057863653
The cross-entropy loss on the validation set 2.106573250304981
The accuracy on the train set 26.946
The accuracy on the validation set 27.19
epoch 12/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.0915093906048026
The cross-entropy loss on the validation set 2.092459998127457
The accuracy on the train set 27.48
The accuracy on the validation set 27.77
epoch 13/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.078121854708453
The cross-entropy loss on the validation set 2.0791599655660904
The accuracy on the train set 27.984
The accuracy on the validation set 28.01
epoch 14/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.0654499069082646
The cross-entropy loss on the validation set 2.0664986340229663
The accuracy on the train set 28.488
The accuracy on the validation set 28.6
epoch 15/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.0536003025963385
The cross-entropy loss on the validation set 2.0546905346087514
The accuracy on the train set 28.856
The accuracy on the validation set 28.99
epoch 16/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.042224725975728
The cross-entropy loss on the validation set 2.0433733608309197
The accuracy on the train set 29.294
The accuracy on the validation set 29.48
epoch 17/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.031614996080048
The cross-entropy loss on the validation set 2.0327901653870564
The accuracy on the train set 29.602
The accuracy on the validation set 29.87
epoch 18/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.0215995449917674
The cross-entropy loss on the validation set 2.0228381582571577
The accuracy on the train set 29.934
The accuracy on the validation set 30.12
epoch 19/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.012083458764872
The cross-entropy loss on the validation set 2.0133180092786924
The accuracy on the train set 30.188
The accuracy on the validation set 30.2
epoch 20/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 2.0031367605997406
The cross-entropy loss on the validation set 2.00439288087504
The accuracy on the train set 30.53
The accuracy on the validation set 30.47
epoch 21/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.994676109211515
The cross-entropy loss on the validation set 1.9959827800169012
The accuracy on the train set 30.788
The accuracy on the validation set 30.86
epoch 22/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.9866037861491816
The cross-entropy loss on the validation set 1.987950205308492
The accuracy on the train set 31.046
The accuracy on the validation set 31.04
epoch 23/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.9789352588905726
The cross-entropy loss on the validation set 1.9802830240343416
The accuracy on the train set 31.262
The accuracy on the validation set 31.38
epoch 24/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.971664878461959
The cross-entropy loss on the validation set 1.9730566495757575
The accuracy on the train set 31.6
The accuracy on the validation set 31.65
```

```
epoch 25/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.9647376814922184
The cross-entropy loss on the validation set 1.9661754822691018
The accuracy on the train set 31.784
The accuracy on the validation set 31.9
epoch 26/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.9581473043790687
The cross-entropy loss on the validation set 1.9596039183667537
The accuracy on the train set 31.938
The accuracy on the validation set 32.04
epoch 27/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.951901305245281
The cross-entropy loss on the validation set 1.9533552889862502
The accuracy on the train set 32.302
The accuracy on the validation set 32.33
epoch 28/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.9458463565657262
The cross-entropy loss on the validation set 1.947358190914334
The accuracy on the train set 32.368
The accuracy on the validation set 32.31
epoch 29/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.940163930544109
The cross-entropy loss on the validation set 1.9417194969451663
The accuracy on the train set 32.592
The accuracy on the validation set 32.61
epoch 30/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.9346313491442844
The cross-entropy loss on the validation set 1.9362585355801094
The accuracy on the train set 32.756
The accuracy on the validation set 32.65
epoch 31/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.9294174025050732
The cross-entropy loss on the validation set 1.9310323377503085
The accuracy on the train set 32.82
The accuracy on the validation set 32.76
epoch 32/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.924358633263011
The cross-entropy loss on the validation set 1.9260145021095403
The accuracy on the train set 33.124
The accuracy on the validation set 33.13
epoch 33/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.9195560358688148
The cross-entropy loss on the validation set 1.9212219141692546
The accuracy on the train set 33.232
The accuracy on the validation set 33.17
epoch 34/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.9150099127930391
The cross-entropy loss on the validation set 1.9167369431602155
The accuracy on the train set 33.298
The accuracy on the validation set 33.17
epoch 35/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.9105633556513288
The cross-entropy loss on the validation set 1.9122474799270057
The accuracy on the train set 33.514
The accuracy on the validation set 33.58
epoch 36/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.9063411328137363
The cross-entropy loss on the validation set 1.9080744447370634
The accuracy on the train set 33.658
The accuracy on the validation set 33.77
epoch 37/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.9022912745451528
The cross-entropy loss on the validation set 1.9041366384976817
The accuracy on the train set 33.762
The accuracy on the validation set 33.53
epoch 38/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.8983125243181345
The cross-entropy loss on the validation set 1.9001308172933593
The accuracy on the train set 33.936
```

```
The accuracy on the validation set 33.92
epoch 39/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.8945616284154687
The cross-entropy loss on the validation set 1.8964015178107405
The accuracy on the train set 34.01
The accuracy on the validation set 34.0
epoch 40/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.8909432432528073
The cross-entropy loss on the validation set 1.8927806530123097
The accuracy on the train set 34.18
The accuracy on the validation set 34.17
The ellapsed time for 40 epochs with lr=0.0001 is: 21
```

In [55]:

```
### PLOT LEARNING CURVES FOR MLP ON 40 EPOCHS FOR LEARNING RATE 0.0001 ###
fig = plt.figure(figsize=(20, 7))

sns.set()

fig.add_subplot(121)
plt.plot(L_2, label="Train")
plt.plot(L_val_2, label="Validation")
plt.xlabel("Epoch")
plt.ylabel("Cross entropy loss")
plt.legend()
plt.title("Loss of neural network with lr=0.0001 vs epoch")

fig.add_subplot(122)
plt.plot(A_2, label="Training")
plt.plot(A_val_2, label="Validation")
plt.xlabel("Epoch")
plt.ylabel("Categorical accuracy")
plt.legend()
plt.title("Accuracy of neural network with lr=0.0001 vs epoch")

plt.show()
```



In [56]:

```
print("The best loss for learning rate 0.0001 on 40 epochs is:", min(L_val_2))
print("The best validation accuracy for learning rate 0.0001 on 40 epochs is:", max(A_val_2))
```

```
The best loss for learning rate 0.0001 on 40 epochs is: 1.8927806530123097
The best validation accuracy for learning rate 0.0001 on 40 epochs is: 34.17
```

Discussion of convergence and performance

In this case, we can see that the train and validation loss are following exactly the same path. It shows a clear sign of underfitting. Similarly for the train and validation accuracy, which gets to a max of 35%. We can see that this model isn't converging fast enough, the learning rate is most probably to slow. Here we don't see the noisiness expected from SGD because the increment is so small (0.0001) that we could only see it by zooming in. Clearly the loss curve is still descending and if we were to increase the number of epochs, we would probably be able to see that, but it will increase the overall training time, which is a downside as we are looking for something both accurate and rapid: efficiency.

LEARNING RATE = 0.1

In [57]:

```
### TRAIN MLP ON 40 EPOCHS WITH LEARNING RATE 0.1 ###
```

```python
# begin timing
import time
start = time.time()

# define model training parameters
epochs = 40
lr = 0.1
batches = 390

L_3, L_val_3, A_3, A_val_3 = train_MLP(epochs, lr, batches, x_train, y_train)

# end timing
end = time.time()
time3 = int((end-start)//60)
print("The ellapsed time for 40 epochs with lr=0.1 is:", time3)
```

```
epoch 1/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.8601089295150681
The cross-entropy loss on the validation set 1.865940829731978
The accuracy on the train set 32.602
The accuracy on the validation set 32.77
epoch 2/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.6824349075407983
The cross-entropy loss on the validation set 1.685877713619952
The accuracy on the train set 40.246
The accuracy on the validation set 40.51
epoch 3/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.718928979381813
The cross-entropy loss on the validation set 1.7326459959578522
The accuracy on the train set 37.596
The accuracy on the validation set 37.56
epoch 4/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.6063371672201814
The cross-entropy loss on the validation set 1.634068699378362
The accuracy on the train set 41.496
The accuracy on the validation set 40.58
epoch 5/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.6233114021624415
The cross-entropy loss on the validation set 1.6532619744365937
The accuracy on the train set 42.11
The accuracy on the validation set 40.84
epoch 6/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.4956137920716557
The cross-entropy loss on the validation set 1.5397214413515317
The accuracy on the train set 45.906
The accuracy on the validation set 44.38
epoch 7/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.4408197015719404
The cross-entropy loss on the validation set 1.4937751681767175
The accuracy on the train set 48.658
The accuracy on the validation set 46.94
epoch 8/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.4794127543776652
The cross-entropy loss on the validation set 1.534751326023053
The accuracy on the train set 46.924
The accuracy on the validation set 44.69
epoch 9/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.4525311811070478
The cross-entropy loss on the validation set 1.5244137746822335
The accuracy on the train set 48.178
The accuracy on the validation set 45.53
epoch 10/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.3495284779065297
The cross-entropy loss on the validation set 1.446462116113535
The accuracy on the train set 51.928
The accuracy on the validation set 48.7
epoch 11/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.3638548069980263
The cross-entropy loss on the validation set 1.4822195692780693
The accuracy on the train set 50.828
The accuracy on the validation set 47.22
epoch 12/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.3163966109358727
```

```
The cross-entropy loss on the validation set 1.4415824889996387
The accuracy on the train set 52.81
The accuracy on the validation set 49.17
epoch 13/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.295097453902336
The cross-entropy loss on the validation set 1.4440068855708856
The accuracy on the train set 53.848
The accuracy on the validation set 49.65
epoch 14/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.2360418345434787
The cross-entropy loss on the validation set 1.4050655803090293
The accuracy on the train set 56.176
The accuracy on the validation set 50.35
epoch 15/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.2568693381726457
The cross-entropy loss on the validation set 1.444585448059512
The accuracy on the train set 55.376
The accuracy on the validation set 49.12
epoch 16/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.1918126424692217
The cross-entropy loss on the validation set 1.4133875313971538
The accuracy on the train set 56.928
The accuracy on the validation set 49.71
epoch 17/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.1884096268103812
The cross-entropy loss on the validation set 1.4354365397403925
The accuracy on the train set 58.022
The accuracy on the validation set 50.05
epoch 18/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.171565828265683
The cross-entropy loss on the validation set 1.4451002830544415
The accuracy on the train set 57.532
The accuracy on the validation set 49.59
epoch 19/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.168916727336716
The cross-entropy loss on the validation set 1.4560359887474263
The accuracy on the train set 58.576
The accuracy on the validation set 49.54
epoch 20/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.0629282918385938
The cross-entropy loss on the validation set 1.3878909137374766
The accuracy on the train set 62.468
The accuracy on the validation set 51.39
epoch 21/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.175997340424714
The cross-entropy loss on the validation set 1.515957265416751
The accuracy on the train set 57.756
The accuracy on the validation set 47.72
epoch 22/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.0091823081212994
The cross-entropy loss on the validation set 1.4088860508177057
The accuracy on the train set 64.648
The accuracy on the validation set 51.07
epoch 23/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.9933161366546706
The cross-entropy loss on the validation set 1.4272082856233816
The accuracy on the train set 64.49
The accuracy on the validation set 50.89
epoch 24/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.9707072618392238
The cross-entropy loss on the validation set 1.430722399212921
The accuracy on the train set 65.696
The accuracy on the validation set 51.0
epoch 25/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.8886948779273212
The cross-entropy loss on the validation set 1.3930296399405864
The accuracy on the train set 68.504
The accuracy on the validation set 52.6
epoch 26/40
[25%][50%][75%][100%]
```

```
The cross-entropy loss on the train set 0.8976188923887141
The cross-entropy loss on the validation set 1.4398267258585382
The accuracy on the train set 68.54
The accuracy on the validation set 51.53
epoch 27/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.8394400577272312
The cross-entropy loss on the validation set 1.4458945724520047
The accuracy on the train set 70.212
The accuracy on the validation set 51.5
epoch 28/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.8446018955554471
The cross-entropy loss on the validation set 1.4882611732810507
The accuracy on the train set 70.244
The accuracy on the validation set 51.52
epoch 29/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.7755571448931785
The cross-entropy loss on the validation set 1.4868813573589077
The accuracy on the train set 72.372
The accuracy on the validation set 52.0
epoch 30/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.8235666271447232
The cross-entropy loss on the validation set 1.5605018877449277
The accuracy on the train set 70.586
The accuracy on the validation set 50.35
epoch 31/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.7449334921139424
The cross-entropy loss on the validation set 1.539850069354663
The accuracy on the train set 73.586
The accuracy on the validation set 51.14
epoch 32/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.6862584638541959
The cross-entropy loss on the validation set 1.547927568518779
The accuracy on the train set 76.198
The accuracy on the validation set 51.79
epoch 33/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.7552991388202592
The cross-entropy loss on the validation set 1.6528180156912313
The accuracy on the train set 72.412
The accuracy on the validation set 49.67
epoch 34/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.7574725513928583
The cross-entropy loss on the validation set 1.6852964435293531
The accuracy on the train set 73.564
The accuracy on the validation set 50.57
epoch 35/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.6166240526388521
The cross-entropy loss on the validation set 1.6320102614792875
The accuracy on the train set 78.098
The accuracy on the validation set 50.78
epoch 36/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.6326748591917006
The cross-entropy loss on the validation set 1.7015237368938414
The accuracy on the train set 77.736
The accuracy on the validation set 50.65
epoch 37/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.5546132557828425
The cross-entropy loss on the validation set 1.700948728035943
The accuracy on the train set 80.596
The accuracy on the validation set 50.55
epoch 38/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.5747212887526594
The cross-entropy loss on the validation set 1.7845900536245611
The accuracy on the train set 79.342
The accuracy on the validation set 49.65
epoch 39/40
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.5039263669435148
The cross-entropy loss on the validation set 1.7748984811394446
The accuracy on the train set 82.476
The accuracy on the validation set 51.27
epoch 40/40
```

```
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.4444687787727113
The cross-entropy loss on the validation set 1.805436683863818
The accuracy on the train set 84.724
The accuracy on the validation set 51.35
The ellapsed time for 40 epochs with lr=0.1 is: 23
```

In [58]:

```
### PLOT LEARNING CURVES FOR MLP ON 40 EPOCHS FOR LEARNING RATE 0.1 ###
fig = plt.figure(figsize=(20, 7))

sns.set()

fig.add_subplot(121)
plt.plot(L_3, label="Train")
plt.plot(L_val_3, label="Validation")
plt.xlabel("Epoch")
plt.ylabel("Cross entropy loss")
plt.legend()
plt.title("Loss of neural network with lr=0.1 vs epoch")

fig.add_subplot(122)
plt.plot(A_3, label="Training")
plt.plot(A_val_3, label="Validation")
plt.xlabel("Epoch")
plt.ylabel("Categorical accuracy")
plt.legend()
plt.title("Accuracy of neural network with lr=0.1 vs epoch")

plt.show()
```

In [59]:

```
print("The best loss for learning rate 0.1 on 40 epochs is:", min(L_val_3))
print("The best validation accuracy for learning rate 0.1 on 40 epochs is:", max(A_val_3))
```

```
The best loss for learning rate 0.1 on 40 epochs is: 1.3878909137374766
The best validation accuracy for learning rate 0.1 on 40 epochs is: 52.6
```

Discussion of convergence and performance

In this case, we can see that while the training loss seems to continue its descent in terms of loss, there is some clear overfitting happening from epoch number 10 onwards on the validation loss. From then on, we can also obseve a decreasing validation accuracy. This could mean that our learning rate is too fast, and our model isn't picking all the variability and dropping important information. Indeed our max validatioin accuracy is around 52%, although it is better than for the learning rate 0.0001, it is clearly overfitting. Also there is lots of noisiness from the SGD but that's expected from the randomness aspect of the optimization. So this model is no good, it should be discarded.

In [60]:

```
### PLOT LEARNING CURVES FOR MLP ON 40 EPOCHS FOR LEARNING RATE 0.1, 0.01 & 0.0001 ###
fig = plt.figure(figsize=(20, 7))

sns.set()

fig.add_subplot(121)
plt.plot(L_1, color='r', label="Train for lr=0.01")
plt.plot(L_val_1, color='r', linestyle='dotted', label="Validation for lr=0.01")
plt.plot(L_2, color='g', label="Train for lr=0.0001")
plt.plot(L_val_2, color='g', linestyle='dotted', label="Validation for lr=0.0001")
plt.plot(L_3, color='b', label="Train for lr=0.1")
plt.plot(L_val_3, color='b', linestyle='dotted', label="Validation for lr=0.1")
plt.xlabel("Epoch")
```

```
plt.ylabel("Cross entropy loss")
plt.legend()
plt.title("Loss of neural network with different learning rates vs epoch")

fig.add_subplot(122)
plt.plot(A_1, color='r', label="Train for lr=0.01")
plt.plot(A_val_1, color='r', linestyle='dotted', label="Validation for lr=0.01")
plt.plot(A_2, color='g', label="Train for lr=0.0001")
plt.plot(A_val_2, color='g', linestyle='dotted', label="Validation for lr=0.0001")
plt.plot(A_3, color='b', label="Train for lr=0.1")
plt.plot(A_val_3, color='b', linestyle='dotted', label="Validation for lr=0.1")
plt.xlabel("Epoch")
plt.ylabel("Categorical accuracy")
plt.legend()
plt.title("Accuracy of neural network with different learning rates vs epoch")

plt.show()
```

Discussion of the three learning rates

- Learning rate=0.1

  loss: training loss is the best out of the three (around 1) however validation loss indicates overfitting over epoch n20 --> discard model

  accuracy: training accuracy is the best out of the three (around 83%) however as said above, strong overfitting which makes the val accuracy around 50% --> discard model because loss is going up intensely

- Learning rate=0.01

  loss: decreasing curve for both validation and training. Validation is starting to slow down, so any idea of increasing the number of epochs to see if accuracy is increased could be explore, however risk of overfitting

  accuracy: validation accuracy goes beyond 50%, which is quite satisfying compared to the other models --> keep model

- Learning rate=0.0001

  loss: both training and validation loss are going down very slowly which indicates that the model has a learning rate that is too slow: underfitting

  accuracy: similarly, accuracy is going up but not fast enough due to the slowness of the model --> increase learning rate or extend number of epochs

1.1.3 Train MLP on 80 epochs with learning rate 0.01

In [61]:

```
### TRAIN MLP ON 80 EPOCHS WITH LEARNING RATE 0.01 ###

# begin timing
import time
start = time.time()

# define model training parameters
epochs = 80
lr = 0.01
batches = 390

L_4, L_val_4, A_4, A_val_4 = train_MLP(epochs, lr, batches, x_train, y_train)
```

```python
# end timing
end = time.time()
time4 = int((end-start)//60)
print("The ellapsed time for 80 epochs with lr=0.01 is:", time4)
```

```
epoch 1/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.8253066864581189
The cross-entropy loss on the validation set 1.83001506925519
The accuracy on the train set 34.81
The accuracy on the validation set 34.38
epoch 2/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.8179892455682882
The cross-entropy loss on the validation set 1.821426230284612
The accuracy on the train set 34.512
The accuracy on the validation set 34.52
epoch 3/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.7383465089244101
The cross-entropy loss on the validation set 1.749193166691496
The accuracy on the train set 38.16
The accuracy on the validation set 38.41
epoch 4/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.8170896557361615
The cross-entropy loss on the validation set 1.8356792284603816
The accuracy on the train set 35.566
The accuracy on the validation set 35.17
epoch 5/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.63250670451363
The cross-entropy loss on the validation set 1.6462553675803087
The accuracy on the train set 42.25
The accuracy on the validation set 42.16
epoch 6/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.60876373724648
The cross-entropy loss on the validation set 1.6241327356577813
The accuracy on the train set 43.808
The accuracy on the validation set 43.59
epoch 7/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.5711798719996761
The cross-entropy loss on the validation set 1.5908815499359714
The accuracy on the train set 44.896
The accuracy on the validation set 44.36
epoch 8/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.6023314557130102
The cross-entropy loss on the validation set 1.6305832217344614
The accuracy on the train set 43.61
The accuracy on the validation set 43.06
epoch 9/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.5232097816432133
The cross-entropy loss on the validation set 1.5516675881419562
The accuracy on the train set 46.084
The accuracy on the validation set 45.15
epoch 10/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.5466539973087687
The cross-entropy loss on the validation set 1.5752738567999316
The accuracy on the train set 45.142
The accuracy on the validation set 44.23
epoch 11/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.4940980622781643
The cross-entropy loss on the validation set 1.5323611700090594
The accuracy on the train set 47.196
The accuracy on the validation set 46.45
epoch 12/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.4905687207855125
The cross-entropy loss on the validation set 1.5279191318275502
The accuracy on the train set 47.478
The accuracy on the validation set 45.97
epoch 13/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.5257971114562767
The cross-entropy loss on the validation set 1.5647341322759127
The accuracy on the train set 45.852
The accuracy on the validation set 44.53
epoch 14/80
```

```
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.45349252370238
The cross-entropy loss on the validation set 1.4980720253038478
The accuracy on the train set 48.824
The accuracy on the validation set 46.9
epoch 15/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.41118181697854
The cross-entropy loss on the validation set 1.4612401882610602
The accuracy on the train set 50.254
The accuracy on the validation set 48.74
epoch 16/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.4581689736987111
The cross-entropy loss on the validation set 1.5071174386543287
The accuracy on the train set 48.404
The accuracy on the validation set 47.02
epoch 17/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.4195072065334995
The cross-entropy loss on the validation set 1.4794999037583212
The accuracy on the train set 49.564
The accuracy on the validation set 47.66
epoch 18/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.3654660233239087
The cross-entropy loss on the validation set 1.426442571438773
The accuracy on the train set 51.634
The accuracy on the validation set 49.9
epoch 19/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.345887924450374
The cross-entropy loss on the validation set 1.4108610684123792
The accuracy on the train set 52.666
The accuracy on the validation set 50.29
epoch 20/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.3716081474625814
The cross-entropy loss on the validation set 1.447263650612091
The accuracy on the train set 51.128
The accuracy on the validation set 48.3
epoch 21/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.3583506182935037
The cross-entropy loss on the validation set 1.4357936736363153
The accuracy on the train set 51.708
The accuracy on the validation set 48.86
epoch 22/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.3119260984634837
The cross-entropy loss on the validation set 1.3948121329960406
The accuracy on the train set 53.238
The accuracy on the validation set 50.52
epoch 23/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.356676159067418
The cross-entropy loss on the validation set 1.4487931207649054
The accuracy on the train set 51.662
The accuracy on the validation set 47.92
epoch 24/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.302563992749256
The cross-entropy loss on the validation set 1.3926578243515786
The accuracy on the train set 53.908
The accuracy on the validation set 50.74
epoch 25/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.308504346454284
The cross-entropy loss on the validation set 1.4053396673921636
The accuracy on the train set 53.554
The accuracy on the validation set 50.16
epoch 26/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.272986220788347
The cross-entropy loss on the validation set 1.3746929656921107
The accuracy on the train set 54.796
The accuracy on the validation set 51.15
epoch 27/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.3056693321817348
The cross-entropy loss on the validation set 1.41276758550762
The accuracy on the train set 52.832
The accuracy on the validation set 50.02
```

```
epoch 28/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.2847749031383318
The cross-entropy loss on the validation set 1.397444508199362
The accuracy on the train set 54.416
The accuracy on the validation set 50.81
epoch 29/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.2608043302670242
The cross-entropy loss on the validation set 1.3842428723555829
The accuracy on the train set 55.476
The accuracy on the validation set 50.09
epoch 30/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.2772326522822108
The cross-entropy loss on the validation set 1.4101996838797062
The accuracy on the train set 54.504
The accuracy on the validation set 49.62
epoch 31/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.246252124124072
The cross-entropy loss on the validation set 1.3757149441959737
The accuracy on the train set 55.974
The accuracy on the validation set 51.27
epoch 32/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.2132160626469102
The cross-entropy loss on the validation set 1.3517359730461371
The accuracy on the train set 56.952
The accuracy on the validation set 51.72
epoch 33/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.2896331323979966
The cross-entropy loss on the validation set 1.4335668069799758
The accuracy on the train set 53.832
The accuracy on the validation set 49.26
epoch 34/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.1849328192426936
The cross-entropy loss on the validation set 1.3342903132701653
The accuracy on the train set 58.1
The accuracy on the validation set 53.04
epoch 35/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.224994113022382
The cross-entropy loss on the validation set 1.3822000662512162
The accuracy on the train set 56.322
The accuracy on the validation set 51.03
epoch 36/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.1708113794337316
The cross-entropy loss on the validation set 1.3336072580949143
The accuracy on the train set 58.754
The accuracy on the validation set 52.54
epoch 37/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.159219021460143
The cross-entropy loss on the validation set 1.3246573974247338
The accuracy on the train set 59.066
The accuracy on the validation set 53.18
epoch 38/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.1434559539025633
The cross-entropy loss on the validation set 1.3154337590848144
The accuracy on the train set 59.908
The accuracy on the validation set 53.55
epoch 39/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.192950351492275
The cross-entropy loss on the validation set 1.3848128799883326
The accuracy on the train set 57.16
The accuracy on the validation set 50.65
epoch 40/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.1566649891846135
The cross-entropy loss on the validation set 1.3484993087904018
The accuracy on the train set 59.064
The accuracy on the validation set 52.21
epoch 41/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.1462259119526772
The cross-entropy loss on the validation set 1.3423243260215356
The accuracy on the train set 59.15
```

