Minería de datos con WEKA y Python

Visualización de datos

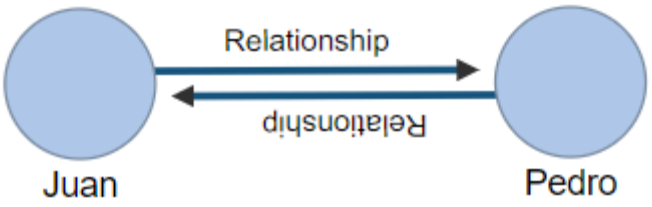
A la hora de obtener una representación visual de las relaciones entre usuarios de una plataforma social, un grafo resulta la opción más indicada.

Debido a su naturaleza basada en NoSQL de grafos, OrientDB proporciona una base sólida para gestionar y visualizar eficientemente grandes redes de grafos, lo que permite representar las relaciones entre usuarios de manera efectiva, flexible y escalable. Para explorar las distintas opciones a la hora de visualizar los datos, es esencial comprender la estructura de los datos a tratar:

La base de datos OrientDB que se tratará contiene datos de usuarios almacenados en nodos, los cuales se representan a través de círculos, y mantienen conexiones entre ellos mediante vectores o aristas, estas aristas definen la relación que mantienen dichos usuarios entre ellos. Estas relaciones se han dividido en dos tipos: *Relationship* o *Block*, las cuales representan el estado de amistades o el deseo de evitar a otros usuarios, respectivamente.