```
The accuracy on the validation set 51.86
epoch 42/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.2011575075534804
The cross-entropy loss on the validation set 1.4078985365459493
The accuracy on the train set 57.274
The accuracy on the validation set 50.34
epoch 43/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.1483479027885393
The cross-entropy loss on the validation set 1.357117828136196
The accuracy on the train set 59.286
The accuracy on the validation set 52.25
epoch 44/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.1600490133918027
The cross-entropy loss on the validation set 1.3789961366442987
The accuracy on the train set 59.05
The accuracy on the validation set 51.56
epoch 45/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.1001708732357132
The cross-entropy loss on the validation set 1.3226824902667387
The accuracy on the train set 60.96
The accuracy on the validation set 53.07
epoch 46/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.1134923357587019
The cross-entropy loss on the validation set 1.3481618630139178
The accuracy on the train set 60.534
The accuracy on the validation set 52.78
epoch 47/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.0882727591980004
The cross-entropy loss on the validation set 1.327130920018443
The accuracy on the train set 61.546
The accuracy on the validation set 53.55
epoch 48/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.0766235778048396
The cross-entropy loss on the validation set 1.3294993333535363
The accuracy on the train set 61.84
The accuracy on the validation set 52.52
epoch 49/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.0745098351055866
The cross-entropy loss on the validation set 1.3348926548929323
The accuracy on the train set 62.022
The accuracy on the validation set 53.44
epoch 50/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.0291476964965531
The cross-entropy loss on the validation set 1.2968034700832374
The accuracy on the train set 64.01
The accuracy on the validation set 54.08
epoch 51/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.035926311655985
The cross-entropy loss on the validation set 1.3148570707307048
The accuracy on the train set 63.392
The accuracy on the validation set 53.43
epoch 52/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.0328789560305984
The cross-entropy loss on the validation set 1.3213357388574347
The accuracy on the train set 63.782
The accuracy on the validation set 53.39
epoch 53/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.0201443425459285
The cross-entropy loss on the validation set 1.3192100090882184
The accuracy on the train set 64.31
The accuracy on the validation set 54.39
epoch 54/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.0270884017994986
The cross-entropy loss on the validation set 1.330858154455412
The accuracy on the train set 63.812
The accuracy on the validation set 53.59
epoch 55/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.0362650199284724
The cross-entropy loss on the validation set 1.3606208294194142
```

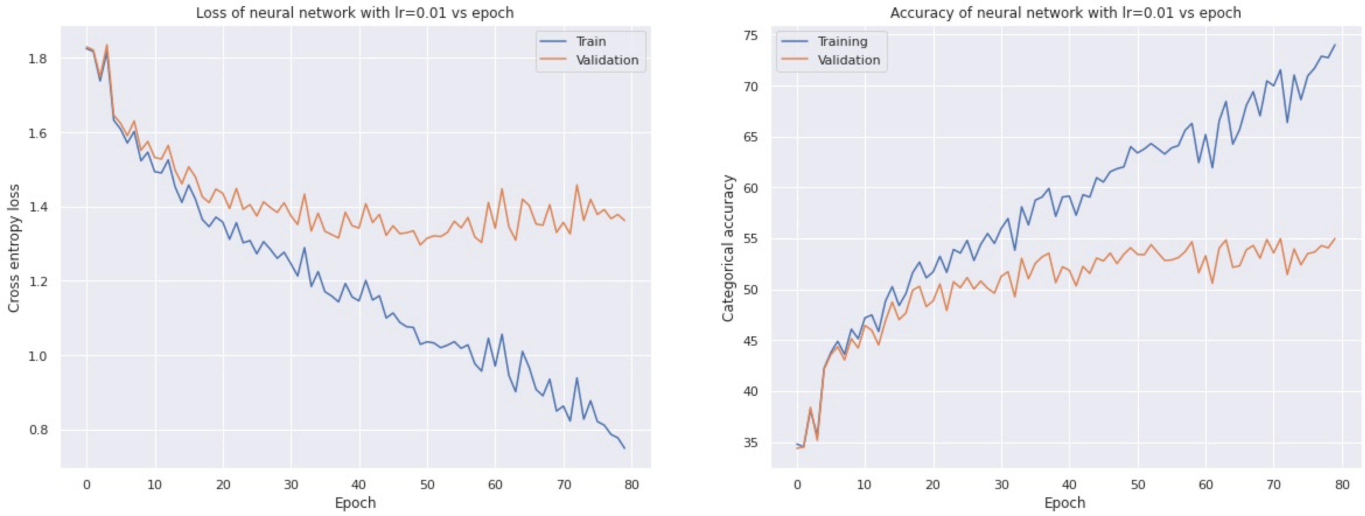
```
The accuracy on the train set 63.284
The accuracy on the validation set 52.83
epoch 56/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.0184535779972115
The cross-entropy loss on the validation set 1.34275194838202
The accuracy on the train set 63.882
The accuracy on the validation set 52.88
epoch 57/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.02789028086027
The cross-entropy loss on the validation set 1.3705206087871744
The accuracy on the train set 64.106
The accuracy on the validation set 53.11
epoch 58/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.9771126492139119
The cross-entropy loss on the validation set 1.3181265221058154
The accuracy on the train set 65.616
The accuracy on the validation set 53.7
epoch 59/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.9570284882132839
The cross-entropy loss on the validation set 1.3033595845879913
The accuracy on the train set 66.294
The accuracy on the validation set 54.67
epoch 60/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.0457272973837621
The cross-entropy loss on the validation set 1.4109613341720488
The accuracy on the train set 62.442
The accuracy on the validation set 51.62
epoch 61/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.9705435885632595
The cross-entropy loss on the validation set 1.341854339715949
The accuracy on the train set 65.204
The accuracy on the validation set 53.3
epoch 62/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.0562700990966627
The cross-entropy loss on the validation set 1.447862974642749
The accuracy on the train set 61.936
The accuracy on the validation set 50.6
epoch 63/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.9461290275524106
The cross-entropy loss on the validation set 1.3460436060257681
The accuracy on the train set 66.55
The accuracy on the validation set 54.04
epoch 64/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.901614478625767
The cross-entropy loss on the validation set 1.309590328248644
The accuracy on the train set 68.45
The accuracy on the validation set 54.85
epoch 65/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 1.0100533561006675
The cross-entropy loss on the validation set 1.4199935243725155
The accuracy on the train set 64.252
The accuracy on the validation set 52.16
epoch 66/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.966416693271911
The cross-entropy loss on the validation set 1.4024199223036657
The accuracy on the train set 65.688
The accuracy on the validation set 52.31
epoch 67/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.9076906027371962
The cross-entropy loss on the validation set 1.3529177551431835
The accuracy on the train set 68.082
The accuracy on the validation set 53.88
epoch 68/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.8907184344449914
The cross-entropy loss on the validation set 1.3497785547490093
The accuracy on the train set 69.406
The accuracy on the validation set 54.3
epoch 69/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.9353251069052415
```

```
The cross-entropy loss on the validation set 1.4051796395467475
The accuracy on the train set 67.038
The accuracy on the validation set 53.06
epoch 70/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.8494141063982865
The cross-entropy loss on the validation set 1.330376789775254
The accuracy on the train set 70.466
The accuracy on the validation set 54.89
epoch 71/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.8630952990986984
The cross-entropy loss on the validation set 1.3573007787950415
The accuracy on the train set 69.982
The accuracy on the validation set 53.57
epoch 72/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.8227493898950721
The cross-entropy loss on the validation set 1.3265064745416997
The accuracy on the train set 71.55
The accuracy on the validation set 54.97
epoch 73/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.9386883254070831
The cross-entropy loss on the validation set 1.4581821710680258
The accuracy on the train set 66.378
The accuracy on the validation set 51.44
epoch 74/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.8276116883312457
The cross-entropy loss on the validation set 1.3627676412956926
The accuracy on the train set 71.042
The accuracy on the validation set 53.96
epoch 75/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.8773484517883361
The cross-entropy loss on the validation set 1.419417512807193
The accuracy on the train set 68.618
The accuracy on the validation set 52.4
epoch 76/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.8212915493420548
The cross-entropy loss on the validation set 1.378707500161246
The accuracy on the train set 70.958
The accuracy on the validation set 53.51
epoch 77/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.811739417139147
The cross-entropy loss on the validation set 1.3920695669712633
The accuracy on the train set 71.718
The accuracy on the validation set 53.65
epoch 78/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.7866468164419161
The cross-entropy loss on the validation set 1.367442112767007
The accuracy on the train set 72.886
The accuracy on the validation set 54.29
epoch 79/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.77757216274712
The cross-entropy loss on the validation set 1.3787554045803398
The accuracy on the train set 72.748
The accuracy on the validation set 54.07
epoch 80/80
[25%][50%][75%][100%]
The cross-entropy loss on the train set 0.7491908010051735
The cross-entropy loss on the validation set 1.3625782637019892
The accuracy on the train set 74.016
The accuracy on the validation set 54.99
The ellapsed time for 80 epochs with lr=0.01 is: 44
```

In [62]:

```python
### PLOT LEARNING CURVES FOR MLP ON 80 EPOCHS FOR LEARNING RATE 0.01 ###
fig = plt.figure(figsize=(20, 7))

sns.set()

fig.add_subplot(121)
plt.plot(L_4, label="Train")
plt.plot(L_val_4, label="Validation")
plt.xlabel("Epoch")
plt.ylabel("Cross entropy loss")
plt.legend()
```

```
plt.title("Loss of neural network with lr=0.01 vs epoch")

fig.add_subplot(122)
plt.plot(A_4, label="Training")
plt.plot(A_val_4, label="Validation")
plt.xlabel("Epoch")
plt.ylabel("Categorical accuracy")
plt.legend()
plt.title("Accuracy of neural network with lr=0.01 vs epoch")

plt.show()
```



In [63]:
```
print("The best loss for learning rate 0.01 on 80 epochs is:", min(L_val_4))
print("The best validation accuracy for learning rate 0.01 on 80 epochs is:", max(A_val_4))
```

```
The best loss for learning rate 0.01 on 80 epochs is: 1.2968034700832374
The best validation accuracy for learning rate 0.01 on 80 epochs is: 54.99
```

Comment on accuracy and convergence

As mentioned in 1.1.1, the learning rate=0.01 presents some nice resulsts over 40 epochs with a still decreasing loss, so we wanted to explore what an increase in number of epochs would do to our model, bewaring of overfitting. What was supposed to happen, happened: there is clear overfitting going on after epoch 50, where the validation loss increases, which also makes our validation accuracy stagnate around the same value that we get from epoch 50. Hence we could re implement a training with 50 epochs for example for two reasons:

1. we saw earlier that this was the best learning rate out of the three tested.
2. beyond 50 epochs, our model is overfitting

Now, let's do a table to summarize the main results from our models implementation:

In [64]:
```
Learning_Rate = [0.01, 0.0001, 0.1, 0.01]
Number_of_Epochs = [40, 40, 40, 80]
Loss_tot_train = [min(L_1), min(L_2), min(L_3), min(L_4)]
Loss_tot_test = [min(L_val_1), min(L_val_2), min(L_val_3), min(L_val_4)]
Acc_tot_train = [max(A_1), max(A_2), max(A_3), max(A_4)]
Acc_tot_test = [max(A_val_1), max(A_val_2), max(A_val_3), max(A_val_4)]
Time = [time1, time2, time3, time4]

comparison = np.array([Learning_Rate, Number_of_Epochs, Loss_tot_train, Loss_tot_test, Acc_tot_train, Acc_tot_tes
comparison = pd.DataFrame(comparison)
comparison.columns = ['Learning Rate','Number of Epochs', 'Training loss', 'Test loss', 'Training accuracy', 'Val
comparison.head()
```

Out[64]:

	Learning Rate	Number of Epochs	Training loss	Test loss	Training accuracy	Validation accuracy	Timing ellapsed
0	0.0100	40.0	1.155633	1.333278	59.870	53.08	22.0
1	0.0001	40.0	1.890943	1.892781	34.180	34.17	21.0
2	0.1000	40.0	0.444469	1.387891	84.724	52.60	23.0
3	0.0100	80.0	0.749191	1.296803	74.016	54.99	44.0

Explanation of results and comparison of models

Learning Rate

Clearly we can see that the setting of the learning rate should be a balance between our model learning not too fast, otherwise it skips important features (lr=0.1), but not too slow either because otherwise it takes to long to converge (lr=0.0001). This is the whole challenge of NN, to be able to balance the hyperparamaters to combine accuracy and rapidity in order to make an efficient model.\ In our example, we see that the best learning rate that we have tested is 0.01 based on the validation accuracy that it yields.

Number of epochs

Regarding the number of epochs, we can see that it's a similar idea as learning rate on the aspect that it's a balance because if too many, you risk overfitting like in the 4th model we tried (lr=0.01 on 80 epochs) and if too few you don't have time to improve you accuracy.

Cross-entropy loss

The cross entropy loss is minimal on the 80 epochs model but otherwise on the 40 epochs with learning rate 0.01, and from our comments beforehand, we have seen that it is a satisfying model.

Accuracy

The best validation accuracy that we get is for the learning rate 0.01 with 53% on 40 epochs and almost 55% on 80 epochs. However, as explained before, we've seen that 80 epochs presents strong signs of overfitting, and also for a twice as long training time, you only get an increase of 1% in accuracy. It isn't a good trade-off for complexity, timing and accuracy. Hence the first one we tried is the best compared to its peers.

Timing

Clearly, the model with twice the number of epochs takes twice the time because it's twice as long, otherwise all models with same number of epochs take all the same time to train. What would be interesting is to compare it to the training time from the tensorflow CNN as we will do in 1.2.4.

Conclusion

From these different models implemented, we have that the model with learning rate 0.01 trained on 40 epochs is the best relative to the other ones in terms of accuracy, loss and training time, and it isn't showing signs of uderfitting not overfitting.

1.2 Convolutional Neural Network (CNN)

In this task we are going to implement a Convolutional Neural Network according to the following architecture:

- 4 hidden layers (activated by ReLu):
  - 3 convolutional layers with 3x3 feature map and 2x2 max pooling layers in between them
  - 1 fully connected layer with 64 neurons
- 1 output layer (activated by softmax):
  - 10 ouputted neurons

We are using stochastic gradient descent (SGD) to fix the optimization with a cross-entropy loss function. Additionally, we are dividing our data into batches of 128 points, like in part 1.1, which gives us $\frac{50000}{128}=391$ batches to train on.

Note that for this task, we are activating GPU in the google colab as it makes our epochs run 25 times faster (from 50s to 2s) because tensorflow is optimised for Colab.

1.2.0 Load data

First and foremost, let's load the necessary libraries for this task, load the data and check the shape

In [65]:
```
### LOAD NECESSARY LIBRARIES ###

import tensorflow as tf
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from tensorflow.keras import datasets, layers, models
```

In [66]:
```
### LOAD DATA FUNCTION FROM FELIX ###

def load_data():
    (x_train, y_train), (x_val, y_val) = tf.keras.datasets.cifar10.load_data()
    x_train = x_train.astype('float32') / 255
    x_val = x_val.astype('float32') / 255

    #convert labels to categorical samples
    y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
```

```
    y_val = tf.keras.utils.to_categorical(y_val, num_classes=10)
    return ((x_train, y_train), (x_val, y_val))

(x_train, y_train), (x_val, y_val) = load_data()
```

In [67]:
```
### SHUFFLE DATA ###
shuffle = np.random.permutation(range(x_train.shape[0]))

x_train = x_train[shuffle]
y_train = y_train[shuffle]
```

In [68]:
```
### CHECK SHAPE OF DATA ###
print(x_train.shape)
print(y_train.shape)
print(x_val.shape)
print(y_val.shape)
```

```
(50000, 32, 32, 3)
(50000, 10)
(10000, 32, 32, 3)
(10000, 10)
```

1.2.1 Implement CNN

Now that our data is loaded, we can implement our first CNN with a learning rate of 0.1 for 40 epochs as follows:

In [69]:
```
### MODEL 1 WITH LR=0.1 ON 40 EPOCHS ###

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow import keras

def get_model(): # function as from CT
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)), # convolutional layer
        MaxPooling2D((2, 2)), # max pooling layer
        Conv2D(64, (3, 3), activation='relu'), # convolutional layer
        MaxPooling2D((2, 2)), # max pooling layer
        Conv2D(64, (3, 3), activation='relu'), # convolutional layer
        Flatten(),
        Dense(64, activation='relu'), # fully connected layer with 64 neurons
        Dense(10, activation='softmax') # output layer with 10 neurons
    ])

    opt = keras.optimizers.SGD(learning_rate=0.1) # select Stochastic Gradient Descent as optimisation method wit
    model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy']) # compile model with cros
    return model
```

In [70]:
```
### GET MODEL 1 SUMMARY ###
model = get_model()
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 32)        896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 64)          36928
_________________________________________________________________
flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
dense (Dense)                (None, 64)                65600
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 122,570
Trainable params: 122,570
Non-trainable params: 0
_________________________________________________________________
```

In [71]:
```
### FIT AND TIME MODEL ###
```

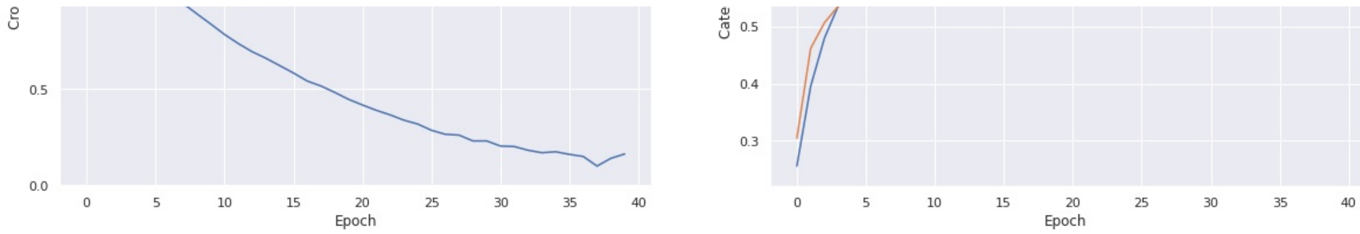
```python
import time
start = time.time()

history = model.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=40, batch_size=128) # turn on GPU

end = time.time()

time21 = int((end - start)/60)
print("The total running time for model with 0.1 lr on 40 epochs is:", time21)
```

```
Epoch 1/40
391/391 [==============================] - 18s 5ms/step - loss: 2.1752 - accuracy: 0.1988 - val_loss: 1.9454 - va
l_accuracy: 0.3046
Epoch 2/40
391/391 [==============================] - 2s 4ms/step - loss: 1.7723 - accuracy: 0.3676 - val_loss: 1.4917 - val
_accuracy: 0.4621
Epoch 3/40
391/391 [==============================] - 2s 4ms/step - loss: 1.4892 - accuracy: 0.4664 - val_loss: 1.3660 - val
_accuracy: 0.5071
Epoch 4/40
391/391 [==============================] - 2s 4ms/step - loss: 1.3165 - accuracy: 0.5294 - val_loss: 1.2791 - val
_accuracy: 0.5362
Epoch 5/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1929 - accuracy: 0.5794 - val_loss: 1.1269 - val
_accuracy: 0.5978
Epoch 6/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1100 - accuracy: 0.6110 - val_loss: 1.2438 - val
_accuracy: 0.5736
Epoch 7/40
391/391 [==============================] - 2s 4ms/step - loss: 1.0274 - accuracy: 0.6397 - val_loss: 1.0638 - val
_accuracy: 0.6204
Epoch 8/40
391/391 [==============================] - 2s 4ms/step - loss: 0.9574 - accuracy: 0.6648 - val_loss: 1.0702 - val
_accuracy: 0.6267
Epoch 9/40
391/391 [==============================] - 2s 4ms/step - loss: 0.8933 - accuracy: 0.6888 - val_loss: 1.0511 - val
_accuracy: 0.6375
Epoch 10/40
391/391 [==============================] - 2s 4ms/step - loss: 0.8314 - accuracy: 0.7102 - val_loss: 0.9921 - val
_accuracy: 0.6524
Epoch 11/40
391/391 [==============================] - 2s 4ms/step - loss: 0.7819 - accuracy: 0.7274 - val_loss: 0.9798 - val
_accuracy: 0.6657
Epoch 12/40
391/391 [==============================] - 2s 4ms/step - loss: 0.7240 - accuracy: 0.7468 - val_loss: 0.9524 - val
_accuracy: 0.6740
Epoch 13/40
391/391 [==============================] - 2s 4ms/step - loss: 0.6818 - accuracy: 0.7619 - val_loss: 0.9790 - val
_accuracy: 0.6750
Epoch 14/40
391/391 [==============================] - 2s 4ms/step - loss: 0.6538 - accuracy: 0.7718 - val_loss: 1.1456 - val
_accuracy: 0.6263
Epoch 15/40
391/391 [==============================] - 2s 4ms/step - loss: 0.6028 - accuracy: 0.7903 - val_loss: 1.0085 - val
_accuracy: 0.6673
Epoch 16/40
391/391 [==============================] - 2s 4ms/step - loss: 0.5600 - accuracy: 0.8045 - val_loss: 1.0303 - val
_accuracy: 0.6731
Epoch 17/40
391/391 [==============================] - 2s 4ms/step - loss: 0.5253 - accuracy: 0.8143 - val_loss: 1.1306 - val
_accuracy: 0.6501
Epoch 18/40
391/391 [==============================] - 2s 4ms/step - loss: 0.5020 - accuracy: 0.8218 - val_loss: 1.0458 - val
_accuracy: 0.6673
Epoch 19/40
391/391 [==============================] - 2s 4ms/step - loss: 0.4738 - accuracy: 0.8342 - val_loss: 1.0603 - val
_accuracy: 0.6819
Epoch 20/40
391/391 [==============================] - 2s 4ms/step - loss: 0.4202 - accuracy: 0.8532 - val_loss: 1.0595 - val
_accuracy: 0.6880
Epoch 21/40
391/391 [==============================] - 2s 4ms/step - loss: 0.3935 - accuracy: 0.8603 - val_loss: 1.1838 - val
_accuracy: 0.6778
Epoch 22/40
391/391 [==============================] - 2s 4ms/step - loss: 0.3701 - accuracy: 0.8703 - val_loss: 1.1261 - val
_accuracy: 0.6813
Epoch 23/40
391/391 [==============================] - 2s 4ms/step - loss: 0.3341 - accuracy: 0.8823 - val_loss: 1.2283 - val
_accuracy: 0.6727
Epoch 24/40
391/391 [==============================] - 2s 4ms/step - loss: 0.3066 - accuracy: 0.8926 - val_loss: 1.2630 - val
_accuracy: 0.6709
Epoch 25/40
391/391 [==============================] - 2s 4ms/step - loss: 0.2929 - accuracy: 0.8967 - val_loss: 1.3098 - val
```

```
_accuracy: 0.6697
Epoch 26/40
391/391 [==============================] - 2s 4ms/step - loss: 0.2586 - accuracy: 0.9077 - val_loss: 1.6614 - val
_accuracy: 0.6405
Epoch 27/40
391/391 [==============================] - 2s 4ms/step - loss: 0.2375 - accuracy: 0.9170 - val_loss: 1.4444 - val
_accuracy: 0.6671
Epoch 28/40
391/391 [==============================] - 2s 4ms/step - loss: 0.2291 - accuracy: 0.9195 - val_loss: 1.6011 - val
_accuracy: 0.6595
Epoch 29/40
391/391 [==============================] - 2s 4ms/step - loss: 0.2009 - accuracy: 0.9293 - val_loss: 1.6154 - val
_accuracy: 0.6578
Epoch 30/40
391/391 [==============================] - 2s 4ms/step - loss: 0.2002 - accuracy: 0.9302 - val_loss: 1.4997 - val
_accuracy: 0.6722
Epoch 31/40
391/391 [==============================] - 2s 4ms/step - loss: 0.1683 - accuracy: 0.9401 - val_loss: 1.7551 - val
_accuracy: 0.6576
Epoch 32/40
391/391 [==============================] - 2s 4ms/step - loss: 0.1782 - accuracy: 0.9374 - val_loss: 1.7586 - val
_accuracy: 0.6633
Epoch 33/40
391/391 [==============================] - 2s 4ms/step - loss: 0.1607 - accuracy: 0.9443 - val_loss: 1.7350 - val
_accuracy: 0.6747
Epoch 34/40
391/391 [==============================] - 2s 4ms/step - loss: 0.1419 - accuracy: 0.9510 - val_loss: 1.8119 - val
_accuracy: 0.6602
Epoch 35/40
391/391 [==============================] - 2s 4ms/step - loss: 0.1458 - accuracy: 0.9499 - val_loss: 1.9635 - val
_accuracy: 0.6638
Epoch 36/40
391/391 [==============================] - 2s 4ms/step - loss: 0.1298 - accuracy: 0.9552 - val_loss: 1.8971 - val
_accuracy: 0.6590
Epoch 37/40
391/391 [==============================] - 2s 4ms/step - loss: 0.1266 - accuracy: 0.9570 - val_loss: 1.9502 - val
_accuracy: 0.6645
Epoch 38/40
391/391 [==============================] - 2s 4ms/step - loss: 0.0893 - accuracy: 0.9706 - val_loss: 2.0956 - val
_accuracy: 0.6668
Epoch 39/40
391/391 [==============================] - 2s 4ms/step - loss: 0.0818 - accuracy: 0.9718 - val_loss: 2.0608 - val
_accuracy: 0.6420
Epoch 40/40
391/391 [==============================] - 2s 4ms/step - loss: 0.1138 - accuracy: 0.9626 - val_loss: 2.2081 - val
_accuracy: 0.6296
The total running time for model with 0.1 lr on 40 epochs is: 1
```

In [72]:

```python
### PLOT LEARNING CURVES FOR MODEL 1  ###

fig = plt.figure(figsize=(20, 7))

sns.set()

fig.add_subplot(121)
plt.plot(history.history['loss'], label='Train')
plt.plot(history.history['val_loss'], label='Validation')
plt.xlabel("Epoch")
plt.ylabel("Cross entropy loss")
plt.legend()
plt.title("Loss vs epoch")

fig.add_subplot(122)
plt.plot(history.history['accuracy'], label='Train')
plt.plot(history.history['val_accuracy'], label='Validation')
plt.xlabel("Epoch")
plt.ylabel("Categorical accuracy")
plt.legend()
plt.title("Accuracy vs epoch")

plt.show()
```



In [73]:
```
print("The best loss for learning rate 0.1 on 40 epochs is:", min(history.history['val_loss']))
print("The best validation accuracy for learning rate 0.1 on 40 epochs is:", max(history.history['val_accuracy']))
```

```
The best loss for learning rate 0.1 on 40 epochs is: 0.9523820281028748
The best validation accuracy for learning rate 0.1 on 40 epochs is: 0.6880000233650208
```

Now, let's discuss the convergence of the model and any possible signatures of underfitting or overfitting.

Convergence of the model

Our validation loss seems to be increasing after epoch 15, which is a sign of overfitting and indeed we can see that the corresponding validation accuracy doesn't increase after epoch 15. It stagnates around 69% of accuracy. However, this is already so much better than the MLP learning values. In terms of noisiness, as for both methods we have used SGD optimization, we can see that the curves are quite noisy but that's just from the randomness aspect of SGD.

1.2.2 Implement L2 regularisation

Now we incorporate an L2 regularisation with a coefficient of 5.10^{-3} in each convolutional layer.

In [74]:
```
### MODEL 2 WITH LR=0.1 ON 40 EPOCHS AND L2 REGULARISATION IN EACH CONVOLUTIONAL LAYER ###

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.regularizers import l2
from tensorflow import keras

def get_model2():
  model = Sequential([
                  Conv2D(32, (3,3), activation='relu', kernel_regularizer=l2(5e-03), input_shape=(32,32,3)),
                  MaxPooling2D((2,2)),
                  Conv2D(64, (3,3), activation='relu', kernel_regularizer=l2(5e-03)),
                  MaxPooling2D((2,2)),
                  Conv2D(64, (3,3), activation='relu', kernel_regularizer=l2(5e-03)),
                  Flatten(),
                  Dense(64, activation='relu'),
                  Dense(10, activation='softmax')
                  ])
  opt = keras.optimizers.SGD(learning_rate=0.1)
  model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model
```

In [75]:
```
### GET MODEL 1 SUMMARY ###
model2 = get_model2()
model2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 30, 30, 32)        896
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 4, 4, 64)          36928
_________________________________________________________________
flatten_1 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)              (None, 64)                65600
_________________________________________________________________
dense_3 (Dense)              (None, 10)                650
=================================================================
Total params: 122,570
Trainable params: 122,570
```

```
Non-trainable params: 0
_________________________________________________________________
```

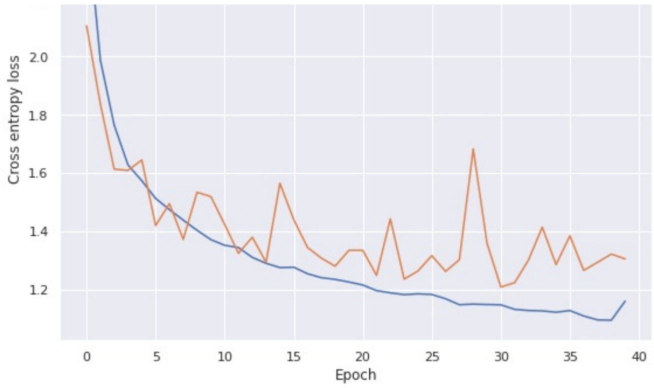
In [76]:
```python
### FIT AND TIME MODEL ###
import time
start = time.time()

history2 = model2.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=40, batch_size=128) # turn on GPU

end = time.time()

time22 = int((end - start)/60)
print("The ellapsed time for 40 epochs with lr 0.1 and L2 regularisation is:", time22)
```

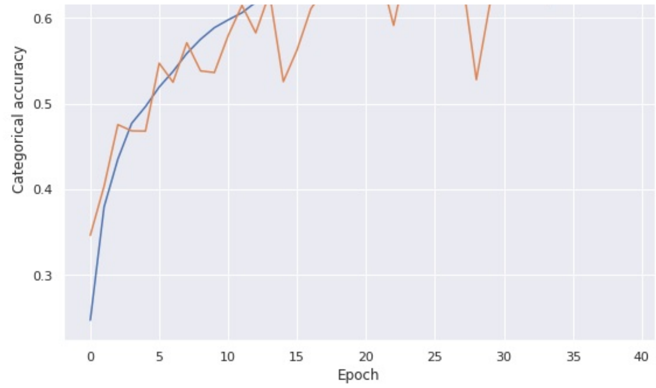
```
Epoch 1/40
391/391 [==============================] - 2s 5ms/step - loss: 2.6640 - accuracy: 0.1904 - val_loss: 2.1061 - val
_accuracy: 0.3463
Epoch 2/40
391/391 [==============================] - 2s 5ms/step - loss: 2.0695 - accuracy: 0.3600 - val_loss: 1.8373 - val
_accuracy: 0.4035
Epoch 3/40
391/391 [==============================] - 2s 4ms/step - loss: 1.8086 - accuracy: 0.4231 - val_loss: 1.6133 - val
_accuracy: 0.4755
Epoch 4/40
391/391 [==============================] - 2s 4ms/step - loss: 1.6461 - accuracy: 0.4697 - val_loss: 1.6089 - val
_accuracy: 0.4682
Epoch 5/40
391/391 [==============================] - 2s 4ms/step - loss: 1.5898 - accuracy: 0.4914 - val_loss: 1.6447 - val
_accuracy: 0.4679
Epoch 6/40
391/391 [==============================] - 2s 4ms/step - loss: 1.5272 - accuracy: 0.5138 - val_loss: 1.4194 - val
_accuracy: 0.5472
Epoch 7/40
391/391 [==============================] - 2s 4ms/step - loss: 1.4761 - accuracy: 0.5335 - val_loss: 1.4944 - val
_accuracy: 0.5250
Epoch 8/40
391/391 [==============================] - 2s 4ms/step - loss: 1.4482 - accuracy: 0.5545 - val_loss: 1.3717 - val
_accuracy: 0.5710
Epoch 9/40
391/391 [==============================] - 2s 4ms/step - loss: 1.3962 - accuracy: 0.5748 - val_loss: 1.5337 - val
_accuracy: 0.5382
Epoch 10/40
391/391 [==============================] - 2s 4ms/step - loss: 1.3784 - accuracy: 0.5857 - val_loss: 1.5188 - val
_accuracy: 0.5363
Epoch 11/40
391/391 [==============================] - 2s 4ms/step - loss: 1.3615 - accuracy: 0.5923 - val_loss: 1.4232 - val
_accuracy: 0.5794
Epoch 12/40
391/391 [==============================] - 2s 4ms/step - loss: 1.3305 - accuracy: 0.6095 - val_loss: 1.3238 - val
_accuracy: 0.6147
Epoch 13/40
391/391 [==============================] - 2s 4ms/step - loss: 1.2973 - accuracy: 0.6262 - val_loss: 1.3791 - val
_accuracy: 0.5826
Epoch 14/40
391/391 [==============================] - 2s 4ms/step - loss: 1.2874 - accuracy: 0.6290 - val_loss: 1.2930 - val
_accuracy: 0.6290
Epoch 15/40
391/391 [==============================] - 2s 4ms/step - loss: 1.2873 - accuracy: 0.6324 - val_loss: 1.5647 - val
_accuracy: 0.5257
Epoch 16/40
391/391 [==============================] - 2s 4ms/step - loss: 1.2838 - accuracy: 0.6357 - val_loss: 1.4399 - val
_accuracy: 0.5629
Epoch 17/40
391/391 [==============================] - 2s 4ms/step - loss: 1.2511 - accuracy: 0.6468 - val_loss: 1.3439 - val
_accuracy: 0.6108
Epoch 18/40
391/391 [==============================] - 2s 4ms/step - loss: 1.2295 - accuracy: 0.6559 - val_loss: 1.3074 - val
_accuracy: 0.6339
Epoch 19/40
391/391 [==============================] - 2s 4ms/step - loss: 1.2324 - accuracy: 0.6586 - val_loss: 1.2796 - val
_accuracy: 0.6452
Epoch 20/40
391/391 [==============================] - 2s 4ms/step - loss: 1.2126 - accuracy: 0.6678 - val_loss: 1.3342 - val
_accuracy: 0.6265
Epoch 21/40
391/391 [==============================] - 2s 4ms/step - loss: 1.2068 - accuracy: 0.6690 - val_loss: 1.3341 - val
_accuracy: 0.6252
Epoch 22/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1787 - accuracy: 0.6763 - val_loss: 1.2480 - val
_accuracy: 0.6543
Epoch 23/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1876 - accuracy: 0.6779 - val_loss: 1.4424 - val
```

```
_accuracy: 0.5913
Epoch 24/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1854 - accuracy: 0.6803 - val_loss: 1.2349 - val
_accuracy: 0.6669
Epoch 25/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1608 - accuracy: 0.6910 - val_loss: 1.2631 - val
_accuracy: 0.6608
Epoch 26/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1760 - accuracy: 0.6895 - val_loss: 1.3159 - val
_accuracy: 0.6369
Epoch 27/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1531 - accuracy: 0.6960 - val_loss: 1.2617 - val
_accuracy: 0.6589
Epoch 28/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1319 - accuracy: 0.7001 - val_loss: 1.3020 - val
_accuracy: 0.6442
Epoch 29/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1524 - accuracy: 0.6968 - val_loss: 1.6829 - val
_accuracy: 0.5279
Epoch 30/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1538 - accuracy: 0.6952 - val_loss: 1.3585 - val
_accuracy: 0.6179
Epoch 31/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1569 - accuracy: 0.6939 - val_loss: 1.2079 - val
_accuracy: 0.6812
Epoch 32/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1235 - accuracy: 0.7070 - val_loss: 1.2228 - val
_accuracy: 0.6689
Epoch 33/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1247 - accuracy: 0.7071 - val_loss: 1.3006 - val
_accuracy: 0.6513
Epoch 34/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1144 - accuracy: 0.7130 - val_loss: 1.4137 - val
_accuracy: 0.6209
Epoch 35/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1177 - accuracy: 0.7108 - val_loss: 1.2850 - val
_accuracy: 0.6543
Epoch 36/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1390 - accuracy: 0.7007 - val_loss: 1.3840 - val
_accuracy: 0.6267
Epoch 37/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1004 - accuracy: 0.7172 - val_loss: 1.2651 - val
_accuracy: 0.6657
Epoch 38/40
391/391 [==============================] - 2s 4ms/step - loss: 1.0775 - accuracy: 0.7265 - val_loss: 1.2928 - val
_accuracy: 0.6555
Epoch 39/40
391/391 [==============================] - 2s 4ms/step - loss: 1.0894 - accuracy: 0.7214 - val_loss: 1.3212 - val
_accuracy: 0.6497
Epoch 40/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1204 - accuracy: 0.7147 - val_loss: 1.3048 - val
_accuracy: 0.6464
The ellapsed time for 40 epochs with lr 0.1 and L2 regularisation is: 1
```

In [77]:

```python
### PLOT LEARNING CURVES FOR MODEL 2  ###

fig = plt.figure(figsize=(20, 7))

sns.set()

fig.add_subplot(121)
plt.plot(history2.history['loss'], label='Train')
plt.plot(history2.history['val_loss'], label='Validation')
plt.xlabel("Epoch")
plt.ylabel("Cross entropy loss")
plt.legend()
plt.title("Loss vs epoch")

fig.add_subplot(122)
plt.plot(history2.history['accuracy'], label='Train')
plt.plot(history2.history['val_accuracy'], label='Validation')
plt.xlabel("Epoch")
plt.ylabel("Categorical accuracy")
plt.legend()
plt.title("Accuracy vs epoch")

plt.show()
```





In [78]:
```
print("The best loss for learning rate 0.1 on 40 epochs with L2 regularisation is:", min(history2.history['val_lo
print("The best validation accuracy for learning rate 0.1 on 40 epochs with L2 regularisation is:", max(history2.
```

```
The best loss for learning rate 0.1 on 40 epochs with L2 regularisation is: 1.2079341411590576
The best validation accuracy for learning rate 0.1 on 40 epochs with L2 regularisation is: 0.6812000274658203
```

Now, let's compare the loss and accuracy of this model 2 compared to model 1:

Using the L2 regularisation allowed us to counter the effect of overfitting, indeed although our validation loss curve is a bit wobbly, its overall trend is going down. This is because L2 regularisation applies penalties on layer parameters which are summed into the loss function that the network optimizes. The validation loss is a bit higher than model 1 (1.23 vs 0.96) but the tradeoff is that we avoid overfitting so it's a good deal. Similarly, our validation accuracy reaches 68% which is lower than model 1 (not significantly though) but without any overfitting.

Now we can explain how the regularisation affects the training procedure: [5]

Regularisation is a method that is widely used in order to avoid overfitting and increase our model interpretability. Essentially, it reduces the variance of the model, without substantial increase in its bias. It encourages sparsity in the weights; it discourages the weights from growing too large, which restricts the capacity of the network. In our case, simply we manage to not have overfitting while achieving a good validation acccuracy.

Mathematically L2 regularisation is defined as: $L(\textbf{w},\alpha)= L_0(\textbf{w}) + \alpha_2 \sum_i w_i^2$, where L_0 is the unconstrained loss (here cross entropy) and α_2 is the regularisation parameter, which is actually a hyperparameter that we have set from the exercise to be $5*10^{-3}$.

1.2.3 Implement Dropout and another regularization: Stopping criterion

In this section, we will try implementing a Dropout with rate 0.5 on the same initial model and then a stopping criteron on the same initial model. This will allows us to compare them all afterwards.

DROPOUT

In [79]:
```
### MODEL 3 WITH LR=0.1 ON 40 EPOCHS, A DROPOUT RATE OF 0.5 BETWEEN CONVOLUTIONAL LAYER ###

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.regularizers import l2
from tensorflow import keras

def get_model3():
  rate = 0.5
  model = Sequential([
                    Conv2D(32, (3,3), activation='relu', input_shape=(32,32,3)),
                    MaxPooling2D((2,2)),
                    Dropout(rate),
                    Conv2D(64, (3,3), activation='relu'),
                    MaxPooling2D((2,2)),
                    Dropout(rate),
                    Conv2D(64, (3,3), activation='relu'),
                    Flatten(),
                    Dense(64, activation='relu'),
                    Dense(10, activation='softmax')
                  ])

  opt = keras.optimizers.SGD(learning_rate=0.1)
  model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
  return model
```

In [80]:
```
### GET MODEL 3 SUMMARY ###
```

```
model3 = get_model3()
model3.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_6 (Conv2D)            (None, 30, 30, 32)        896       
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 15, 15, 32)        0         
_________________________________________________________________
dropout (Dropout)            (None, 15, 15, 32)        0         
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 13, 13, 64)        18496     
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 6, 6, 64)          0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 6, 6, 64)          0         
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 4, 4, 64)          36928     
_________________________________________________________________
flatten_2 (Flatten)          (None, 1024)              0         
_________________________________________________________________
dense_4 (Dense)              (None, 64)                65600     
_________________________________________________________________
dense_5 (Dense)              (None, 10)                650       
=================================================================
Total params: 122,570
Trainable params: 122,570
Non-trainable params: 0
_________________________________________________________________
```

In [81]:

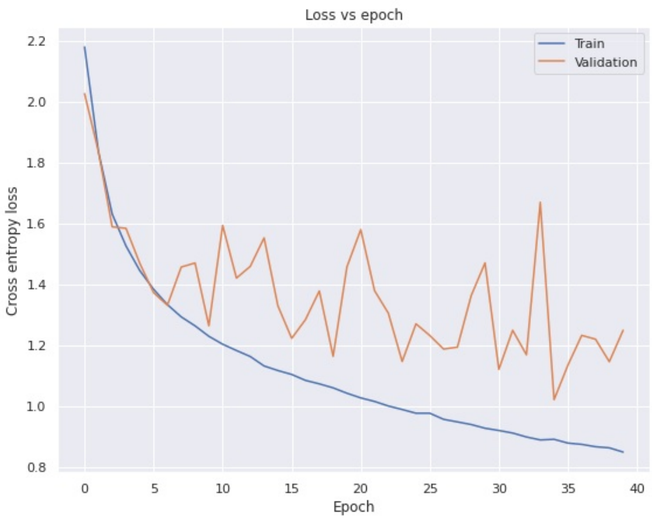
```
### FIT MODEL 3 ###
import time
start = time.time()

history3 = model3.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=40, batch_size=128) # turn on GPU

end = time.time()

time23 = int((end - start)/60)
print("The ellapsed time for 40 epochs with lr 0.1 and Dropout:", time23)
```

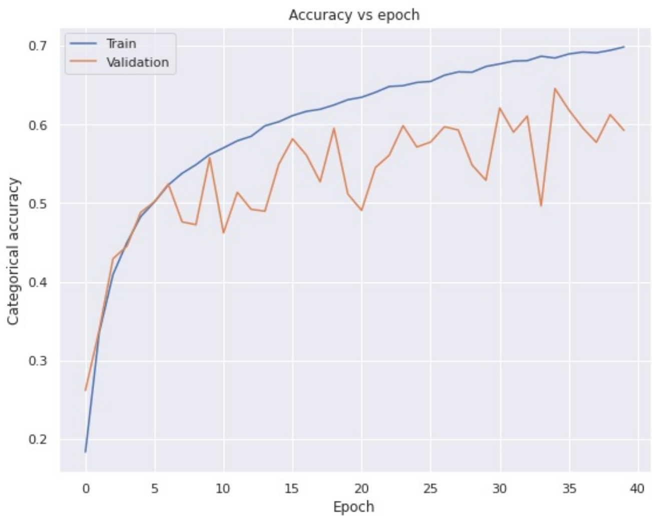
```
Epoch 1/40
391/391 [==============================] - 2s 5ms/step - loss: 2.2568 - accuracy: 0.1379 - val_loss: 2.0268 - val
_accuracy: 0.2619
Epoch 2/40
391/391 [==============================] - 2s 4ms/step - loss: 1.9177 - accuracy: 0.3075 - val_loss: 1.8375 - val
_accuracy: 0.3385
Epoch 3/40
391/391 [==============================] - 2s 4ms/step - loss: 1.6685 - accuracy: 0.3962 - val_loss: 1.5892 - val
_accuracy: 0.4292
Epoch 4/40
391/391 [==============================] - 2s 4ms/step - loss: 1.5470 - accuracy: 0.4419 - val_loss: 1.5843 - val
_accuracy: 0.4451
Epoch 5/40
391/391 [==============================] - 2s 4ms/step - loss: 1.4552 - accuracy: 0.4779 - val_loss: 1.4691 - val
_accuracy: 0.4881
Epoch 6/40
391/391 [==============================] - 2s 4ms/step - loss: 1.3970 - accuracy: 0.4988 - val_loss: 1.3727 - val
_accuracy: 0.5016
Epoch 7/40
391/391 [==============================] - 2s 4ms/step - loss: 1.3434 - accuracy: 0.5190 - val_loss: 1.3315 - val
_accuracy: 0.5241
Epoch 8/40
391/391 [==============================] - 2s 4ms/step - loss: 1.3028 - accuracy: 0.5321 - val_loss: 1.4569 - val
_accuracy: 0.4760
Epoch 9/40
391/391 [==============================] - 2s 4ms/step - loss: 1.2745 - accuracy: 0.5450 - val_loss: 1.4704 - val
_accuracy: 0.4725
Epoch 10/40
391/391 [==============================] - 2s 4ms/step - loss: 1.2276 - accuracy: 0.5627 - val_loss: 1.2641 - val
_accuracy: 0.5574
Epoch 11/40
391/391 [==============================] - 2s 4ms/step - loss: 1.2212 - accuracy: 0.5637 - val_loss: 1.5936 - val
_accuracy: 0.4621
Epoch 12/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1780 - accuracy: 0.5793 - val_loss: 1.4207 - val
_accuracy: 0.5136
Epoch 13/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1674 - accuracy: 0.5843 - val_loss: 1.4593 - val
_accuracy: 0.4920
```

```
Epoch 14/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1253 - accuracy: 0.5988 - val_loss: 1.5530 - val
_accuracy: 0.4896
Epoch 15/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1180 - accuracy: 0.6004 - val_loss: 1.3296 - val
_accuracy: 0.5492
Epoch 16/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1061 - accuracy: 0.6101 - val_loss: 1.2227 - val
_accuracy: 0.5815
Epoch 17/40
391/391 [==============================] - 2s 4ms/step - loss: 1.0844 - accuracy: 0.6181 - val_loss: 1.2846 - val
_accuracy: 0.5609
Epoch 18/40
391/391 [==============================] - 2s 4ms/step - loss: 1.0735 - accuracy: 0.6195 - val_loss: 1.3785 - val
_accuracy: 0.5270
Epoch 19/40
391/391 [==============================] - 2s 4ms/step - loss: 1.0525 - accuracy: 0.6263 - val_loss: 1.1637 - val
_accuracy: 0.5949
Epoch 20/40
391/391 [==============================] - 2s 4ms/step - loss: 1.0455 - accuracy: 0.6316 - val_loss: 1.4576 - val
_accuracy: 0.5115
Epoch 21/40
391/391 [==============================] - 2s 4ms/step - loss: 1.0214 - accuracy: 0.6392 - val_loss: 1.5801 - val
_accuracy: 0.4906
Epoch 22/40
391/391 [==============================] - 2s 4ms/step - loss: 1.0123 - accuracy: 0.6436 - val_loss: 1.3795 - val
_accuracy: 0.5451
Epoch 23/40
391/391 [==============================] - 2s 4ms/step - loss: 0.9960 - accuracy: 0.6505 - val_loss: 1.3053 - val
_accuracy: 0.5607
Epoch 24/40
391/391 [==============================] - 2s 4ms/step - loss: 0.9789 - accuracy: 0.6523 - val_loss: 1.1467 - val
_accuracy: 0.5983
Epoch 25/40
391/391 [==============================] - 2s 4ms/step - loss: 0.9783 - accuracy: 0.6515 - val_loss: 1.2705 - val
_accuracy: 0.5712
Epoch 26/40
391/391 [==============================] - 2s 4ms/step - loss: 0.9746 - accuracy: 0.6549 - val_loss: 1.2321 - val
_accuracy: 0.5775
Epoch 27/40
391/391 [==============================] - 2s 4ms/step - loss: 0.9580 - accuracy: 0.6614 - val_loss: 1.1875 - val
_accuracy: 0.5969
Epoch 28/40
391/391 [==============================] - 2s 4ms/step - loss: 0.9448 - accuracy: 0.6656 - val_loss: 1.1939 - val
_accuracy: 0.5928
Epoch 29/40
391/391 [==============================] - 2s 4ms/step - loss: 0.9424 - accuracy: 0.6664 - val_loss: 1.3634 - val
_accuracy: 0.5483
Epoch 30/40
391/391 [==============================] - 2s 4ms/step - loss: 0.9256 - accuracy: 0.6751 - val_loss: 1.4707 - val
_accuracy: 0.5289
Epoch 31/40
391/391 [==============================] - 2s 4ms/step - loss: 0.9169 - accuracy: 0.6801 - val_loss: 1.1208 - val
_accuracy: 0.6207
Epoch 32/40
391/391 [==============================] - 2s 4ms/step - loss: 0.9068 - accuracy: 0.6808 - val_loss: 1.2493 - val
_accuracy: 0.5900
Epoch 33/40
391/391 [==============================] - 2s 4ms/step - loss: 0.9013 - accuracy: 0.6793 - val_loss: 1.1687 - val
_accuracy: 0.6104
Epoch 34/40
391/391 [==============================] - 2s 4ms/step - loss: 0.8893 - accuracy: 0.6888 - val_loss: 1.6699 - val
_accuracy: 0.4963
Epoch 35/40
391/391 [==============================] - 2s 4ms/step - loss: 0.8886 - accuracy: 0.6850 - val_loss: 1.0212 - val
_accuracy: 0.6454
Epoch 36/40
391/391 [==============================] - 2s 4ms/step - loss: 0.8788 - accuracy: 0.6879 - val_loss: 1.1346 - val
_accuracy: 0.6185
Epoch 37/40
391/391 [==============================] - 2s 4ms/step - loss: 0.8717 - accuracy: 0.6932 - val_loss: 1.2325 - val
_accuracy: 0.5957
Epoch 38/40
391/391 [==============================] - 2s 4ms/step - loss: 0.8547 - accuracy: 0.6959 - val_loss: 1.2200 - val
_accuracy: 0.5771
Epoch 39/40
391/391 [==============================] - 2s 4ms/step - loss: 0.8644 - accuracy: 0.6941 - val_loss: 1.1460 - val
_accuracy: 0.6123
Epoch 40/40
391/391 [==============================] - 2s 4ms/step - loss: 0.8357 - accuracy: 0.7056 - val_loss: 1.2494 - val
_accuracy: 0.5923
The ellapsed time for 40 epochs with lr 0.1 and Dropout: 1
```

In [82]:

```python
### PLOT LEARNING CURVES FOR MODEL 3  ###

fig = plt.figure(figsize=(20, 7))

sns.set()

fig.add_subplot(121)
plt.plot(history3.history['loss'], label='Train')
plt.plot(history3.history['val_loss'], label='Validation')
plt.xlabel("Epoch")
plt.ylabel("Cross entropy loss")
plt.legend()
plt.title("Loss vs epoch")

fig.add_subplot(122)
plt.plot(history3.history['accuracy'], label='Train')
plt.plot(history3.history['val_accuracy'], label='Validation')
plt.xlabel("Epoch")
plt.ylabel("Categorical accuracy")
plt.legend()
plt.title("Accuracy vs epoch")

plt.show()
```





In [83]:

```python
print("The best loss for learning rate 0.1 on 40 epochs with dropout is:", min(history3.history['val_loss']))
print("The best validation accuracy for learning rate 0.1 on 40 epochs with dropout is:", max(history3.history['\
```

```
The best loss for learning rate 0.1 on 40 epochs with dropout is: 1.0212005376815796
The best validation accuracy for learning rate 0.1 on 40 epochs with dropout is: 0.6453999876976013
```

Comment

The Dropout technique essentially drops out randomly neurons in the network. This has a similar effect as the regularisation. It also prevents neurons from co-adapting too much to each other. We can see that results in terms of loss and accuracy are similar to the one with L2 reg, although a bit smaller in accuracy but better loss, this model doesn't present any sign of overfitting. Here our hyperparameter is the dropout rate, which is set by the exercise to 0.5.

EARLY STOPPPING

In [84]:

```python
### MODEL 4 WITH LR=0.1 ON 40 EPOCHS AND AN EARLY STOPPING REGULARISATION###

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping

def get_model4():
  model = Sequential([
                      Conv2D(32, (3,3), activation='relu', input_shape=(32,32,3)),
                      MaxPooling2D((2,2)),
                      Conv2D(64, (3,3), activation='relu'),
                      MaxPooling2D((2,2)),
                      Conv2D(64, (3,3), activation='relu'),
                      Flatten(),
                      Dense(64, activation='relu'),
                      Dense(10, activation='softmax')
                    ])

  opt = keras.optimizers.SGD(learning_rate=0.1)
```

```
    model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
    return model
```

In [85]:
```
### GET MODEL 4 SUMMARY ###
model4 = get_model4()
model4.summary()
```

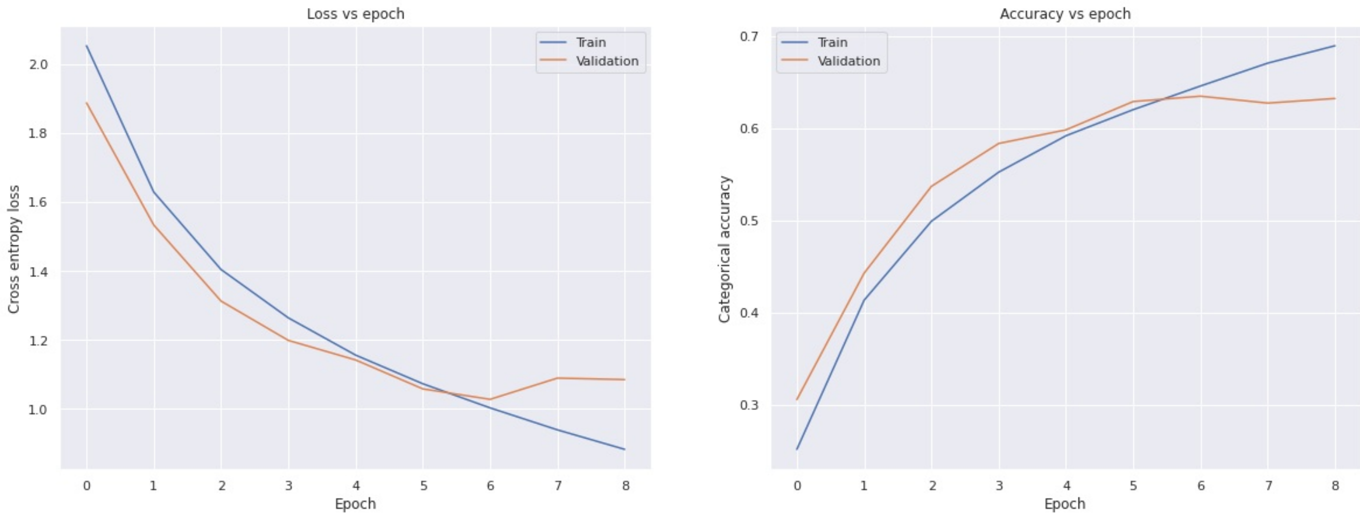
```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_9 (Conv2D)            (None, 30, 30, 32)        896
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 13, 13, 64)        18496
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 4, 4, 64)          36928
_________________________________________________________________
flatten_3 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_6 (Dense)              (None, 64)                65600
_________________________________________________________________
dense_7 (Dense)              (None, 10)                650
=================================================================
Total params: 122,570
Trainable params: 122,570
Non-trainable params: 0
_________________________________________________________________
```

In [86]:
```
### FIT MODEL 4 ###
import time
start = time.time()

early_stopping = EarlyStopping(patience=2, monitor='val_accuracy')
history4 = model4.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=40, batch_size=128, callbacks=[ear

end = time.time()

time24 = int((end - start)/60)
print("The ellapsed time for 40 epochs with lr 0.1 and stopping criterion is:", time24)
```

```
Epoch 1/40
391/391 [==============================] - 2s 5ms/step - loss: 2.1808 - accuracy: 0.1882 - val_loss: 1.8874 - val
_accuracy: 0.3056
Epoch 2/40
391/391 [==============================] - 2s 4ms/step - loss: 1.7145 - accuracy: 0.3819 - val_loss: 1.5332 - val
_accuracy: 0.4427
Epoch 3/40
391/391 [==============================] - 2s 4ms/step - loss: 1.4489 - accuracy: 0.4820 - val_loss: 1.3128 - val
_accuracy: 0.5368
Epoch 4/40
391/391 [==============================] - 2s 4ms/step - loss: 1.2789 - accuracy: 0.5454 - val_loss: 1.1988 - val
_accuracy: 0.5833
Epoch 5/40
391/391 [==============================] - 2s 4ms/step - loss: 1.1691 - accuracy: 0.5840 - val_loss: 1.1421 - val
_accuracy: 0.5981
Epoch 6/40
391/391 [==============================] - 2s 4ms/step - loss: 1.0849 - accuracy: 0.6157 - val_loss: 1.0582 - val
_accuracy: 0.6288
Epoch 7/40
391/391 [==============================] - 2s 4ms/step - loss: 1.0120 - accuracy: 0.6430 - val_loss: 1.0279 - val
_accuracy: 0.6346
Epoch 8/40
391/391 [==============================] - 2s 4ms/step - loss: 0.9420 - accuracy: 0.6718 - val_loss: 1.0895 - val
_accuracy: 0.6271
Epoch 9/40
391/391 [==============================] - 2s 4ms/step - loss: 0.8819 - accuracy: 0.6902 - val_loss: 1.0851 - val
_accuracy: 0.6321
The ellapsed time for 40 epochs with lr 0.1 and stopping criterion is: 0
```

In [87]:
```
### PLOT LEARNING CURVES FOR MODEL 4  ###

fig = plt.figure(figsize=(20, 7))

sns.set()

fig.add_subplot(121)
```

```
plt.plot(history4.history['loss'], label='Train')
plt.plot(history4.history['val_loss'], label='Validation')
plt.xlabel("Epoch")
plt.ylabel("Cross entropy loss")
plt.legend()
plt.title("Loss vs epoch")

fig.add_subplot(122)
plt.plot(history4.history['accuracy'], label='Train')
plt.plot(history4.history['val_accuracy'], label='Validation')
plt.xlabel("Epoch")
plt.ylabel("Categorical accuracy")
plt.legend()
plt.title("Accuracy vs epoch")

plt.show()
```



In [88]:

```
print("The best loss for learning rate 0.1 on 40 epochs with early stopping criterion is:", min(history4.history[
print("The best validation accuracy for learning rate 0.1 on 40 epochs with early stopping criterion is:", max(hi
```

```
The best loss for learning rate 0.1 on 40 epochs with early stopping criterion is: 1.027906894683838
The best validation accuracy for learning rate 0.1 on 40 epochs with early stopping criterion is: 0.6345999836921
692
```

Comment

When we set the patience to 15 with the val_accuracy metric, we can see that our stopping criterion is too big and doesn't prevent overfitting. So we might want to try lower patience, or change the stopping criterion to val_loss for example. When we try patience=5, we get a val accuracy of 70% with very low overfitting. Now let's change our metric to val_loss with patience=15 to see what it does: overfitting and val accuracy 67.8%. So let's keep the previous idea, patience=5 with val_accuracy metric. Here our hyperparameter is the patience.

Now we can compare the results of our stopping criterion to the L2 regularisation, the Dropout and the standard model as follows:

In [89]:

```
### VALUES FOR CNN ###

Learning_Rate2 = [0.1, 0.1, 0.1, 0.1]
Number_of_Epochs2 = [40, 40, 40, 40]
Loss_tot_train2 = [min(history.history['loss']), min(history2.history['loss']), min(history3.history['loss']), mi
Loss_tot_test2 = [min(history.history['val_loss']), min(history2.history['val_loss']), min(history3.history['val_
Acc_tot_train2 = [max(history.history['accuracy']), max(history2.history['accuracy']), max(history3.history['accu
Acc_tot_test2 = [max(history.history['val_accuracy']), max(history2.history['val_accuracy']), max(history3.histor
Regularisation = ['none', 'L2=5*10^-3', 'dropout rate=0.5', 'early stopping:patience=15']
run_epoch2 = [time21, time22, time23, time24]


comparison2 = np.array([Learning_Rate2, Number_of_Epochs2, Regularisation, Loss_tot_train2, Loss_tot_test2, Acc_t
comparison2 = pd.DataFrame(comparison2)
comparison2.columns = ['Learning Rate','Number of Epochs', 'Regularisation','Training loss', 'Test loss', 'Traini
comparison2.head()
```

Out[89]:

	Learning Rate	Number of Epochs	Regularisation	Training loss	Test loss	Training accuracy	Validation accuracy	Timing ellapsed
0	0.1	40	none	0.09950757771730423	0.9523820281028748	0.9662399888038635	0.6880000233650208	1
1	0.1	40	L2=5*10^-3	1.0938355922698975	1.2079341411590576	0.7190399765968323	0.6812000274658203	1
2	0.1	40	dropout rate=0.5	0.8490926623344421	1.0212005376815796	0.698360025882721	0.6453999876976013	1
3	0.1	40	early stopping:patience=15	0.8829795122146606	1.027906894683838	0.689300000667572	0.6345999836921692	0

1.2.4 Compare results from MLP and CNN

In [90]:

```
### VALUES FOR MLP ###

Learning_Rate = [0.01, 0.0001, 0.1, 0.01]
Number_of_Epochs = [40, 40, 40, 80]
Loss_tot_train = [min(L_1), min(L_2), min(L_3), min(L_4)]
Loss_tot_test = [min(L_val_1), min(L_val_2), min(L_val_3), min(L_val_4)]
Acc_tot_train = [max(A_1), max(A_2), max(A_3), max(A_4)]
Acc_tot_test = [max(A_val_1), max(A_val_2), max(A_val_3), max(A_val_4)]
Time = [time1, time2, time3, time4]

comparison = np.array([Learning_Rate, Number_of_Epochs, Loss_tot_train, Loss_tot_test, Acc_tot_train, Acc_tot_tes
comparison = pd.DataFrame(comparison)
comparison.columns = ['Learning Rate','Number of Epochs', 'Training loss', 'Test loss', 'Training accuracy', 'Val
comparison.head()
```

Out[90]:

	Learning Rate	Number of Epochs	Training loss	Test loss	Training accuracy	Validation accuracy	Timing ellapsed
0	0.0100	40.0	1.155633	1.333278	59.870	53.08	22.0
1	0.0001	40.0	1.890943	1.892781	34.180	34.17	21.0
2	0.1000	40.0	0.444469	1.387891	84.724	52.60	23.0
3	0.0100	80.0	0.749191	1.296803	74.016	54.99	44.0

Compare MLP and CNN

Accuracy from the models

Overall, our CNN fits better the data, one could think it is because the used library tensorflow is optimized for this, but also it has less parameters. Also in general the SGD optimization gives us quite noisy curves in both cases, (which is re-assuring because that means we are correctly implementing the stochasticity aspect of the method), but we still manage to get satisfying results, especially with CNN. From the MLP, we get a best val accuracy around 53%, which is already 5 times better than randomn predictions, but with our CNN, we increase this number to almost 70% validation accuracy, which is much more satisfying for a model to implement in the real world. On top of that we have implemented measures to prevent overfitting while still getting high accuracies around 70%, compared to barely scratching 53% validation accuracy with the MLP.

Computational time over same number of epochs

Using our homemade MLP on 40 epochs, we get a training of a bit less than 1/2 hour, but on the same number of epochs with CNN, we get a training time of less than a minute (with GPU activated). This is mainly because tensorflow is optimised with GPU on colab, but also, our CNN has 15 times less parameters than our MLP (see below).

Number of parameters in the models

In the part 1.1, our model has around 1.8 million parameters vs in part 1.2 around 0.12 million parameters. Note that we have 50,000 samples.

A rule of thumb (Zhang et al. (2017) [9]) is that an i-layer network can fit a model with i*n + d parameters, with n number of samples and d the dimension. If a model has more parameters, it doesn't necessarily overfit but it's not ideal. In our case we have :

- MLP from 1.1 has a total of 6 layers With n=50,000 and d=3,072 the rule of thumb that a max number of parameters for a good fit should be 6*50,000+3,072 = 303,072. But our model has 1.8 million parameters.
- CNN from 1.2 has a total of 5 layers. With n=50000 and d=3072 we get from the rule of thumb that a max number of parameters for a good fit should be 5*50,000+3,072 = 253,072. Our model has 122,750 parameters.

Clearly, as described above, we have better accuracy results on our 1.2 so maybe it is do to the fact that our number of parameters is closer to that rule of thumb to the number of samples than in 1.1.

TASK 2: UNSUPERVISED LEARNING

In this task, we are working with a dataset which describes a karate club. We have 3 pieces of information:

1. A feature matrix F which characterizes the personality profile of all the individuals. We have N=34 samples (individuals) with each p=100 features capturing different traits. So our feature matrix F is N x p.
2. An adjacency matrix A which characterizes the social network of friendships between the members of the club. We have N=34 nodes

(members) and E=78 edges corresponding to the friendships between them. The adjacency matrix A is N x N. `

3. A ground truth acrimonious split of the data into two separate groups which we'll only use at the end to compare our results.

2.0 Exploratory Data Analysis

First of all, let's implement an exploratory data analysis of our matrices.

In [101…

```
### LOAD NECESSARY LIBRARIES ###
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import scipy as sc
```

FEATURE MATRIX

In [102…

```
### FEATURE MATRIX F ###
F = pd.read_csv("/content/feature_matrix_karate_club.csv")
F.drop(F.columns[[0]], axis=1, inplace=True) #delete first column
F.head()
```

Out[102…

	0	1	2	3	4	5	6	7	8	9	10	11	12	
0	0.148640	-0.187110	-0.109395	0.111767	-0.024913	-0.117825	-0.072237	0.109746	-0.088250	0.210684	0.150548	0.073093	0.123050	0.01
1	0.207001	-0.018979	-0.094483	0.013424	-0.106083	0.252485	0.092830	0.002718	0.224389	0.061539	-0.011964	-0.124062	0.363754	-0.18
2	0.093962	-0.262451	-0.033010	0.108551	0.008679	-0.077413	0.051345	0.082396	-0.020187	0.186536	0.089231	0.135319	0.090328	0.02
3	0.212280	-0.047187	-0.116486	-0.004627	0.025335	0.020590	-0.123587	0.059001	-0.034748	0.131113	0.042459	0.030327	0.337262	0.12
4	0.053653	0.122337	-0.135267	-0.017862	-0.012216	0.064982	-0.053867	0.120750	-0.060154	0.204295	-0.206539	-0.015263	0.217170	-0.14

5 rows × 100 columns

In [103…

```
### CHECK NUMBER OF UNIQUE VALUES PER FEATURE ###
F.nunique()
```

Out[103…

```
0      34
1      34
2      34
3      34
4      34
       ..
95     34
96     34
97     34
98     34
99     34
Length: 100, dtype: int64
```

In [104…

```
### LOOK AT DISTRIBUTION OF EACH FEATURE ###
F.describe()
```

Out[104…

	0	1	2	3	4	5	6	7	8	9	10	11	
count	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.
mean	0.092756	-0.045116	-0.078589	0.022380	-0.046291	0.101231	-0.006378	-0.007856	-0.013739	0.093518	0.081096	-0.053237	0.
std	0.075746	0.103148	0.063907	0.078538	0.074103	0.119355	0.115739	0.106600	0.115659	0.140047	0.121088	0.146182	0.
min	-0.113510	-0.262451	-0.161568	-0.165533	-0.230700	-0.117825	-0.206979	-0.279716	-0.210795	-0.302302	-0.210334	-0.386797	0.
25%	0.040423	-0.127374	-0.123831	-0.016255	-0.065949	0.017477	-0.072496	-0.070706	-0.087543	0.055788	0.002626	-0.135123	0.
50%	0.092534	-0.047921	-0.087450	0.023166	-0.048777	0.086824	-0.019321	0.022096	-0.037313	0.123904	0.105982	-0.007264	0.
75%	0.146899	0.029343	-0.045687	0.067570	-0.007922	0.188491	0.056004	0.070105	0.035076	0.193471	0.154811	0.053742	0.
max	0.212280	0.145507	0.107496	0.203272	0.155897	0.379257	0.231879	0.121299	0.292380	0.341475	0.275987	0.163082	0.

8 rows × 100 columns

Looking at the distribution of the features, we can think of normalizing our feature matrix to get zero mean as follows:

In [105…

```
def normalize(X): # SHOULD I NORMALIZE?
```

```
    mu = np.mean(X, axis=0)
    std = np.std(X, axis=0)
    std_filled = std.copy()
    std_filled[std==0] = 1.
    Xbar = ((X-mu)/std_filled)
    return Xbar
```

In [106…

```
F = normalize(F)
```

In [107…

```
### LOOK AT DISTRIBUTION OF EACH FEATURE ###
F.describe()
```

Out[107…

	0	1	2	3	4	5	6	7	8
count	3.400000e+01	3.400000e+01	3.400000e+01	3.400000e+01	3.400000e+01	3.400000e+01	3.400000e+01	3.400000e+01	3.400000e+01
mean	1.436759e-16	6.530724e-17	-5.061311e-17	1.959217e-17	7.836868e-17	-8.816477e-17	4.081702e-18	-4.571507e-17	1.387779e-17
std	1.015038e+00	1.015038e+00	1.015038e+00	1.015038e+00	1.015038e+00	1.015038e+00	1.015038e+00	1.015038e+00	1.015038e+00
min	-2.764060e+00	-2.138706e+00	-1.317962e+00	-2.428625e+00	-2.525984e+00	-1.862937e+00	-1.759277e+00	-2.588626e+00	-1.729387e+00
25%	-7.012928e-01	-8.094677e-01	-7.185871e-01	-4.993276e-01	-2.692725e-01	-7.122803e-01	-5.798594e-01	-5.984494e-01	-6.477116e-01
50%	-2.976697e-03	-2.760409e-02	-1.407435e-01	1.015424e-02	-3.404955e-02	-1.225251e-01	-1.135055e-01	2.852051e-01	-2.068871e-01
75%	7.255440e-01	7.327133e-01	5.225768e-01	5.840373e-01	5.255696e-01	7.420933e-01	5.470948e-01	7.423384e-01	4.284069e-01
max	1.601677e+00	1.875837e+00	2.955578e+00	2.337879e+00	2.769517e+00	2.364440e+00	2.089525e+00	1.229801e+00	2.686543e+00

8 rows × 100 columns

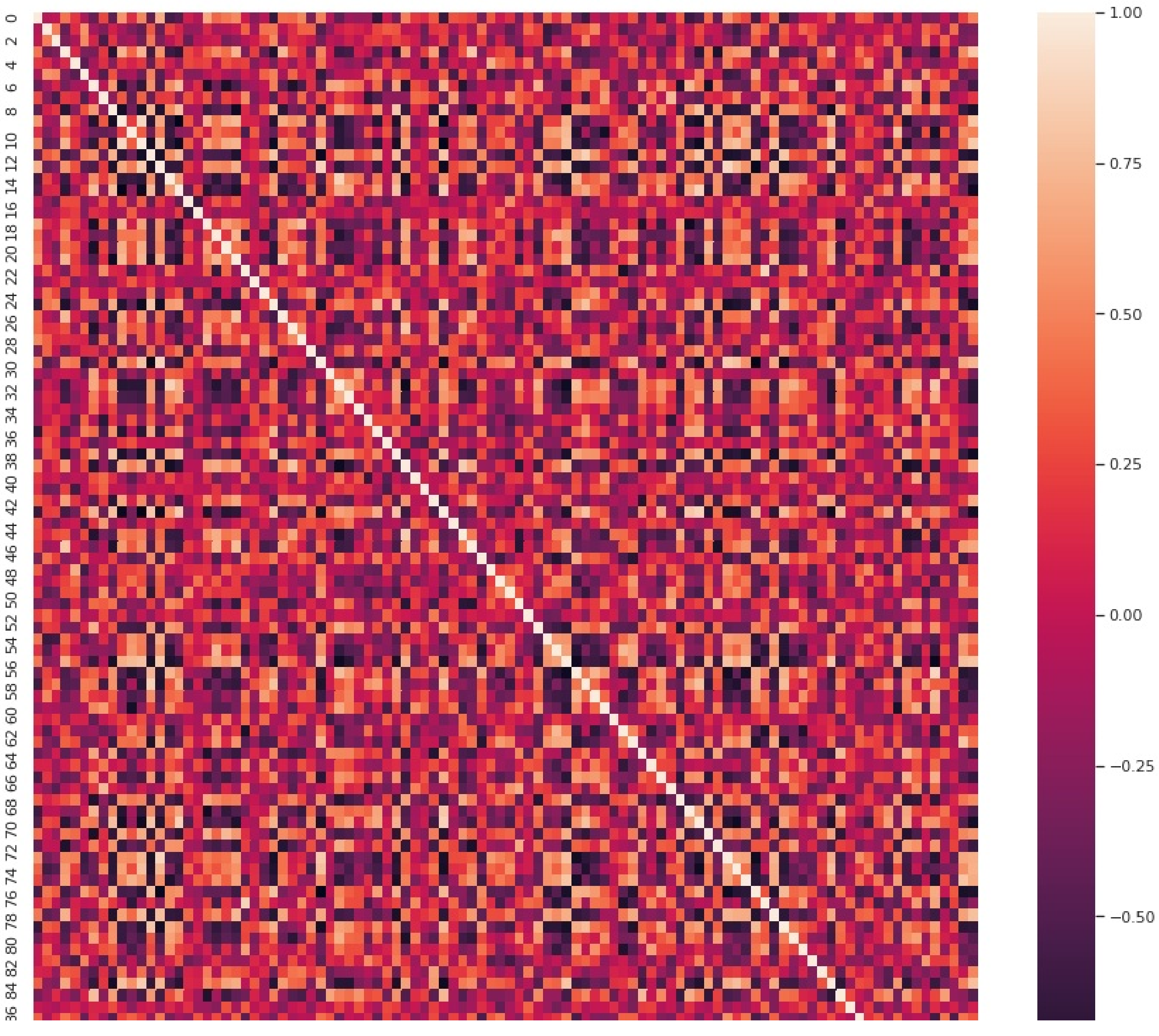
Now we can see that the data is well-centered around 0.

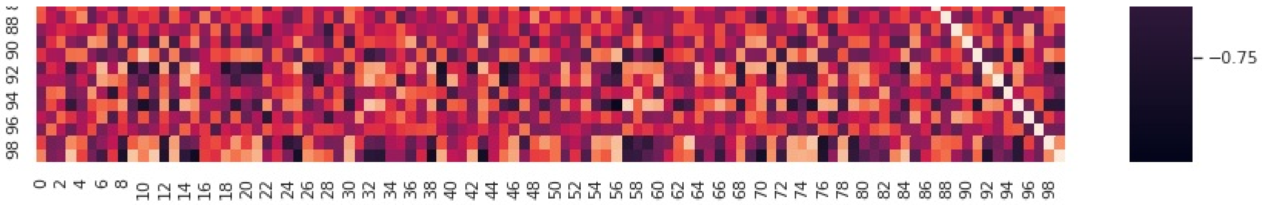
In [108…

```
### CORRELATION PLOT OF FEATURES ###
corr_matF = F.corr().round(2)
fix, ax = plt.subplots(figsize=(15,15))
sns.heatmap(data = corr_matF, annot=False)
```

Out[108…

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0e2d7a11d0>
```





ADJACENCY MATRIX

In [109…

```python
### ADJACENCY MATRIX A ###
A = pd.read_csv("/content/karate_club_graph.csv")
A.drop(A.columns[[0]], axis=1, inplace=True) #delete first column
A.head()
```

Out[109…

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	3
0	0.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
1	1.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1
2	1.0	1.0	0.0	1.0	0.0	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0
3	1.0	1.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
4	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0

In [110…

```python
### CHECK NUMBER OF UNIQUE VALUES PER FEATURE ###
A.nunique()
```

Out[110…

```
0     2
1     2
2     2
3     2
4     2
5     2
6     2
7     2
8     2
9     2
10    2
11    2
12    2
13    2
14    2
15    2
16    2
17    2
18    2
19    2
20    2
21    2
22    2
23    2
24    2
25    2
26    2
27    2
28    2
29    2
30    2
31    2
32    2
33    2
dtype: int64
```

In [111…

```python
A.describe()
```

Out[111…

	0	1	2	3	4	5	6	7	8	9	10	11	
count	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.000000	34.
mean	0.470588	0.264706	0.294118	0.176471	0.088235	0.117647	0.117647	0.117647	0.147059	0.058824	0.088235	0.029412	0.
std	0.506640	0.447811	0.462497	0.386953	0.287902	0.327035	0.327035	0.327035	0.359491	0.238833	0.287902	0.171499	0.
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.

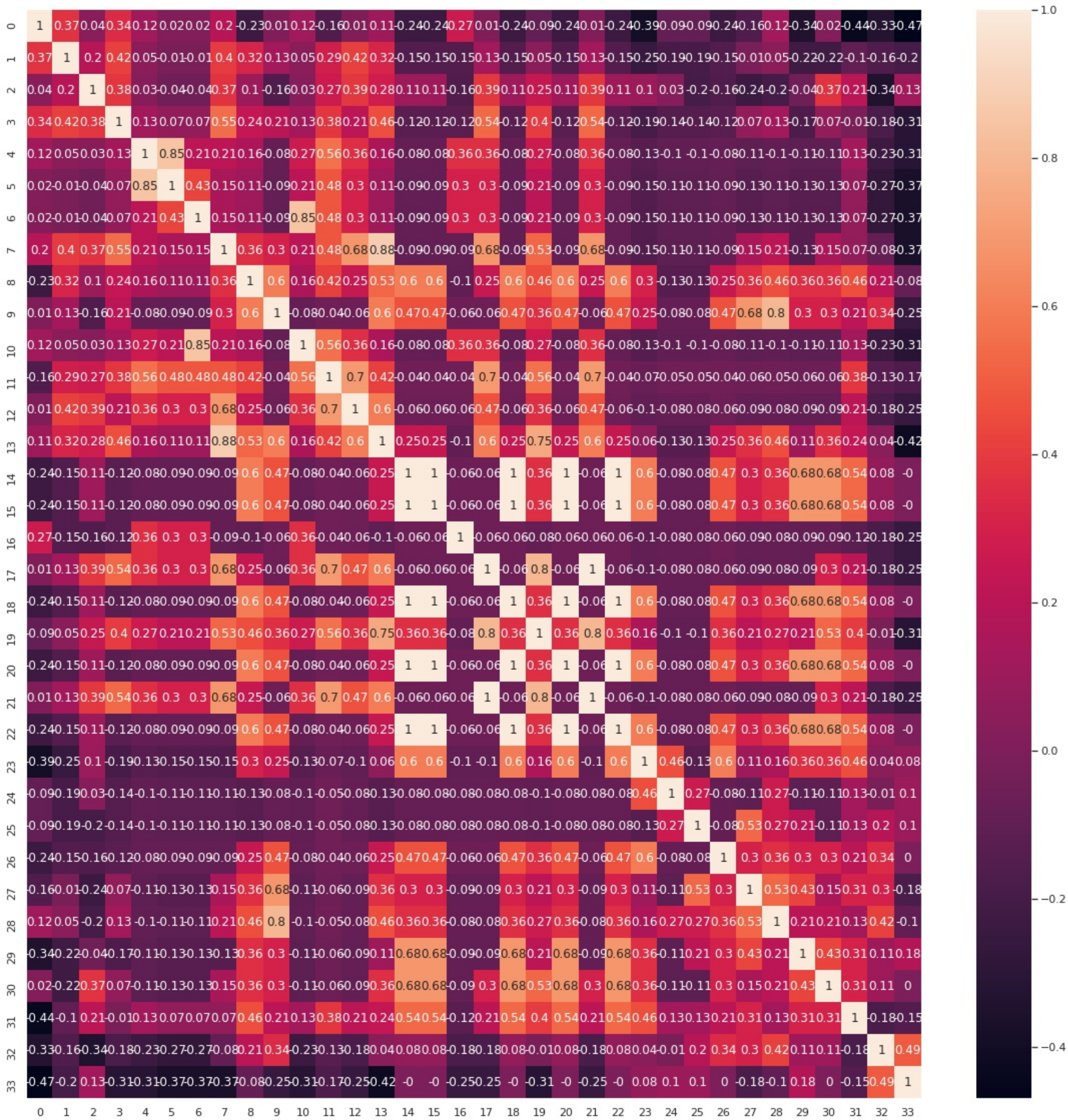
75%	1.000000	0.750000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.

This table is expected as it is binarily encoded so has mean zero. There is no need to standardize its entries.

In [112…

```
### CORRELATION MATRIX ###
corr_matA = A.corr().round(2)
fix, ax = plt.subplots(figsize=(20,20))
sns.heatmap(data = corr_matA, annot=True)
```

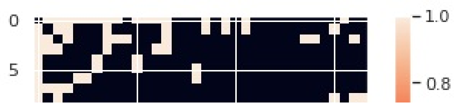
Out[112… <matplotlib.axes._subplots.AxesSubplot at 0x7f0e2f1f2050>

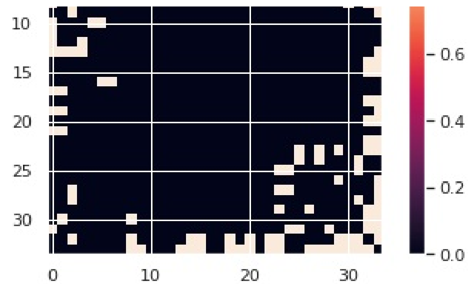


We can see that a few nodes are highly correlated.

In [113…

```
# plot a heatmap of the distance matrix # KEEP OR NOT?
plt.imshow(A)
plt.colorbar();
```





GROUND TRUTH

In [114…
```python
# Ground truth
GT = pd.read_csv("/content/ground_truth_karate_club.csv")
GT.drop(GT.columns[[0]], axis=1, inplace=True) #delete first column
GT.head()
```

Out[114…

	0
0	Mr. Hi
1	Mr. Hi
2	Mr. Hi
3	Mr. Hi
4	Mr. Hi

2.1 Clustering of the feature matrix

In this first task, we are implementing a k-means algorithm to compute a clustering of the feature matrix using NumPy only.

2.1.0 Load and normalize data

First of all, let's load and normalize the data.

In [115…
```python
### IMPORT NECESSARY PACKAGES ###
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import scipy as sc
```

In [116…
```python
### LOAD FEATURE MATRIX ###
F = pd.read_csv("/content/feature_matrix_karate_club.csv")
F.drop(F.columns[[0]], axis=1, inplace=True) #delete first column
F.head()
```

Out[116…

	0	1	2	3	4	5	6	7	8	9	10	11	12	
0	0.148640	-0.187110	-0.109395	0.111767	-0.024913	-0.117825	-0.072237	0.109746	-0.088250	0.210684	0.150548	0.073093	0.123050	0.0
1	0.207001	-0.018979	-0.094483	0.013424	-0.106083	0.252485	0.092830	0.002718	0.224389	0.061539	-0.011964	-0.124062	0.363754	-0.1
2	0.093962	-0.262451	-0.033010	0.108551	0.008679	-0.077413	0.051345	0.082396	-0.020187	0.186536	0.089231	0.135319	0.090328	0.0
3	0.212280	-0.047187	-0.116486	-0.004627	0.025335	0.020590	-0.123587	0.059001	-0.034748	0.131113	0.042459	0.030327	0.337262	0.1
4	0.053653	0.122337	-0.135267	-0.017862	-0.012216	0.064982	-0.053867	0.120750	-0.060154	0.204295	-0.206539	-0.015263	0.217170	-0.1

5 rows × 100 columns

In [117…
```python
F.shape #shape is coherent with exercise, N=34 samples and p=100 features capturing different traits
```

Out[117… (34, 100)

In [118…
```python
F = F.to_numpy() # transform to DataFrame to numpy array
```

In [119…
```python
def normalize(X): # normalize function
    mu = np.mean(X, axis=0)
    std = np.std(X, axis=0)
```

```
    std_filled = std.copy()
    std_filled[std==0] = 1.
    Xbar = ((X-mu)/std_filled)
    return Xbar
```

In [120…
```
F = normalize(F) # normalize data
```

2.1.1 Obtain optimised clusterings for k-means according to within distance

First of all, let's define a k-means algorithm and iterate 100 times over different values of k from 2 to 10 and average.

We then plot the average within-cluster distance as a function of k.

Let's define the within-cluster distance as follows:

$W_k = \sum_{i=1}^k \sum_{x\in C_q} (x-c_q)(x-c_q)^T$, where C_q is the set of points in cluster q and c_q is the center of the cluster q.

In [121…
```
### K MEANS ALGORITHM from CT ###

def k_means(k, X, max_iter):
    n_samples, n_features = X.shape

    labels = np.random.randint(low=0, high=k, size=n_samples)

    while (len(np.unique(labels)) < k): # make sure there are no empty clusters assigned in the initialization
        labels = np.random.randint(low=0, high=k, size=n_samples)

    X_labels = np.append(X, labels.reshape(-1,1), axis=1)
    centroids = np.zeros((k, n_features))

    for i in range(k):
        centroids[i] = np.mean([x for x in X_labels if x[-1]==i], axis=0)[0:n_features]

    #iterations
    new_labels = np.zeros(len(X))
    difference = 0

    for i in range(max_iter):
        # distances: between data points and centroids
        distances = np.array([np.linalg.norm(X - c, axis=1) for c in centroids])
        # new_labels: computed by finding centroid with minimal distance
        new_labels = np.argmin(distances, axis=0)

        if (labels==new_labels).all():
            # labels unchanged
            labels = new_labels
            break
        else:
            # labels changed
            # difference: percentage of changed labels
            difference = np.mean(labels!=new_labels)
            labels = new_labels
            for c in range(k):
                if len(X[labels==c])>0:
                # computing centroids by taking the mean over associated data points
                    centroids[c] = np.mean(X[labels==c], axis=0)
                else: # make sure that if any clusters are empty at the end, we assign a random number to it so t
                    centroids[c] = X[np.random.randint(low=0, high=len(X), size=1)]
    return centroids, labels
```

In [122…
```
### WITHIN DISTANCE FUNCTION ###

def within_distance2(X, k, labels, centroids):
    W = 0 # initalize distance for summation
    N = X.shape[0]
    for c in range(k): # double sum for matrix filling
        for i in range(N):
            if labels[i]==c:
                W += ( (X[i]-centroids[c]) @ (X[i]-centroids[c]).T)
    return W
```

Now that we have defined our 2 main functions, we can loop over 100 iterations, and different values of k to find the average within distance.

In [123…
```
### Loop over 100 random initialisation for k means between 2 and 10 ###

centroids, labels = [], []
max_iter = 15
W = np.zeros(9)

for i in range(100):
    for k in range(2,11):
        centroids, labels = k_means(k, F, max_iter)
        # compute within cluster distance
```

```
        W[k-2] += within_distance2(F, k, labels, centroids)/100
```

In [124...

```
### Plot of the average within-cluster distance as a function of k ###

import matplotlib.pyplot as plt
import seaborn as sns
k = np.arange(2,11,1)

sns.set()

plt.figure(figsize=(13,5))
plt.plot(k, W, '-o', color='g', label='within-distance')
plt.axhline(y=W[k==4], c='b', alpha=0.3)
plt.scatter(k[2], W[k==4], color='b', label='Elbow')
plt.title("Average within-cluster distance as a function of k")
plt.xlabel("k")
plt.ylabel("W(C)")
plt.legend()
plt.show()

k_optimal = k[2]
print('{} is the optimal number of clusters.'.format(k[2]))
```


```
4 is the optimal number of clusters.
```

It is not obvious that the elbow is at k=4. It does look like it, but we might wonder what about k=2?.

So let's try a different range for our k-means starting at 1 clusters:

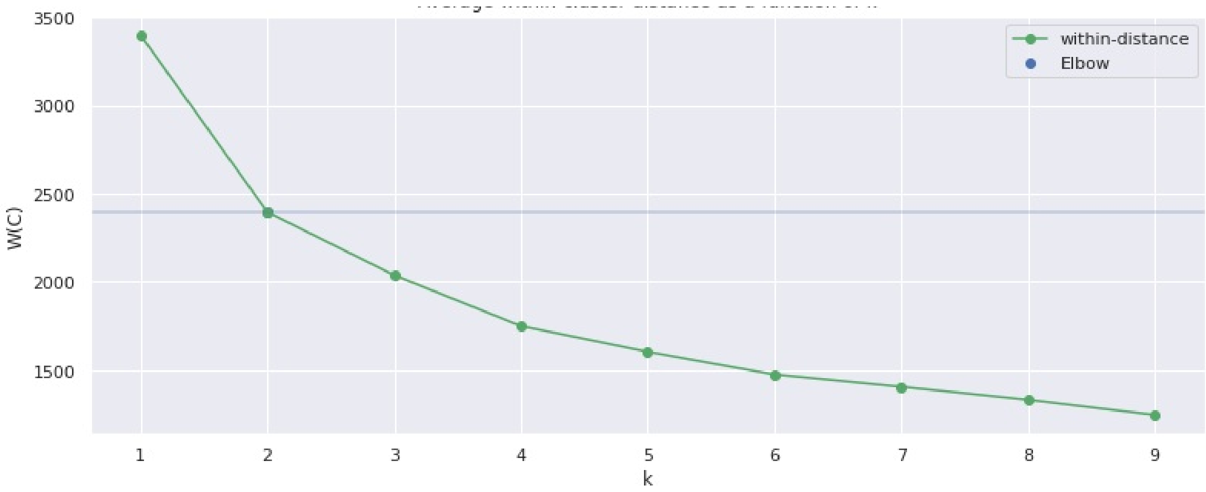
In [125...

```
### Loop over 100 random initialisation for k means between 1 and 9 ###

centroids, labels = [], []
max_iter = 15
W1 = np.zeros(9)

for i in range(100):
    for k in range(1,10):
        centroids, labels = k_means(k, F, max_iter)
        # compute within cluster distance
        W1[k-1] += within_distance2(F, k, labels, centroids)/100
```

In [126...

```
### Plot of the average within-cluster distance as a function of k ###

import matplotlib.pyplot as plt
import seaborn as sns
k = np.arange(1,10,1)

sns.set()

plt.figure(figsize=(13,5))
plt.plot(k, W1, '-o', color='g', label='within-distance')
plt.axhline(y=W1[k==2], c='b', alpha=0.3)
plt.scatter(k[1], W1[k==2], color='b', label='Elbow')
plt.title("Average within-cluster distance as a function of k")
plt.xlabel("k")
plt.ylabel("W(C)")
plt.legend()
plt.show()

k_optimal = k[2]
print('{} is the optimal number of clusters.'.format(k[1]))
```

Average within-cluster distance as a function of k




```
2 is the optimal number of clusters.
```

We see that k=2 is the optimal number of clusters when we change the range of k-means to look at. We will see later that actually the CH score agrees with that but nevertheless, the task asked to look for an elbow in the range $[2,10]$, in that case, our elbow comes for $k=4$ clusters.

Comment

From this method, the elbow seems to be around $k=4$. However it is not really obvious. In fact, if we plot on different numbers of clusters we see a clearer elbow at $k=2$. We will see later that in fact with another metric: the CH score confirms that different hypothesis of having in fact only 2 clusters.

2.1.2 Obtain optimal clustering for k means according to CH score

Now, let's code the Calinski-Harabasz score (CH score) and plot it as a function of increasing k to find the optimal clustering according to that score.

Essentially we want to find the k for which the CH score is the highest.

Mathematically, we can define the CH score as follows:

$CH = \frac{tr(W_k)}{tr(B_k)} \frac{n_E - k}{k-1}$, tr is the trace of the matrix

where we have defined

- the within distance: $W_k = \sum_{q=1}^k \sum_{x\in C_q} (x-c_q)(x-c_q)^T$
- the between distance: $B_k = \sum_{q=1}^k n_q (c_q-c_E)(c_q-c_E)^T$.

And where C_q is the set of points in cluster q, c_q is the center of the cluster q, c_E is the center of E and n_q the number of points in clusters q (for a set E of size n_E).

In [127…

```
### BETWEEN DISTANCE FUNCTION ###

def between_distance(X, k, labels, centroids):
    mean = np.mean(X, axis=0)
    distance = 0 # initialize summation

    for c in range(k): # loop over number of clusters
        idx = sum(j==c for j in labels) # number of points in cluster k
        distance += idx * ((centroids[c]-mean)@(centroids[c]-mean).T)
    return distance
```

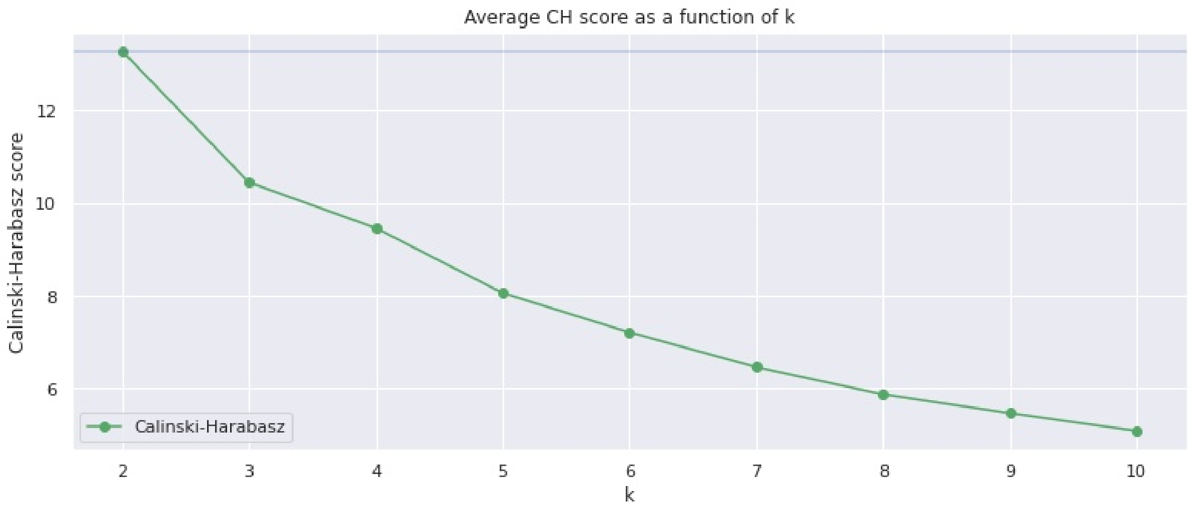
In [128…

```
### CALINSKI HARABASZ SCORE FUNCTION ###

def CH_score(X, k, labels, centroids):
    N = X.shape[0] # number of samples

    B = between_distance(X,k,labels,centroids) # compute between distance
    W = within_distance2(X,k,labels,centroids) # compute within distance

    CH = B/W * (N-k)/(k-1) # compute actual score
    return CH
```

Now, let's do 100 iterations over our k-means and computing our CH score to average it and plot.

In [129…
```
### Loop over 100 random initialisation for k means between 2 and 10 ###

centroids, labels = [], []
max_iter = 15
CH = np.zeros(9)

for j in range(100):
    for k in range(2,11):
        centroids, labels = k_means(k, F, max_iter)
        # compute CH score
        CH[k-2] += CH_score(F, k, labels, centroids)/100
```

In [130…
```
### Plot of the average CH score as a function of k ###

import matplotlib.pyplot as plt

k = np.arange(2,11,1)
sns.set()

plt.figure(figsize=(13,5))
plt.plot(k, CH,'-o', color='g', label='Calinski-Harabasz')
plt.axhline(y=max(CH), c='b', alpha=0.3)
plt.title("Average CH score as a function of k")
plt.xlabel("k")
plt.ylabel("Calinski-Harabasz score")
plt.legend()
plt.show()

k_optimal = k[np.argmax(CH)]
print('{} is the optimal number of clusters.'.format(k[np.argmax(CH)]))
```




```
2 is the optimal number of clusters.
```

Comment

Thanks to the CH score, we find that the best k is 2, because that's the k for which our CH score is the highest.

2.1.3 Evaluation of robustness of k means

In this part, we want to evaluate how robust our k-means clustering is. We will do that by exploring the variance of our k over 100 iterations and also by looking at the ARI score that is defined a bit later in the coursework.

In [131…
```
### Compute quartiles for our within-distance ###

max_iter = 15
W2, W3, W4, W5, W6, W7, W8, W9, W10 = [], [], [], [], [], [], [], [], []

for j in range(100):
    c2, l2 = k_means(2, F, max_iter)
    c3, l3 = k_means(3, F, max_iter)
    c4, l4 = k_means(4, F, max_iter)
    c5, l5 = k_means(5, F, max_iter)
    c6, l6 = k_means(6, F, max_iter)
    c7, l7 = k_means(7, F, max_iter)
    c8, l8 = k_means(8, F, max_iter)
    c9, l9 = k_means(9, F, max_iter)
    c10, l10 = k_means(10, F, max_iter)
    W2.append(within_distance2(F, 2, l2, c2))
    W3.append(within_distance2(F, 3, l3, c3))
    W4.append(within_distance2(F, 4, l4, c4))
```

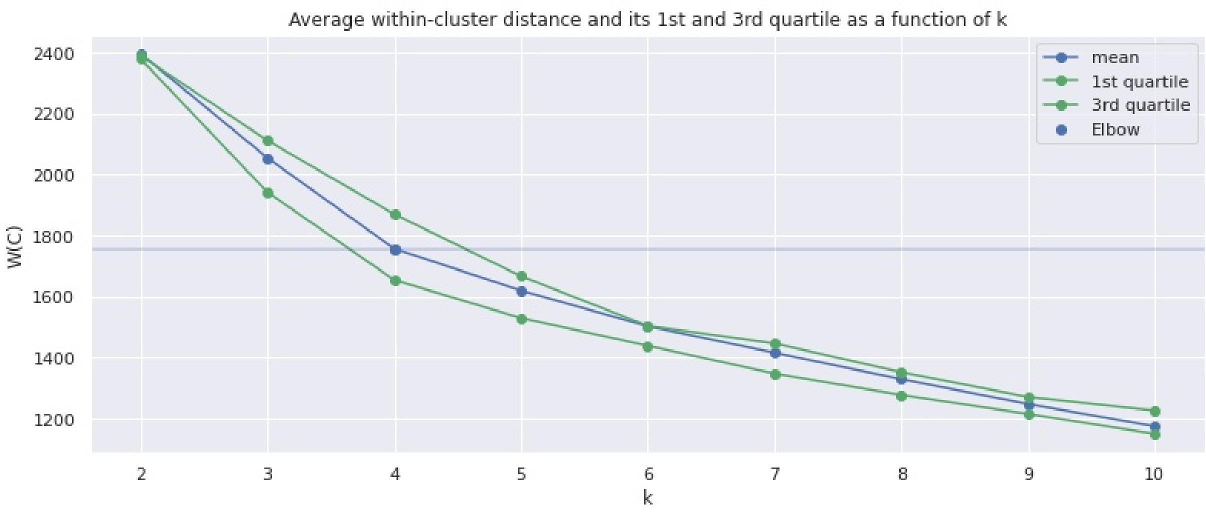
```
    W5.append(within_distance2(F, 5, l5, c5))
    W6.append(within_distance2(F, 6, l6, c6))
    W7.append(within_distance2(F, 7, l7, c7))
    W8.append(within_distance2(F, 8, l8, c8))
    W9.append(within_distance2(F, 9, l9, c9))
    W10.append(within_distance2(F, 10, l10, c10))
```

In [132…

```
### Compute 1st & 3rd quantile for each k and stack them ###

q21, q23 = np.percentile(W2, 25), np.percentile(W2, 75)
q31, q33 = np.percentile(W3, 25), np.percentile(W3, 75)
q41, q43 = np.percentile(W4, 25), np.percentile(W4, 75)
q51, q53 = np.percentile(W5, 25), np.percentile(W5, 75)
q61, q63 = np.percentile(W6, 25), np.percentile(W6, 75)
q71, q73 = np.percentile(W7, 25), np.percentile(W7, 75)
q81, q83 = np.percentile(W8, 25), np.percentile(W8, 75)
q91, q93 = np.percentile(W9, 25), np.percentile(W9, 75)
q101, q103 = np.percentile(W10, 25), np.percentile(W10, 75)

q1 = np.stack((q21, q31, q41, q51, q61, q71, q81, q91, q101))
q3 = np.stack((q23, q33, q43, q53, q63, q73, q83, q93, q103))
```

In [133…

```
### Plot of the average within-cluster distance and its 1st and 3rd quartile as a function of k ###
import matplotlib.pyplot as plt

k = np.arange(2,11,1)
sns.set()

plt.figure(figsize=(13,5))
plt.plot(k, W, '-o', color='b', label='mean')
plt.plot(k, q1, '-o', color='g', label='1st quartile')
plt.plot(k, q3, '-o', color='g', label='3rd quartile')

plt.axhline(y=W[k==4], c='b', alpha=0.3)
plt.scatter(k[2], W[k==4], color='b', label='Elbow')
plt.title("Average within-cluster distance and its 1st and 3rd quartile as a function of k")
plt.xlabel("k")
plt.ylabel("W(C)")
plt.legend()

plt.show()
```



Comment

This plot shows that for k=2, the variance of the within distance is extremely low (as 1st and 3rd quartile are on the same point as the mean), which makes our assumption that k=2 from the CH score is the best clustering fairly robutst. Because essentially it means that over 100 iterations, the clusterings are all very very similar. However, regarding k=3 and k=4 say, we see that the variance is wider so clustering is more volatile and less relient.

Now let's define the ARI score (which we explain in more details in 2.3.3 where it is asked). But essentially it measures similarity in two data clusterings. For us, it'd be interesting to use this metric to compare our clusterings solutions over random initialization for given k (number of clusters).

In [134…

```
### CREATE CONSISTENCY TABLE FUNCTION ### CHANGE ARI

def cons_t(c1, c2):

    N = c1.size # pick length of cluster 1 vector

    # pick vector from cluster 2 with the number of unique entries
```

```python
    n1 = np.unique(c2)
    N1 = n1.size

    # pick vector from cluster 2 with the number of unique entries
    n2 = np.unique(c1)
    N2 = n2.size

    # ininitialize consistency table to fill
    cons_t = np.zeros((N2+1, N1+1))

    for i in range(N): # loop over cluster size
        cons_t[c1[i], c2[i]] += 1 # update

    # define sum of rows and columns
    s_1 = np.sum(cons_t, axis=1) # column sum
    s_2 = np.sum(cons_t, axis=0) # row sum

    for j in range(N2): # loop over number of unique entries
        cons_t[j, N1] = s_1[j] # update

    for k in range(N1): # loop over number of unique entries
        cons_t[N2, k] = s_2[k] # update

    return cons_t
```

In [135…

```python
### CREATE ADJUSTED RAND INDEX FUNCTION ###

def ARI(c1, c2):

    # sums from top of fraction
    s_top_1 = 0
    s_top_2 = 0
    # sums from 2nd term of top of fraction
    s_top_21= 0
    s_top_22= 0

    # sums from bottom of fraction
    s_bottom_1 = 0
    s_bottom_2 = 0

    n = c1.size

    # choose unique values from c2
    n1 = np.unique(c2)
    N1 = n1.size

    # choose unique values from c1
    n2 = np.unique(c1)
    N2 = n2.size

    # create initial CT
    CT = cons_t(c1,c2)

    # double loop for first term of top of fraction
    for k in range(N2):
        for l in range(N1):
            s_top_1 += sc.special.binom(CT[k,l],2)

    # compute first sum of second term of top of fraction
    for k in range(N2):
        s_top_21 += sc.special.binom(CT[k,N1],2)

    # compute second sum of second term of top of fraction
    for l in range(N1):
        s_top_22 += sc.special.binom(CT[N2,l],2)

    # divide 2nd term of top of fraction by the combinatorics
    s_top_2 = s_top_21 * s_top_22 / sc.special.binom(n,2)

    # compute first sum from bottom of frac
    for k in range(N2):
        s_bottom_1 += sc.special.binom(CT[k,N1],2)

    for l in range(N1):
        s_bottom_1 += sc.special.binom(CT[N2,l],2)

    s_bottom_1 /= 2 #result for first sum from bottom of frac

    # compute seond sum from bottom of frac
    s_bottom_2 = s_top_2

    return (s_top_1 - s_top_2)/(s_bottom_1-s_bottom_2)
```

In [136…

```python
### LOOK AT ARI FOR 2 SOLUTIONS 100 ITERATIONS OF K-MEANS FOR K=2 ###
labels21=[]
labels22=[]
ARI2 = []
for i in range(100):
```

```
  centroids201, labels201 = k_means(2, F, 15)
  centroids202, labels202 = k_means(2, F, 15)
  labels21.append(np.array(labels201))
  labels22.append(np.array(labels202))
  ARI2.append(ARI(np.array(labels21[i-1]), np.array(labels22[i-1])))

kari2 = np.mean(ARI2)
```

In [137…

```
### LOOK AT ARI FOR 2 SOLUTIONS 100 ITERATIONS OF K-MEANS FOR K=3 ###

# initialize empty vectors to append
labels31=[]
labels32=[]
ARI3 = []
for i in range(100): # loop 100 times
  centroids301, labels301 = k_means(3, F, 15)
  centroids302, labels302 = k_means(3, F, 15)
  labels31.append(np.array(labels301)) # create a vector of 100 solutions of k-means for k=3
  labels32.append(np.array(labels302)) # create a vector of 100 solutions of k-means for k=3
  ARI3.append(ARI(np.array(labels31[i-1]), np.array(labels32[i-1]))) # compare pairwise

kari3 = np.mean(ARI3) # compute mean
```

In [138…

```
### LOOK AT ARI FOR 2 SOLUTIONS 100 ITERATIONS OF K-MEANS FOR K=4 ###
labels41=[]
labels42=[]
ARI4 = []
for i in range(100):
  centroids401, labels401 = k_means(4, F, 15)
  centroids402, labels402 = k_means(4, F, 15)
  labels41.append(np.array(labels401))
  labels42.append(np.array(labels402))
  ARI4.append(ARI(np.array(labels41[i-1]), np.array(labels42[i-1])))

kari4 = np.mean(ARI4)
```

In [139…

```
### LOOK AT ARI FOR 2 SOLUTIONS 100 ITERATIONS OF K-MEANS FOR K=5 ###
labels51=[]
labels52=[]
ARI5 = []
for i in range(100):
  centroids501, labels501 = k_means(5, F, 15)
  centroids502, labels502 = k_means(5, F, 15)
  labels51.append(np.array(labels501))
  labels52.append(np.array(labels502))
  ARI5.append(ARI(np.array(labels51[i-1]), np.array(labels52[i-1])))

kari5 = np.mean(ARI5)
```

In [140…

```
### LOOK AT ARI FOR 2 SOLUTIONS 100 ITERATIONS OF K-MEANS FOR K=6 ###
labels61=[]
labels62=[]
ARI6 = []
for i in range(100):
  centroids601, labels601 = k_means(6, F, 15)
  centroids602, labels602 = k_means(6, F, 15)
  labels61.append(np.array(labels601))
  labels62.append(np.array(labels602))
  ARI6.append(ARI(np.array(labels61[i-1]), np.array(labels62[i-1])))

kari6 = np.mean(ARI6)
```

In [141…

```
### LOOK AT ARI FOR 2 SOLUTIONS 100 ITERATIONS OF K-MEANS FOR K=7 ###
labels71=[]
labels72=[]
ARI7 = []
for i in range(100):
  centroids701, labels701 = k_means(7, F, 15)
  centroids702, labels702 = k_means(7, F, 15)
  labels71.append(np.array(labels701))
  labels72.append(np.array(labels702))
  ARI7.append(ARI(np.array(labels71[i-1]), np.array(labels72[i-1])))

kari7 = np.mean(ARI7)
```

In [142…

```
### LOOK AT ARI FOR 2 SOLUTIONS 100 ITERATIONS OF K-MEANS FOR K=8 ###
labels81=[]
labels82=[]
ARI8 = []
for i in range(100):
  centroids801, labels801 = k_means(8, F, 15)
  centroids802, labels802 = k_means(8, F, 15)
  labels81.append(np.array(labels801))
  labels82.append(np.array(labels802))
  ARI8.append(ARI(np.array(labels81[i-1]), np.array(labels82[i-1])))
```

```
kari8 = np.mean(ARI8)
```

In [143...

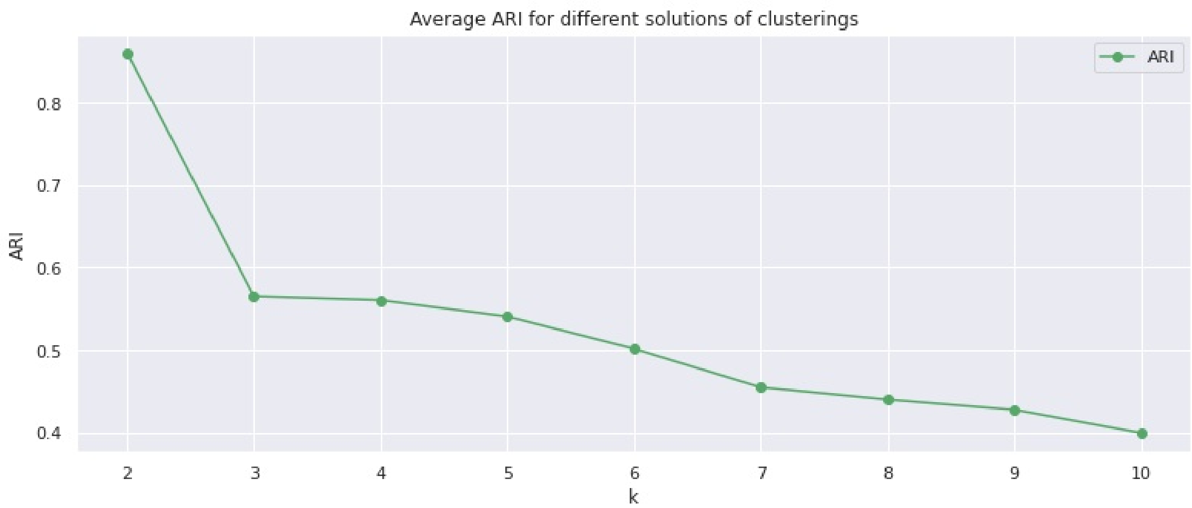
```
### LOOK AT ARI FOR 2 SOLUTIONS 100 ITERATIONS OF K-MEANS FOR K=9 ###
labels91=[]
labels92=[]
ARI9 = []
for i in range(100):
  centroids901, labels901 = k_means(9, F, 15)
  centroids902, labels902 = k_means(9, F, 15)
  labels91.append(np.array(labels901))
  labels92.append(np.array(labels902))
  ARI9.append(ARI(np.array(labels91[i-1]), np.array(labels92[i-1])))

kari9 = np.mean(ARI9)
```

In [144...

```
### LOOK AT ARI FOR 2 SOLUTIONS 100 ITERATIONS OF K-MEANS FOR K=10 ###
labels101=[]
labels102=[]
ARI10 = []
for i in range(100):
  centroids1001, labels1001 = k_means(10, F, 15)
  centroids1002, labels1002 = k_means(10, F, 15)
  labels101.append(np.array(labels1001))
  labels102.append(np.array(labels1002))
  ARI10.append(ARI(np.array(labels101[i-1]), np.array(labels102[i-1])))

kari10 = np.mean(ARI10)
```

In [145...

```
### PLOT RESULTS FOR ARI WRT TO DIFFERENT NUMBER OF CLUSTERS ###

kari= [kari2, kari3, kari4, kari5, kari6, kari7, kari8, kari9, kari10]
k = np.arange(2,11,1)
sns.set()

plt.figure(figsize=(13,5))
plt.plot(k, kari, '-o', color='g', label='ARI')
plt.title("Average ARI for different solutions of clusterings")
plt.xlabel("k")
plt.ylabel("ARI")
plt.legend()

plt.show()
```



Comment

The ARI score of our different solutions against the number of clusters shows that actually, the best k is k=2. Indeed it has the highest ARI score, which indicates that even throughout random initialization, our solutions for clustering are consistent with each other. However when we increase the number of clusters, as seen with the quartiles plot and this one too, the variability in clustering solutions increases and so the solutions are less robust.

In conclusion, the combination of the variance analysis (k=2) of the within distance, the ARI score (k=2) and the CH score (k=2) could push us to choose k=2 as the best cluster.

2.2 Dimensionality reduction of the feature matrix

In this part, we look at Principal Component Analysis to carry out dimensionality reduction on the feature matrix. Indeed we have 100 features on only 34 samples, so it is quite hard to understand how all these datapoints interact with each other. Therefore, let's try to capture as much variability in the data as possible in fewer dimensions.

2.2.0 Load & normalize data

In [146…
```
### IMPORT NECESSARY PACKAGES ###
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy as sc
```

In [147…
```
### LOAD FEATURE MATRIX ###
F = pd.read_csv("/content/feature_matrix_karate_club.csv")
F.drop(F.columns[[0]], axis=1, inplace=True) #delete first column
F.head()
```

Out[147…

	0	1	2	3	4	5	6	7	8	9	10	11	12	
0	0.148640	-0.187110	-0.109395	0.111767	-0.024913	-0.117825	-0.072237	0.109746	-0.088250	0.210684	0.150548	0.073093	0.123050	0.01
1	0.207001	-0.018979	-0.094483	0.013424	-0.106083	0.252485	0.092830	0.002718	0.224389	0.061539	-0.011964	-0.124062	0.363754	-0.18
2	0.093962	-0.262451	-0.033010	0.108551	0.008679	-0.077413	0.051345	0.082396	-0.020187	0.186536	0.089231	0.135319	0.090328	0.02
3	0.212280	-0.047187	-0.116486	-0.004627	0.025335	0.020590	-0.123587	0.059001	-0.034748	0.131113	0.042459	0.030327	0.337262	0.12
4	0.053653	0.122337	-0.135267	-0.017862	-0.012216	0.064982	-0.053867	0.120750	-0.060154	0.204295	-0.206539	-0.015263	0.217170	-0.14

5 rows × 100 columns

In [148…
```
### TRANSFORM TO NUMPY ARRAY###
F = F.to_numpy()
```

In [149…
```
### NORMALIZE FUNCTION ###
def normalize(X):
    mu = np.mean(X, axis=0)
    std = np.std(X, axis=0)
    std_filled = std.copy()
    std_filled[std==0] = 1.
    Xbar = ((X-mu)/std_filled)
    return Xbar
```

In [150…
```
F = normalize(F) # normalize feature
```

In [151…
```
print(F)
```

```
[[ 0.74887402 -1.39730361 -0.48929966 ... -0.28512148  0.13252166
   0.26538724]
 [ 1.53093264  0.25719319 -0.25244759 ... -1.10116311 -1.14727011
  -0.61985789]
 [ 0.01616094 -2.1387055   0.72392518 ... -1.85094005  0.10920858
   0.85215522]
 ...
 [-0.2830589   0.6513791   0.79281966 ...  0.57247145 -1.03630708
  -0.68431521]
 [ 0.9547694  -0.84102227 -0.95349772 ... -0.49432713  1.38469424
   0.73618725]
 [-1.29672531 -0.60330908 -0.38802066 ...  0.13062611 -1.18517828
  -0.28073908]]
```

2.2.1 Plot of data with respect to d-dimensional PCA space

In this first task, we are asked to plot our data in 3 different dimensions of the PCA space: for d=1, d=2 and d=3.

In order to do that, we need to implement dimensional reduction on our data as follows:

In [152…
```
### IMPLEMENT PRINCIPAL COMPONENT ANALYSIS from CT ###

from scipy.sparse import linalg

def pca_function(X, k):

    # create covariance matrix S
    C = 1/(len(X)-1) * X.T @ X

    # compute eigenvalues and eigenvectors using the eigsh scipy function
    eigenvalues, eigenvectors = linalg.eigsh(C, k, which="LM", return_eigenvectors=True)

    # sorting the eigenvectors and eigenvalues from largest to smallest eigenvalue
```

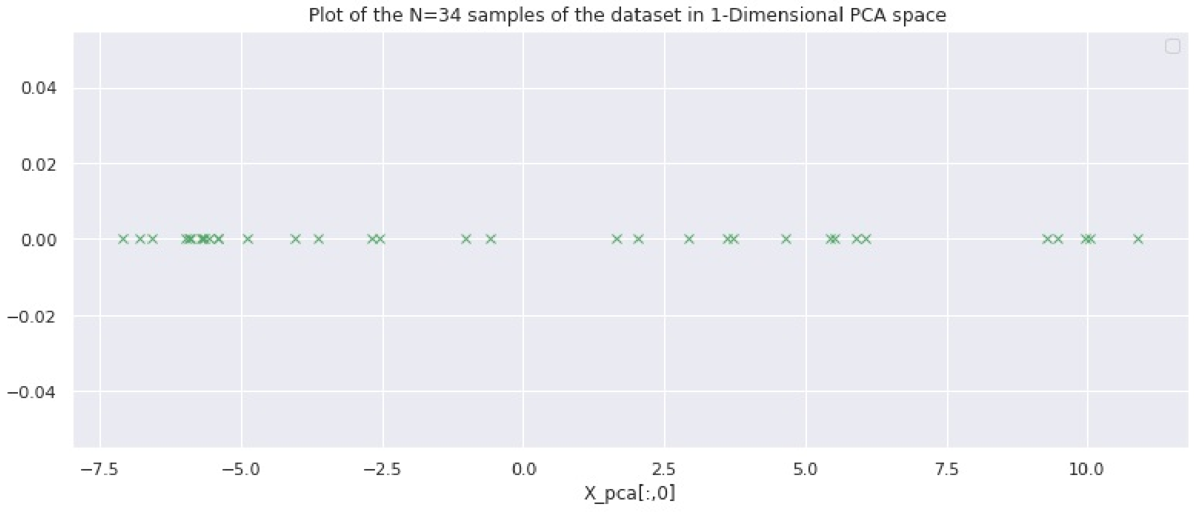
```
    sorted_index = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[sorted_index]
    eigenvectors =eigenvectors[:,sorted_index]

    # transform our data
    X_pca = X @ eigenvectors

    return X_pca, eigenvectors, eigenvalues
```

d=1 dimensional PCA space

In [153…

```
### Plot of the N=34 samples of the dataset in 1-Dimensional PCA space ###

d = 1
X_pca, eigenvectors, eigenvalues = pca_function(F,d) #evec, evals, projections
val = 0.
ar = X_pca[:,0]

sns.set()

plt.figure(figsize=(13,5))
plt.plot(ar, np.zeros_like(ar) + val,"x", c='g')
plt.title("Plot of the N=34 samples of the dataset in 1-Dimensional PCA space ")
plt.xlabel("X_pca[:,0]")
plt.legend()
plt.show()

explained_variances = eigenvalues / eigenvalues.sum()
print('The explained variance for the first principle component is: {}'.format(explained_variances))
```

```
No handles with labels found to put in legend.
```



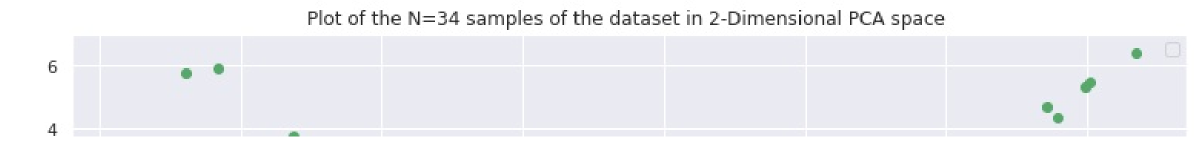

```
The explained variance for the first principle component is: [1.]
```

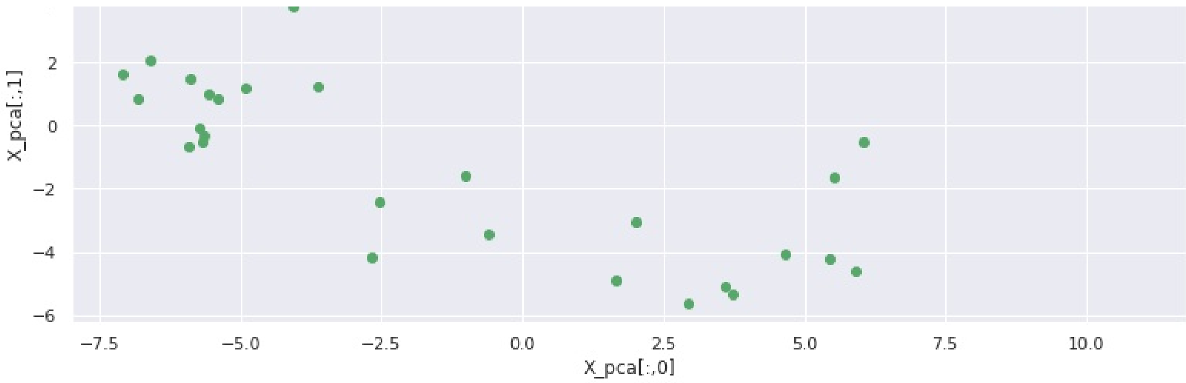
d=2 dimensional PCA space

In [154…

```
### Plot of the N=34 samples of the dataset in 2-Dimensional PCA space ###

d = 2
X_pca, eigenvectors, eigenvalues = pca_function(F,d) #evec, evals, projections

sns.set()

plt.figure(figsize=(13,5))
plt.scatter(X_pca[:,0], X_pca[:,1], c='g')
plt.title("Plot of the N=34 samples of the dataset in 2-Dimensional PCA space ")
plt.xlabel("X_pca[:,0]")
plt.ylabel("X_pca[:,1]")
plt.legend()
plt.show()

explained_variances = eigenvalues / eigenvalues.sum()
print('The explained variance for the first 2 principle components is: {}'.format(explained_variances))
```

```
No handles with labels found to put in legend.
```



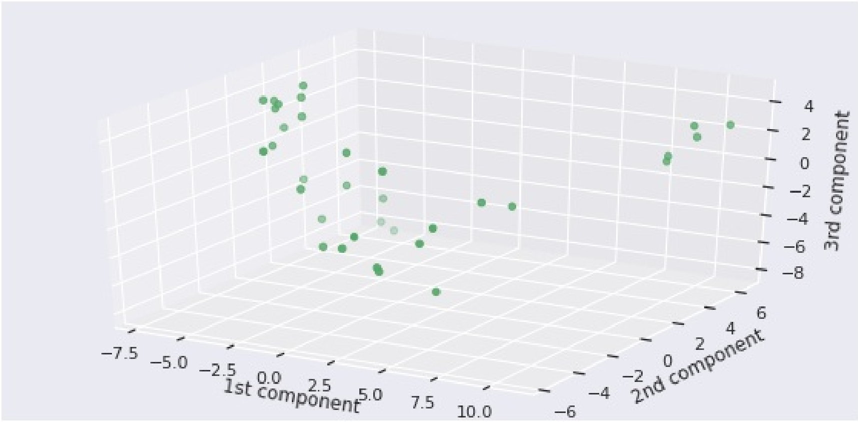



```
The explained variance for the first 2 principle components is: [0.72436226 0.27563774]
```

d=3 dimensional PCA space

In [155…

```python
### Plot of the N=34 samples of the dataset in 3-Dimensional PCA space ###
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

d = 3
X_pca, eigenvectors, eigenvalues = pca_function(F,d) # evec, evals, projections

sns.set()

# fix our axis values
fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(111, projection='3d')
xs = X_pca[:,0]
ys = X_pca[:,1]
zs = X_pca[:,2]

# plot
ax.scatter3D(xs, ys, zs, c='g')
ax.set_xlabel('1st component')
ax.set_ylabel('2nd component')
ax.set_zlabel('3rd component')
plt.show()

# compute explained variance
explained_variances = eigenvalues / eigenvalues.sum()
print('The explained variance for the first three principle components is: {}'.format(explained_variances))
```




```
The explained variance for the first three principle components is: [0.58964354 0.22437394 0.18598252]
```

Comment

Using PCA, and plotting in 1, 2 and 3-d PCA dimensions, we can see approximatively 3 clusters emerging, which agrees with the ARI score we computed in 2.1.3. But essentially our elbow method indicated to pick k=4 and CH score indicated k=2. So now we can visually see the data, we hesitate between k=2 and k=3.

In 1-D there clearly is one cluster on the left, and more the the right there are two smaller clusters which maybe could be one cluster. In 2-D, same idea, we see one defined cluster on the top left, but it's less obvious as to whether there's one additional one or 2 additional ones. In 3-D, however, it seems that there is actually 2 clusters, which is consistent with what we have found from the 100 initialization of our k-means algorithm. Overall, one might lean toward k=3 clusters because of our robustness analysis of k=2 in k-means.

2.2.2 Explained variance in PCA

Now we want to explore the proportion of explained variance of the PCA approximations of reduced dimensionality in the range for d in $[1,10]$.

Essentially, explained variance for one eigenvalue is just $\frac{e_i}{\sum_i e_i}$ for e_i an eigenvalue of our feature matrix.

In [156…

```
### Loop to create vector of explained variance of the PCA for dimensions between 1 and 10 ###

e_var = []

for d in range(1,11):
    X_pca, eigenvectors, eigenvalues = pca_function(F,d)
    e_var.append(eigenvalues / eigenvalues.sum())

print("Explained variance is:", pd.DataFrame(e_var))
```

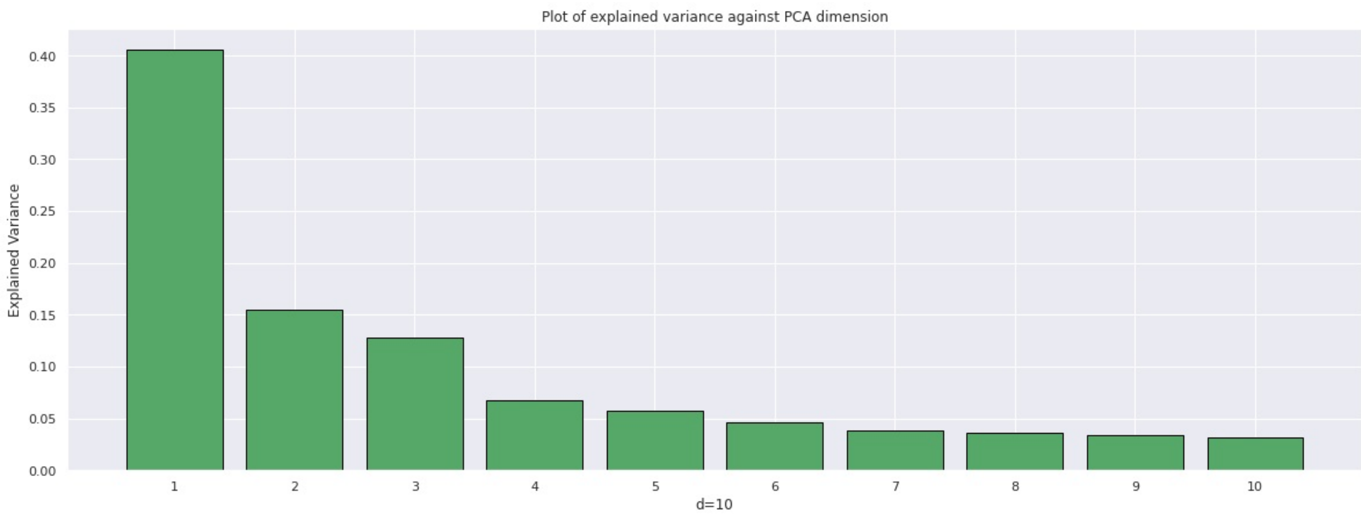
```
Explained variance is:           0         1         2  ...         7         8         9
0  1.000000       NaN       NaN  ...       NaN       NaN       NaN
1  0.724362  0.275638       NaN  ...       NaN       NaN       NaN
2  0.589644  0.224374  0.185983  ...       NaN       NaN       NaN
3  0.537290  0.204452  0.169469  ...       NaN       NaN       NaN
4  0.499294  0.189994  0.157485  ...       NaN       NaN       NaN
5  0.472425  0.179769  0.149010  ...       NaN       NaN       NaN
6  0.451933  0.171971  0.142546  ...       NaN       NaN       NaN
7  0.434250  0.165243  0.136969  ...  0.039126       NaN       NaN
8  0.419268  0.159542  0.132244  ...  0.037777  0.034501       NaN
9  0.405868  0.154443  0.128017  ...  0.036569  0.033398  0.031961

[10 rows x 10 columns]
```

Comment

Thanks to this table, we can see that the first 3 biggest eigenvalues account for more than 2/3 of the variability of the data when d=10, so we could make a fair assumption to reduce to d=3 when analysing data.

In [157…

```
### Plot of the explained variances of the PCA for dimensions between 1 and 10 ###

sns.set()

plt.figure(figsize=(20,7))
plt.bar(['1', '2', '3', '4', '5', '6', '7', '8', '9', '10'], e_var[9], color='g', edgecolor='black')
plt.xlabel('d=10')
plt.ylabel('Explained Variance')
plt.title('Plot of explained variance against PCA dimension')

plt.show()
```



Comment

The spectral decomposition of F^TF is analogous to PCA as per lecture notes. Indeed PCA approximates our feature matrix to a lower rank matrix by taking the first k values from SVD. When we look at the plot for d=10, we see that clearly the first 3 components accounts for around 68% of the variability of the model. This is satisfying because more than two thirds of the data is explained in only 3 dimensions, compared to the 100 features in the beginning from which it was complicated to make inference from.

Mathematically speaking (from the PCA CT), we have that: the eigenvalues from the correlation matrix $\frac{1}{n-1}F^TF$ are the

λ_i and the eigenvalues of F^TF (matrix of singular values) are the s_i. Essentially we have that $\lambda_i = s_i^2/(n-1)$.

The sum of all the eigenvalues represents the whole variance, hence that's why we look at explained variance, which is just $\frac{e_i}{\sum_i e_i}$.

Hence the conclusions from above.

2.3 Graph-based analysis

Now let's implement a graph based analysis of the adjacency matrix A which corresponds to friendships from the karate club.

2.3.0 Load data

In [158…

```
### LOAD NECESSARY PACKAGES ###
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy as sc
import networkx as nx
```

In [159…

```
### Adjacency matrix A ###
A = pd.read_csv("/content/karate_club_graph.csv")
A.drop(A.columns[[0]], axis=1, inplace=True) #delete first column
A.head()
```

Out[159…

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0	1.0	1.0	0.0
1	1.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
2	1.0	1.0	0.0	1.0	0.0	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0
3	1.0	1.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0
4	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0

	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	3
0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
1	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0

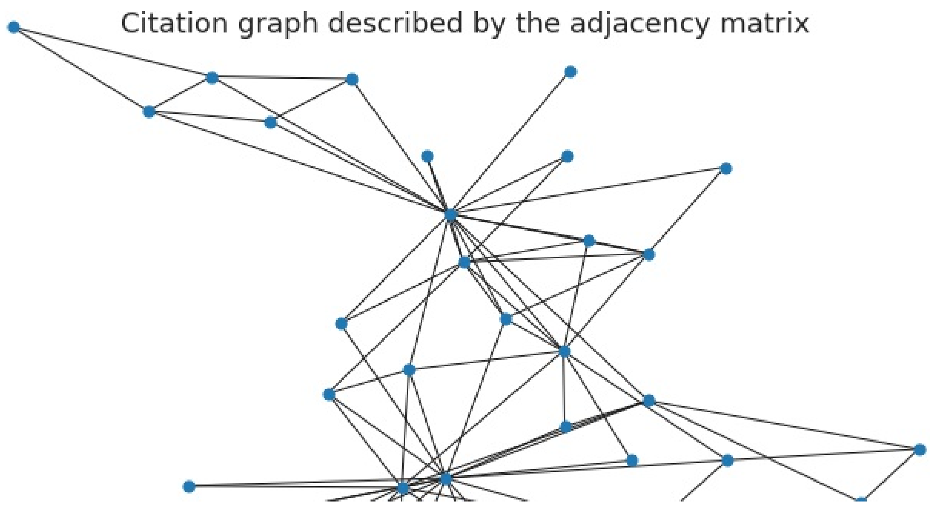
In [160…

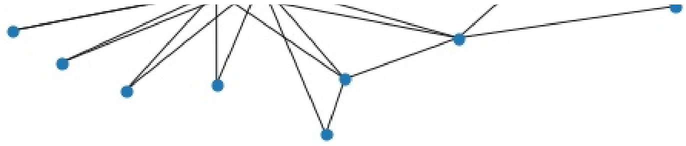
```
A.shape
```

Out[160… (34, 34)

In [163…

```
### PLOT CITATION GRAPH FOR THE ADJACENCY MATRIX ###

import networkx as nx
plt.figure(figsize=(9,6))

# defining the graph
G = nx.from_numpy_matrix(np.array(A))

# drawing the citation graph
pos = nx.spring_layout(G)
nx.draw(G, pos, node_size=50)

plt.suptitle('Citation graph described by the adjacency matrix', fontsize=18)

plt.show()
```





2.3.1 Centralities

In this task we want to obtain 3 measures of centrality:

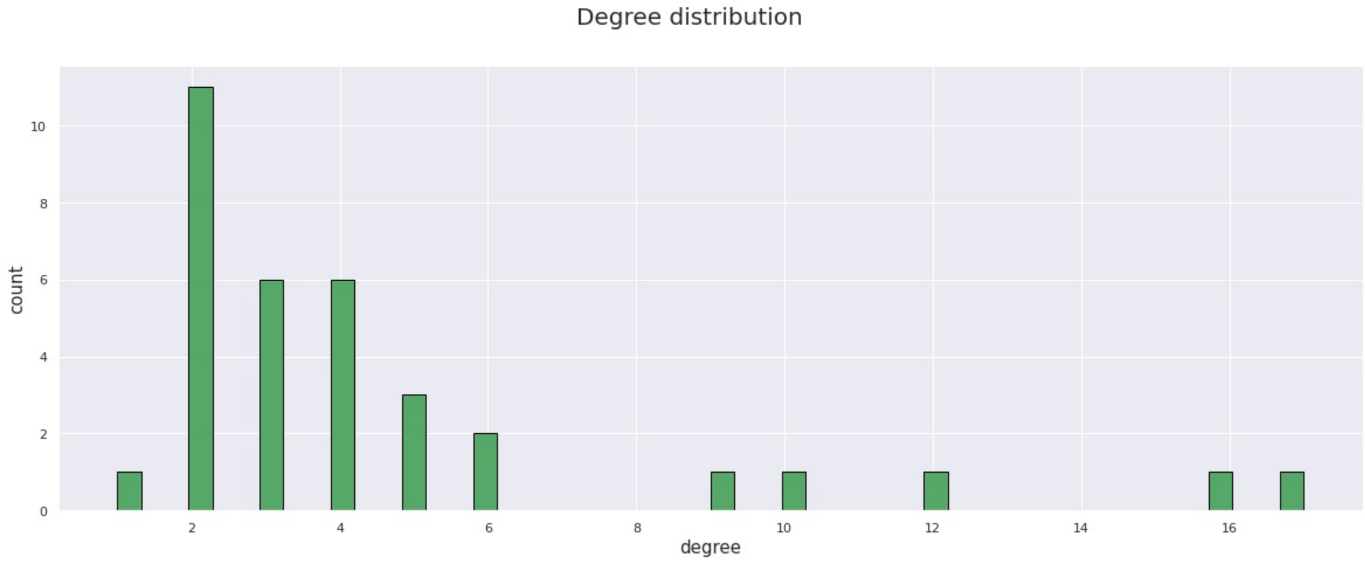
1. Degree Centrality: which computes the number of edges attached to one node (we divide it by twice the number of edges to normalise it as per the formula: $c_d = \frac{A\textbf{1}}{2E}$)
2. PageRank: which is a weighted random walk on the graph with the following formula: $c_{PR} = \alpha (AD^{-1})c_{PR} + (1-\alpha)\frac{1}{N} $, $\alpha \in [0,1]$ and setting $\alpha=0.85$ is a common practice
3. Eigenvector Centrality: which assigns higher centrality to nodes that are themselves connected to other highly central nodes: $Ac_e = \lambda c_e$

In [164…
```
### MAKE A INTO AN ARRAY ###
A = A.to_numpy()
```

In [165…
```
### Find degree of each node ###
degree = A.sum(axis=1)
degree
```

Out[165…
```
array([16.,  9., 10.,  6.,  3.,  4.,  4.,  4.,  5.,  2.,  3.,  1.,  2.,
        5.,  2.,  2.,  2.,  2.,  2.,  3.,  2.,  2.,  2.,  5.,  3.,  3.,
        2.,  4.,  3.,  4.,  4.,  6., 12., 17.])
```

In [166…
```
### PLOT DISTRIBUTION OF DEGREE OF NODES ###

import seaborn as sns
sns.set() # degree distribution of graph

plt.figure(figsize=(20,7))
plt.hist(degree, bins=50, color='g', edgecolor='black')

plt.xlabel('degree', fontsize=15)
plt.ylabel('count', fontsize=15)
plt.suptitle('Degree distribution', fontsize=20)

plt.show()
```



Comment

Here we look at the distribution of the nodes against their degrees and we see that there are 11 nodes with degree 2 and then then there's a decreasing amount of nodes that have higher degrees.

DEGREE CENTRALITY

In [167…

```
### Degree Centrality function ###

def degree_centrality(A):

    N = A.shape[0]
    d = A @ np.ones(N)
    return d / np.sum(d)
```

In [168…

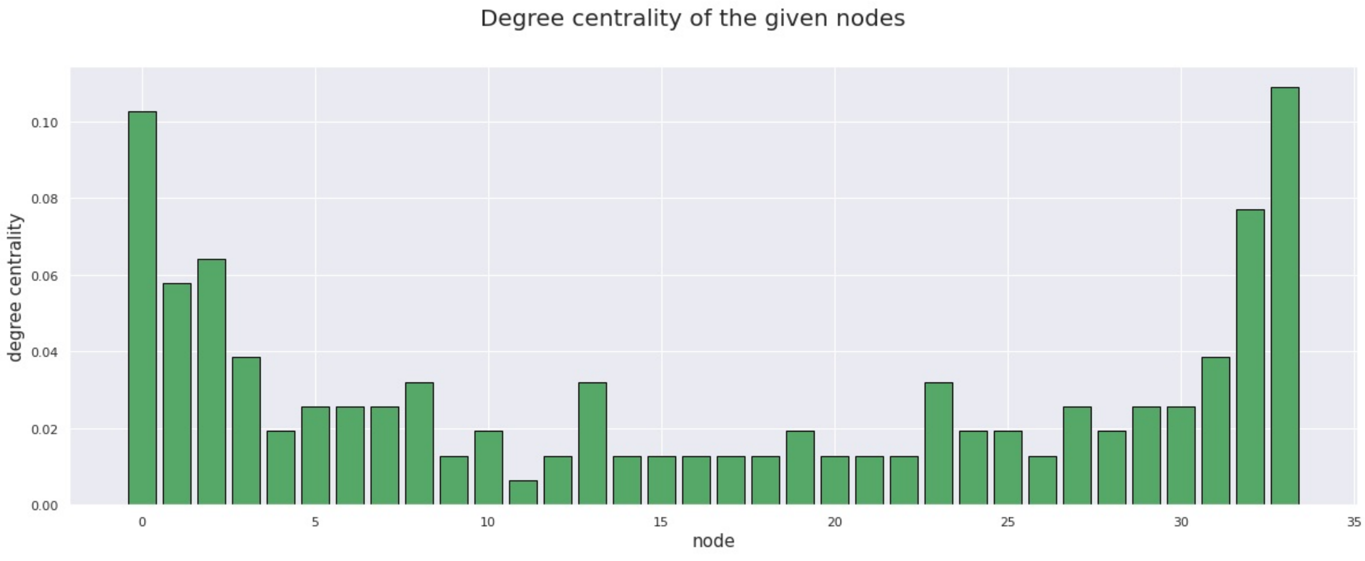
```
### DISPLAY DEGREE CENTRALITY VALUES ###
degree_list = degree_centrality(A)
pd.DataFrame(np.array(degree_list).reshape(1,34)).head()
```

Out[168…

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0.102564	0.057692	0.064103	0.038462	0.019231	0.025641	0.025641	0.025641	0.032051	0.012821	0.019231	0.00641	0.012821	0.032051

In [169…

```
### PLOT DEGREE CENTRALITY OF NODES ###

nodes = np.arange(0,34,1)

sns.set()

plt.figure(figsize=(20,7))
plt.bar(nodes,degree_list, color='g', edgecolor='black')

plt.xlabel('node', fontsize=15)
plt.ylabel('degree centrality', fontsize=15)
plt.suptitle('Degree centrality of the given nodes', fontsize=20)

plt.show()
```



In [170…

```
print('Node with maximal degree centrality:', nodes[np.argmax(degree_list)])
```

```
Node with maximal degree centrality: 33
```

Comment

Thanks to this plot of the distribution of the degreee centrality with respect to the node numbers, we can see that there are nodes which have the same degree centrality (when we rank them, note that we might get lists a bit varying even though it's ranking correctly), and some nodes stand out for particularly small or high values, such as node 11 has very small degree centrality and node 33 has highest degree centrality.

PAGERANK

In [171…

```
### PAGERANK FUNCTION ###

def trans_M(A):
    N1 = A.shape[0]
    d = A @ np.ones(N1)
    D = np.diag(d)
    M = np.linalg.inv(D)@A
    return M

def pagerank(A, num_iterations: int = 100, d: float = 0.85):
    M = (trans_M(A)).T
```

```
    N2 = M.shape[1]
    pr_cent = np.random.rand(N2, 1)
    pr_cent = pr_cent / np.linalg.norm(pr_cent, 1)
    M_pred = (d * M + (1 - d) / N2)

    for i in range(num_iterations):
        pr_cent = M_pred @ pr_cent
    return pr_cent.reshape(1,34).ravel()
```

In [172…

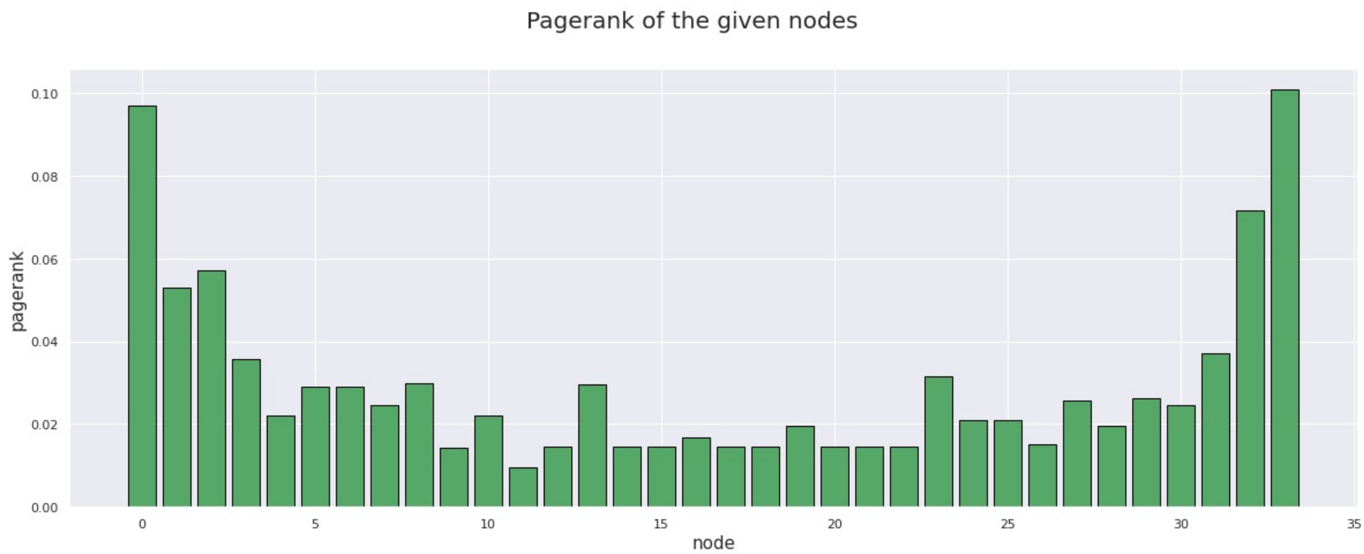
```
### DISPLAY PAGERANK VALUE ###
pagerank_list = pagerank(A)
pd.DataFrame(np.array(pagerank_list).reshape(1,34)).head()
```

Out[172…

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
0	0.096997	0.052877	0.057079	0.03586	0.021978	0.029111	0.029111	0.02449	0.029766	0.014309	0.021978	0.009565	0.014645	0.029536	0.(

In [173…

```
### PLOT PAGERANK OF GIVEN NODES ###

nodes = np.arange(0,34,1)

sns.set()

plt.figure(figsize=(20,7))
plt.bar(nodes, pagerank_list, color='g', edgecolor='black')

plt.xlabel('node', fontsize=15)
plt.ylabel('pagerank', fontsize=15)
plt.suptitle('Pagerank of the given nodes', fontsize=20)

plt.show()
```



In [174…

```
print('Node with maximal pagerank:', nodes[np.argmax(pagerank_list)])
```

```
Node with maximal pagerank: 33
```

Comment

Similarly to the degree centrality distribution, we can see some nodes with the same pagerank values, and others standing out whether it's small or high values. This is because the graph isn't uniform and it allows to depict relationship between friends.

EIGENVECTOR CENTRALITY

In [175…

```
### Eigenvector Centrality function ###
from scipy.sparse import linalg

def eigenvector_centrality(A):
    eigenvalue, eigenvector = linalg.eigsh(A, 1, which="LM", return_eigenvectors=True)
    return np.abs(eigenvector).reshape(1,34).ravel()
```

In [176…

```
### DISPLAY EIGENVECTOR CENTRALITY ###
eigenvector_list = eigenvector_centrality(A)
pd.DataFrame(np.array(eigenvector_list).reshape(1,34)).head()
```

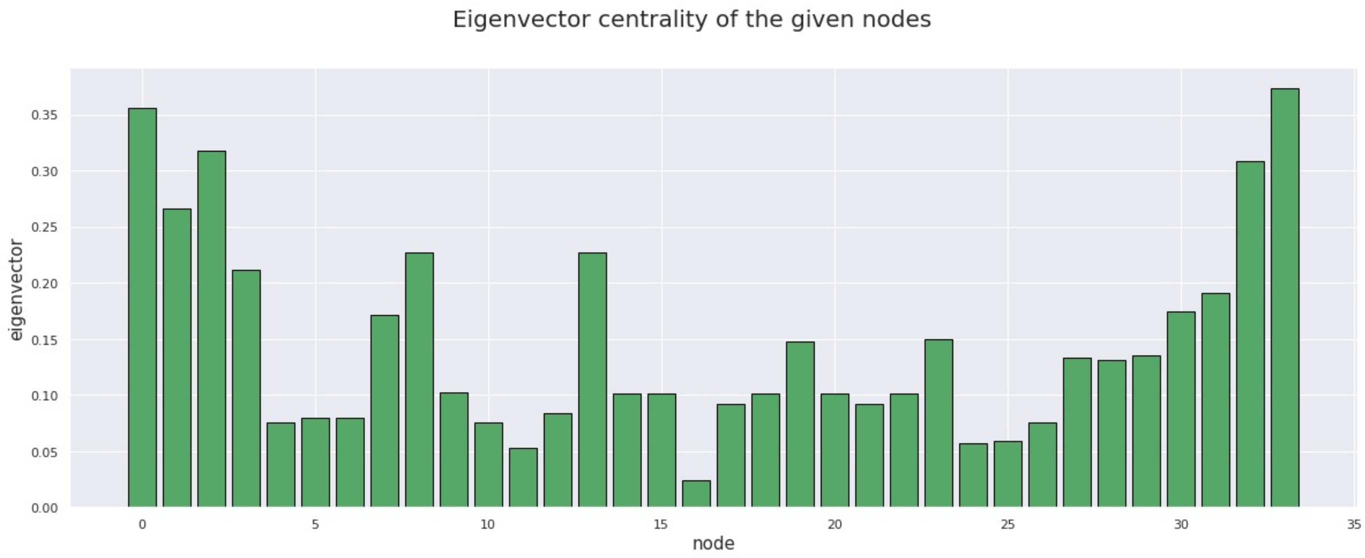
Out[176…

	0	1	2	3	4	5	6	7	8	9	10	11	12	13

0	0.355491	0.26596	0.317193	0.21118	0.075969	0.079483	0.079483	0.17096	0.227404	0.102674	0.075969	0.052856	0.084255	0.226473	0.1(

In [177…

```
### PLOT EIGENVECTOR CENTRALITY OF GIVEN NODES ###
nodes = np.arange(0,34,1)

sns.set()

plt.figure(figsize=(20,7))
plt.bar(nodes, eigenvector_list, color='g', edgecolor='black')

plt.xlabel('node', fontsize=15)
plt.ylabel('eigenvector', fontsize=15)
plt.suptitle('Eigenvector centrality of the given nodes', fontsize=20)

plt.show()
```



In [178…

```
print('Node with maximal eigenvector centrality:', nodes[np.argmax(eigenvector_list)])
```

```
Node with maximal eigenvector centrality: 33
```

Comment

Contrary to the degree centrality and the pagerank distributions, in the eigenvector centrality there are more values with higher eigenvector centrality in the middle nodes, while still having peaks at extremity nodes.

Now, we want to look at the similarity between the centrality measures and the associated node rankings.

In [179…

```
### DISPLAY IN A DATAFRAME THE VALUES AND PREVIEW###
B = np.array([degree_centrality(A), eigenvector_centrality(A), pagerank(A)])
B = pd.DataFrame(B.T)
B.columns = ['Degree', 'PageRank', 'Eigenvector Centrality']
B.head()
```

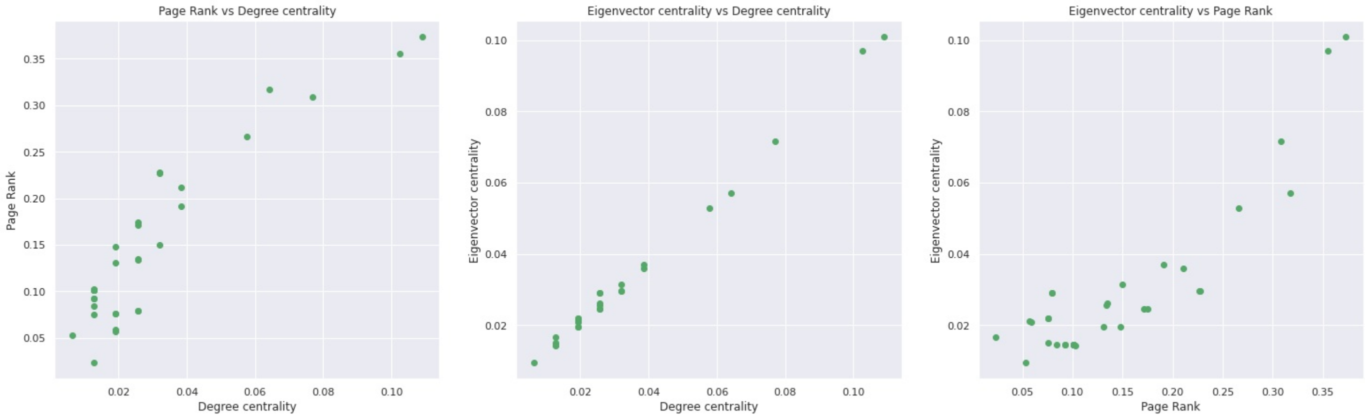
Out[179…

	Degree	PageRank	Eigenvector Centrality
0	0.102564	0.355491	0.096997
1	0.057692	0.265960	0.052877
2	0.064103	0.317193	0.057079
3	0.038462	0.211180	0.035860
4	0.019231	0.075969	0.021978

In [180…

```
### PLOT VALUES OF DEGREE CENTRALITIES AGAINST EACH OTHER   ###

B = B.to_numpy()

fig = plt.figure(figsize=(25, 7))

sns.set()

fig.add_subplot(131)
plt.scatter(B[:,0], B[:,1], c='g')
```

```
plt.xlabel("Degree centrality")
plt.ylabel("Page Rank")
plt.title("Page Rank vs Degree centrality")

fig.add_subplot(132)
plt.scatter(B[:,0], B[:,2], c='g')
plt.xlabel("Degree centrality")
plt.ylabel("Eigenvector centrality")
plt.title("Eigenvector centrality vs Degree centrality")

fig.add_subplot(133)
plt.scatter(B[:,1], B[:,2], c='g')
plt.xlabel("Page Rank")
plt.ylabel("Eigenvector centrality")
plt.title("Eigenvector centrality vs Page Rank")

plt.show()
```

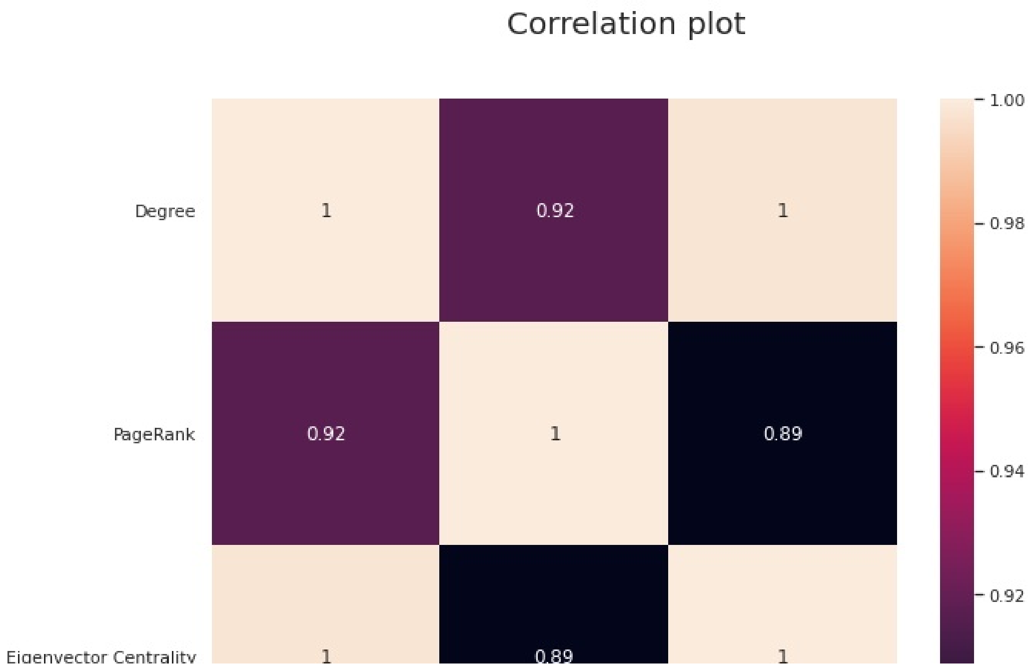


Comment

These scatter plots of our different degree centrality measures for their values show the relationships between them. Indeed we see an almost linear relationship between eigenvector centrality and degree centrality, whilst between the other 2 pairs, it seems that pagerank skews a bit the relationship.

In [181…

```
### PLOT CORRELATION MATRIX OF THE VALUES ###

sns.set()

plt.figure(figsize=(10,8))

# computing correlation matrix of rankings
values_corr = pd.DataFrame(B).corr()

xy_labels = ['Degree', 'PageRank', 'Eigenvector Centrality']
ax = sns.heatmap(values_corr, annot=True, xticklabels=xy_labels, yticklabels=xy_labels)

plt.yticks(rotation = 0)
plt.suptitle('Correlation plot', fontsize=20)

plt.show()
```



Comment

This correlation matrix shows us that in terms of values, the eigenvector centrality and the degree centrality are highly correlated. Regarding PageRank and degree centrality, it seems that page rank skews a bit the degree centrality and the PageRank skews a bit the eigenvector centrality.

And in general all these measures are highly correlated because they define some kind of centrality over the same graph.

However, what we want to look at is rather the ranking of the nodes associated to the order of the values of our centrality measures.

Now, let's look at the node rankings according to the different centrlaity measures:

In [182…

```
### SORT IN DESCENDING ORDER THE CENTRALITY MEASURES OF OUR GRAPH  ####

sorted_deg = np.array([x for _,x in sorted(zip(degree_centrality(A), nodes), reverse=True)])
sorted_pr = np.array([x for _,x in sorted(zip(pagerank(A), nodes), reverse=True)])
sorted_ec = np.array([x for _,x in sorted(zip(eigenvector_centrality(A), nodes), reverse=True)])
```

In [183…

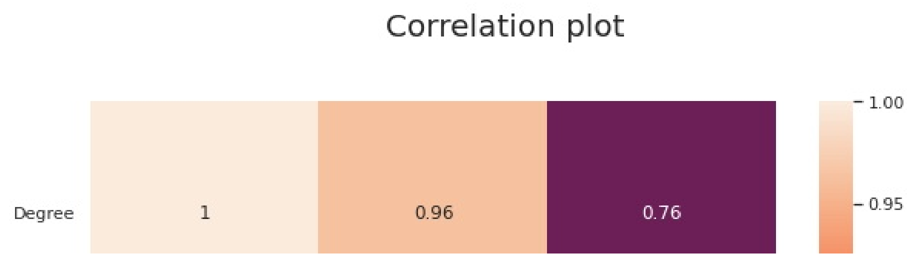
```
### ORDER THE VALUES TO PLOT CORRELATION MATRIX ###

# initialization
rankings = np.zeros((len(sorted_pr),3))

# for loop
for i in range(len(sorted_pr)):
    # ranking for degree centrality
    rankings[sorted_deg[i],0] = i+1
    # ranking for pagerank
    rankings[sorted_pr[i],1] = i+1
    # ranking for eigenvector centrality
    rankings[sorted_ec[i],2] = i+1

# result
rankings_df = pd.DataFrame(rankings).head(30)
rankings_df.columns = ['Degree','PageRank','Eigenvector Centrality']
rankings_df.head()
```

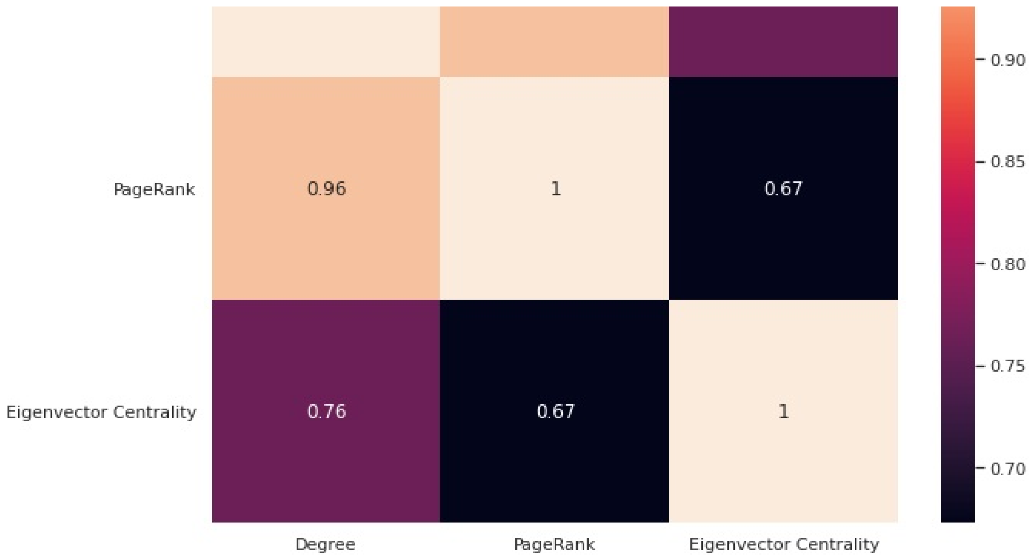
Out[183…

	Degree	PageRank	Eigenvector Centrality
0	2.0	2.0	2.0
1	5.0	5.0	5.0
2	4.0	4.0	3.0
3	7.0	7.0	8.0
4	22.0	18.0	29.0

In [184…

```
### PLOT CORRELATION MATRIX OF THE RANKINGS

sns.set()

plt.figure(figsize=(10,8))

# computing correlation matrix of rankings
ranking_corr = pd.DataFrame(rankings).corr()

xy_labels = ['Degree', 'PageRank', 'Eigenvector Centrality']
ax = sns.heatmap(ranking_corr, annot=True, xticklabels=xy_labels, yticklabels=xy_labels)

plt.yticks(rotation = 0)
plt.suptitle('Correlation plot', fontsize=20)

plt.show()
```





Comment on confusion matrix

Clearly we can see that there is a high correlation in the node rankings for PageRank and degree centrality, whilst less obvious for degree and eigenvector centrality, and pagerank and eigenvector centrality.

As we will see later on in the distribution of the top 8 most central nodes according to degree centrality and pagerank, it will confirm our result from this confusion matrix that these rankings are highly correlated (indeed we'll see that the distributions are the same).

In general these three methods are quite different so it's not that surprising that it's not all highly correlated.

2.3.2 Community detection

In this task, we use the NetworkX library to implement a Clauset-Newman-Moore greedy modularity maximisation algorithm in order to compute the optimal number of communities.

In [185…

```
### IMPORT NECESSARY LIBRARIES ###
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy as sc
import networkx as nx
```

In [186…

```
### Adjacency matrix A ###
A = pd.read_csv("/content/karate_club_graph.csv")
A.drop(A.columns[[0]], axis=1, inplace=True) #delete first column
A.head()
```

Out[186…

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	3
0	0.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
1	1.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1
2	1.0	1.0	0.0	1.0	0.0	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0
3	1.0	1.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
4	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0

In [187…

```
### COMPUTE COMMUNITIES ###
from networkx.algorithms.community import greedy_modularity_communities
G = nx.from_numpy_matrix(np.array(A))
pos = nx.spring_layout(G)

CNM_list = list(greedy_modularity_communities(G)) # labels

communities = [sorted(CNM_list[i]) for i in range(len(CNM_list))]
```

In [188…

```
print('Optimal number of communities k*:', len(communities))
```

```
Optimal number of communities k*: 3
```

Now, let's print the corresponding partitions of the karate club graph to the optimal number of communities k=3.

In [189…

```
### PRINT PARTITIONS OF OUR NODES WITHIN THE SAME COMMUNITY ###
partition = pd.DataFrame({'communities': np.arange(0,3,1),'partitions': np.array(communities)})
partition.set_index('communities', inplace=True)
partition.head(3)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2: VisibleDeprecationWarning: Creating an ndarray fr
om ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or sha
pes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray
```

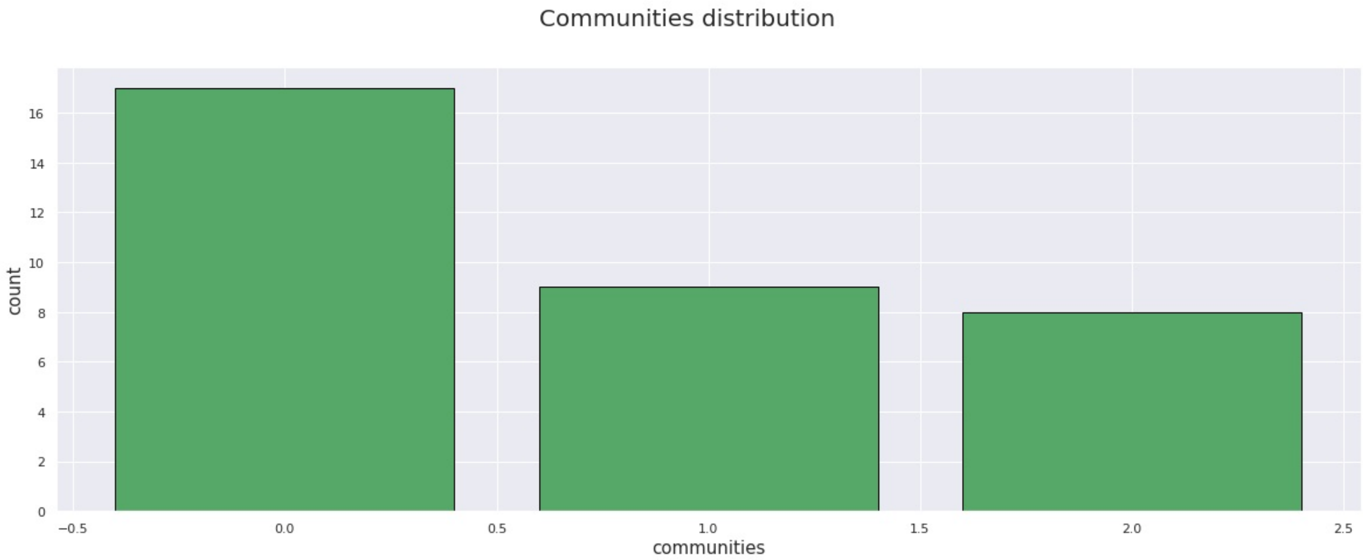
Out[189…

	partitions
communities	
0	[8, 14, 15, 18, 20, 22, 23, 24, 25, 26, 27, 28...
1	[1, 2, 3, 7, 9, 12, 13, 17, 21]
2	[0, 4, 5, 6, 10, 11, 16, 19]

Now plot the communities distribution to see their size relative to each other.

In [190…

```
### COMMUNITIES DISTRIBUTION PLOT ###

# initialization
partition_dist = np.zeros(3)

# for loop: computing size of each community
for i in range(len(communities)):
    partition_dist[i] = (len(communities[i]))

# plot
sns.set()

plt.figure(figsize=(20,7))
plt.bar(np.arange(0,3), partition_dist, color='g', edgecolor='black')
plt.xlabel('communities', fontsize=15)
plt.ylabel('count', fontsize=15)
plt.suptitle('Communities distribution', fontsize=20)

plt.show()
```



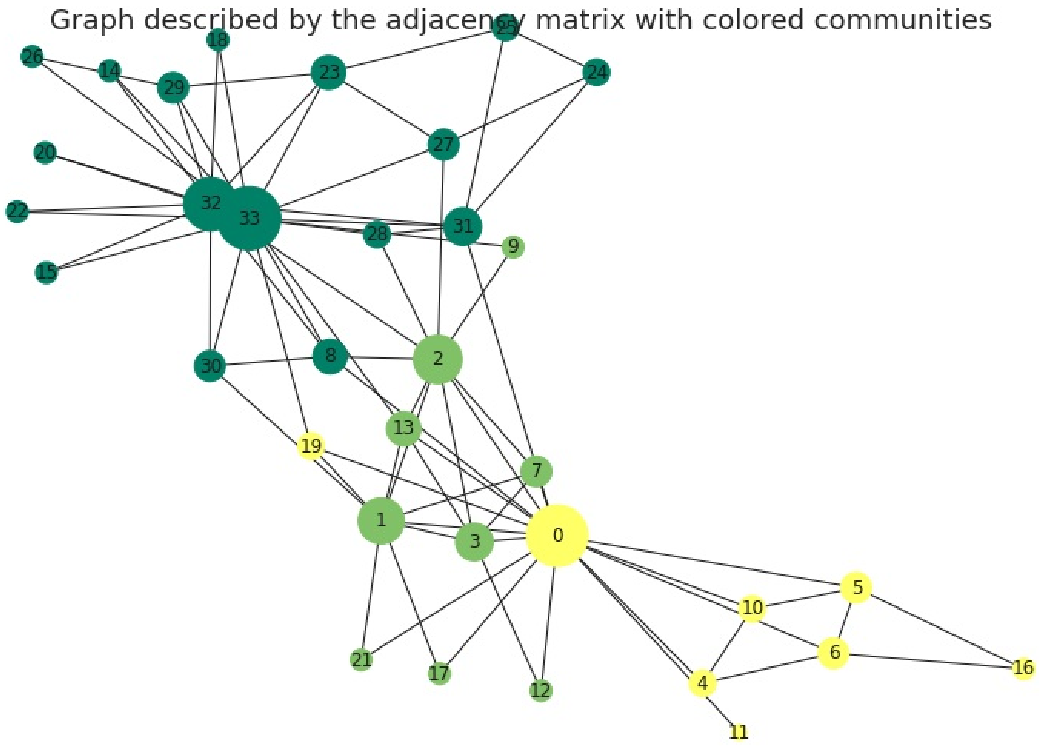
In [191…

```
### FIND CLUSTERS FOR OUR BEST NUMBER OF COMMUNITIES ###

# initialization
n = np.arange(len(communities)) # pick range of number of communitities
color_map1 = [0]*34 # choose how to color the graph

# for loop: labeling each node by its community number
for i in range(len(n)): # loop over number of communities
    for j in range(len(communities[i])): # loop over size of each community
        color_map1[communities[i][j]] = i
```

In [192…

```
### GRAPH OF ADJACENCY MATRIX WRT TO THEIR COMMUNITIES ###

deg = np.array(degree)*100 # pre-make size of node for better visualization

sns.set_style("white")
```

```python
plt.figure(figsize=(10,7))
nx.draw(G, pos, node_size=deg, cmap='summer', node_color=color_map1, with_labels = True)
plt.suptitle('Graph described by the adjacency matrix with colored communities', fontsize=18)

plt.show()
```



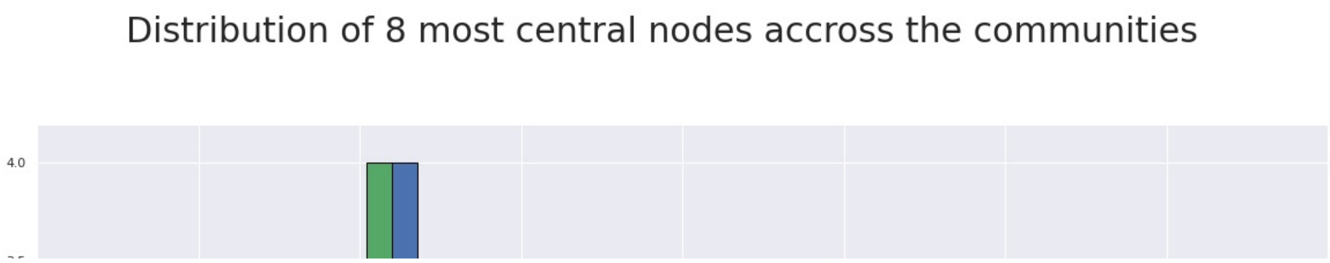
Now let's plot the distribution of the top 8 most central nodes according to degree centrality and PageRank:

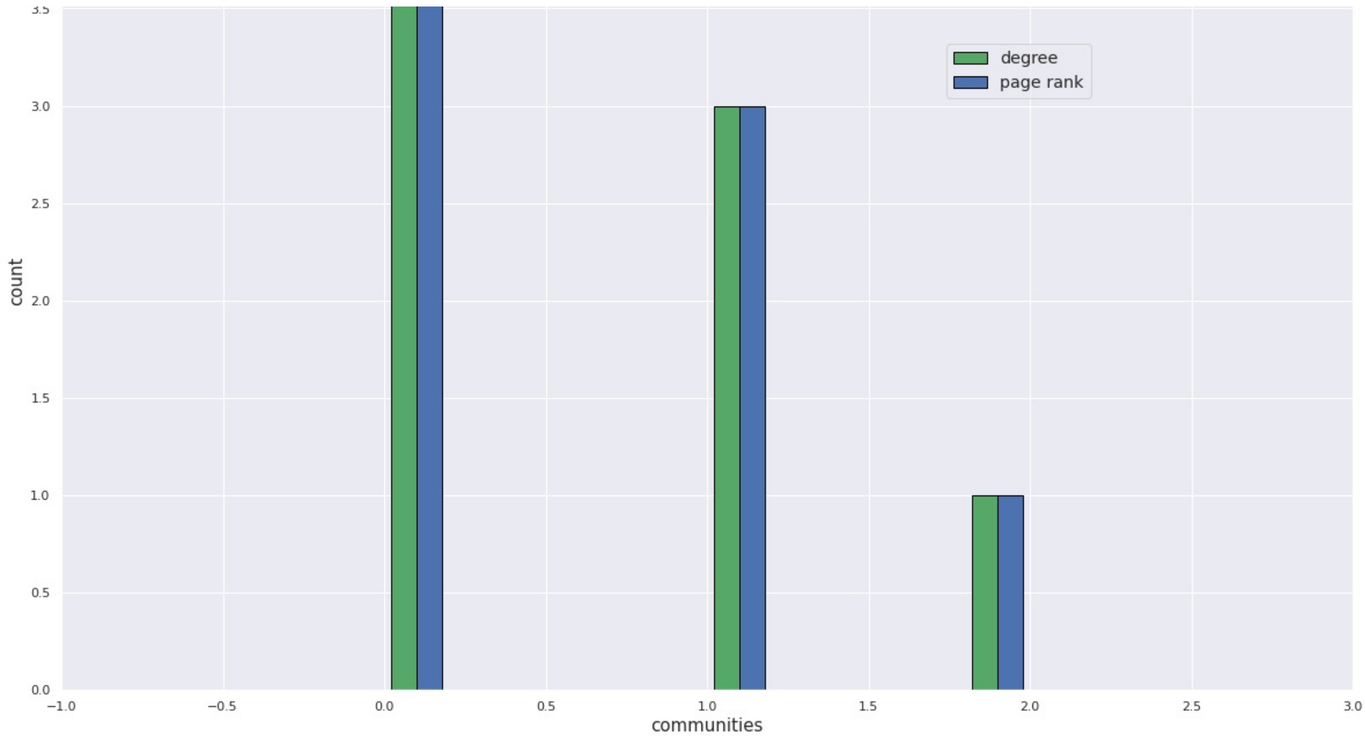
In [193…

```python
### COMPUTE THE 8 MOST CENTRAL NODES ACCORDING TO DEGREE CENTRALITY AND PAGERANK ###

# initialization: pick top 8 nodes from degree centrality and pagerank
cent_deg, cent_pr = [], []
top_nodes_deg = sorted_deg[0:8]
top_nodes_pr = sorted_pr[0:8]

# for loop to extract most central nodes according to our 2 centrality measures
for node in range(8):
    for i in range(len(communities)):
        for j in range(len(communities[i])):
            # extracting 8 most central nodes according to degree centrality
            if top_nodes_deg[node]==communities[i][j]:
                cent_deg.append(i)
            # extracting 8 most central nodes according to page rank
            if top_nodes_pr[node]==communities[i][j]:
                cent_pr.append(i)
```

In [194…

```python
### DISTRIBUTION OF 8 MOST CENTRAL NODES ###

sns.set()

x_dist8 = [cent_deg, cent_pr]

plt.figure(figsize=(20,13))
plt.hist(x_dist8, label=['degree','page rank'], color=['g', 'b'], edgecolor='black')
plt.xlim(-1,len(communities))
plt.xlabel('communities', fontsize=15)
plt.ylabel('count', fontsize=15)
plt.suptitle('Distribution of 8 most central nodes accross the communities', fontsize=30)

plt.legend(loc="upper right", bbox_to_anchor=(0.81,0.81), borderaxespad=1, frameon=True, fontsize='large')

plt.show()
```





Comment

This confirms our second correlation matrix. Page Rank and Degree centrality have the same distribution over the 3 communities for the top 8 most central nodes. Indeed this isn't surprising as we saw our correlated their rankings were in 2.3.1.

2.3.3 Comparing clusterings

Now we use the Adjusted Rand Index (ARI) to compare clusterings. We can define the ARI as follows: $ARI = \frac{\sum_{i,j} {n_{i,j} \choose 2} - \left[\sum_i {a_{i} \choose 2} \sum_j {b_{j} \choose 2} \right] / {n \choose 2}}{ \frac{1}{2}\left[\sum_i {a_{i} \choose 2} \sum_j {b_{j} \choose 2} \right] - \left[\sum_i {a_{i} \choose 2} \sum_j {b_{j} \choose 2} \right] / {n \choose 2}}$

In [195…
```
### IMPORT NECESSARY PACKAGES ###
import pandas as pd
import numpy as np
```

In [196…
```
### LOAD GROUND TRUTH MATRIX ###
GT = pd.read_csv("/content/ground_truth_karate_club.csv")
GT.drop(GT.columns[[0]], axis=1, inplace=True) # delete first column
GT.head()
```

Out[196…

	0
0	Mr. Hi
1	Mr. Hi
2	Mr. Hi
3	Mr. Hi
4	Mr. Hi

In [197…
```
### CONVERT GT TO NUMPY ###
GT = GT.to_numpy()
```

In [198…
```
### CREATE CONSISTENCY TABLE FUNCTION ###

def cons_t(c1, c2):

    N = c1.size # pick length of cluster 1 vector

    # pick vector from cluster 2 with the number of unique entries
    n1 = np.unique(c2)
    N1 = n1.size

    # pick vector from cluster 2 with the number of unique entries
    n2 = np.unique(c1)
    N2 = n2.size
```

```python
    # ininitialize consistency table to fill
    cons_t = np.zeros((N2+1, N1+1))

    for i in range(N): # loop over cluster size
        cons_t[c1[i], c2[i]] += 1 # update

    # define sum of rows and columns
    s_1 = np.sum(cons_t, axis=1) # column sum
    s_2 = np.sum(cons_t, axis=0) # row sum

    for j in range(N2): # loop over number of unique entries
        cons_t[j, N1] = s_1[j] # update

    for k in range(N1): # loop over number of unique entries
        cons_t[N2, k] = s_2[k] # update

    return cons_t
```

In [199…

```python
### CREATE ADJUSTED RAND INDEX FUNCTION ###

def ARI(c1, c2):

    # sums from top of fraction
    s_top_1 = 0
    s_top_2 = 0
    # sums from 2nd term of top of fraction
    s_top_21= 0
    s_top_22= 0

    # sums from bottom of fraction
    s_bottom_1 = 0
    s_bottom_2 = 0

    n = c1.size

    # choose unique values from c2
    n1 = np.unique(c2)
    N1 = n1.size

    # choose unique values from c1
    n2 = np.unique(c1)
    N2 = n2.size

    # create initial CT
    CT = cons_t(c1,c2)

    # double loop for first term of top of fraction
    for k in range(N2):
        for l in range(N1):
            s_top_1 += sc.special.binom(CT[k,l],2)

    # compute first sum of second term of top of fraction
    for k in range(N2):
        s_top_21 += sc.special.binom(CT[k,N1],2)

    # compute second sum of second term of top of fraction
    for l in range(N1):
        s_top_22 += sc.special.binom(CT[N2,l],2)

    # divide 2nd term of top of fraction by the combinatorics
    s_top_2 = s_top_21 * s_top_22 / sc.special.binom(n,2)

    # compute first sum from bottom of frac
    for k in range(N2):
        s_bottom_1 += sc.special.binom(CT[k,N1],2)

    for l in range(N1):
        s_bottom_1 += sc.special.binom(CT[N2,l],2)

    s_bottom_1 /= 2 #result for first sum from bottom of frac

    # compute seond sum from bottom of frac
    s_bottom_2 = s_top_2

    return (s_top_1 - s_top_2)/(s_bottom_1-s_bottom_2)
```

In [200…

```python
# initialization
n = np.arange(len(communities)) # pick range of number of communitities
color_map1 = [0]*34 # choose how to color the graph
labels32 = []

# for loop: labeling each node by its community number
for m in range(100):
  for i in range(len(n)): # loop over number of communities
    for j in range(len(communities[i])): # loop over size of each community
      color_map1[communities[i][j]] = i
      labels32.append(color_map1)
```

```
labels32 = np.array(labels32)
```

In [201…

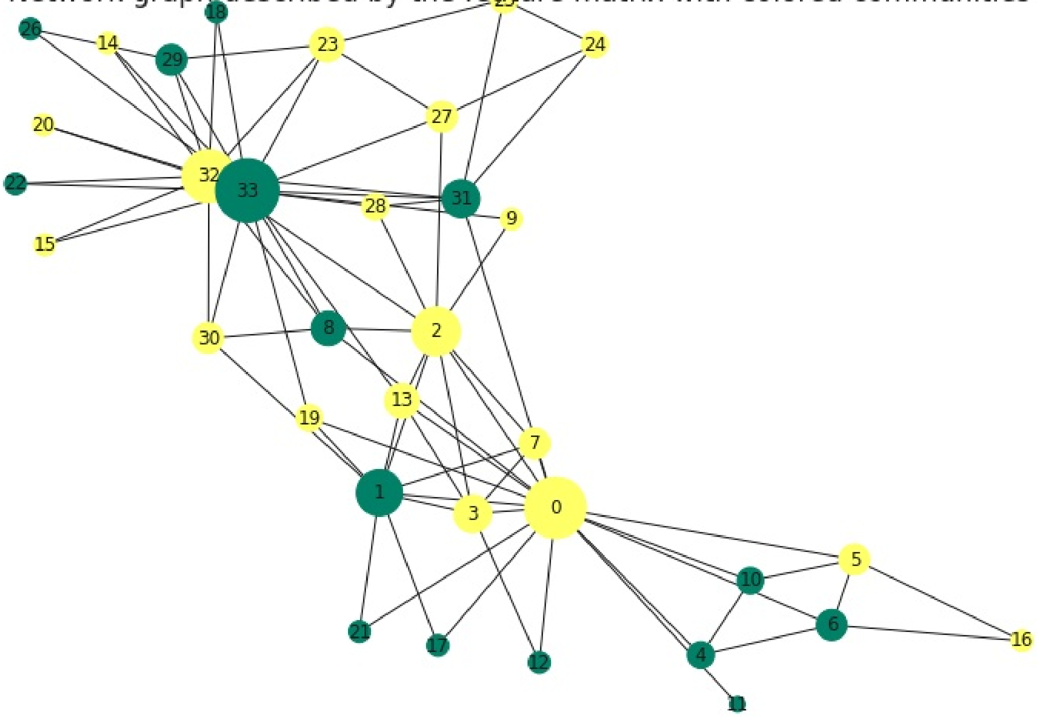
```
### LOOK AT ARI FOR DIFFERENT SOLUTIONS OVER 100 ITERATIONS ###
labels31 = []
labels33 = []
ARI31 = []
ARI32 = []
ARI33 = []
ground_truth = [0 if label=="Mr. Hi" else 1 for label in GT]

for i in range(100):
  centroids31, labels301 = k_means(2, F, 15)
  ground_truth = ground_truth
  labels31.append(np.array(labels301))
  labels33.append(np.array(ground_truth))
  ARI31.append(ARI(np.array(labels31[i-1]), np.array(labels32[i-1])))
  ARI32.append(ARI(np.array(labels31[i-1]), np.array(labels33[i-1])))
  ARI33.append(ARI(np.array(labels32[i-1]), np.array(labels33[i-1])))

ari31 = np.mean(ARI31)
ari32 = np.mean(ARI32)
ari33 = np.mean(ARI33)

print("The ARI between k-means and CNM is:", ari31)
print("The ARI between k-means and GT is:", ari32)
print("The ARI between CNM and GT is:", ari33)
```

```
The ARI between k-means and CNM is: -0.027790378065932687
The ARI between k-means and GT is: 0.01154456267004692
The ARI between CNM and GT is: 0.5684394071490848
```

In [204…

```
### PLOT NETWORK WITH 2 COMMUNITIES (best k from k means) ###

sns.set_style("white")

plt.figure(figsize=(10,7))
nx.draw(G, pos, with_labels=True, cmap='summer', node_size=deg, node_color=two_one)
plt.suptitle('Network graph described by the feature matrix with colored communities', fontsize=18)

plt.show()
```

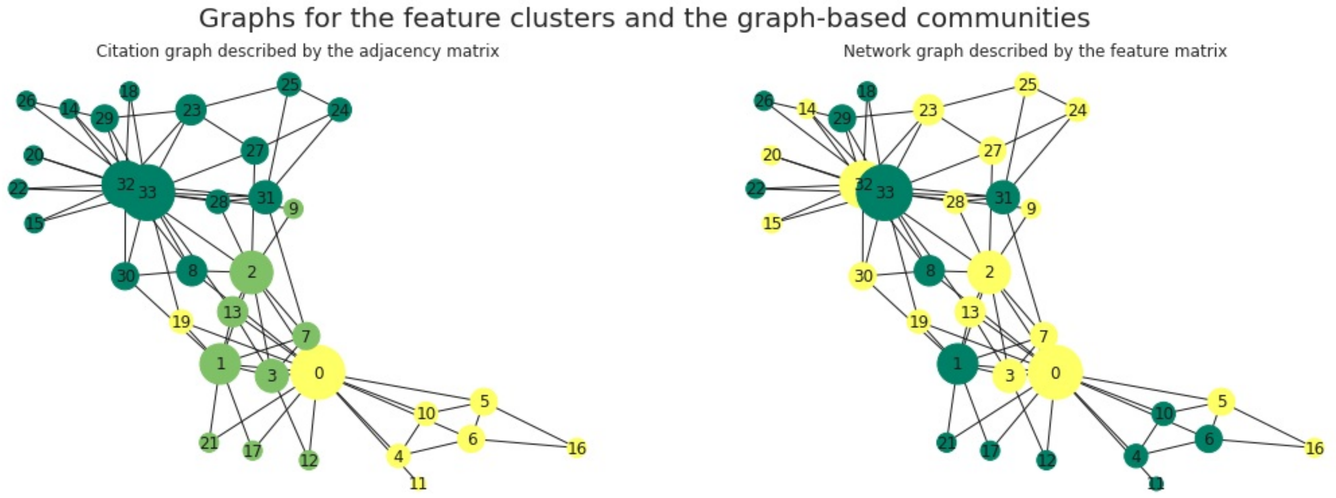
Network graph described by the feature matrix with colored communities



In [205…

```
### COMPARISON OF GRAPHS WITH 2 AND 3 COMMUNITIES ###

sns.set_style("white")

fig = plt.figure(figsize = (18,6))

ax = fig.add_subplot(121)
nx.draw(G, pos, with_labels=True, node_size=deg, cmap='summer', node_color=two_three)
plt.title('Citation graph described by the adjacency matrix')
```

```
ax = fig.add_subplot(122)
nx.draw(G, pos, with_labels=True, node_size=deg, cmap='summer', node_color=two_one)
plt.title('Network graph described by the feature matrix')

plt.suptitle('Graphs for the feature clusters and the graph-based communities', fontsize=20)


plt.show()
```



Comment

The ARI score is the best for CNM method with k=3, although the true number of cluster is actually 2 (from ground truth). Maybe the method is better from graph theory. Indeed when we look at the plotted graph for k=3, we see immediate clustering thanks to the colouring which makes sense, whereas for k=2, there's much more mix in the nodes, which looks less exact. Indeed k-means has much randomness involved, and even if there's two clusters as the best solution when we compare the clusterings from k-means and the ground truth, they don't necessarily have the same split which is why we get a low ARI sometimes. Like in part 2.1.3 where I have looked at ARI pairs over 100 random initialization of k-means and averaged the mean, here we find that the ARI is best for the ground truth and the CNM solution.

Overall we have seen that with k-means we get a best k of 2, and from Clauset-Newman-Moore we get a best k of 3. We are using a different method so it's not too worrying.

References

1. CH score
2. 3-D plot
3. Explained variance and PCA
4. Page Rank
5. ARI score
6. Regularisation
7. L1-L2
8. Cross-Entropy differentiation
9. Rule of thumb for number of parameters in a neural network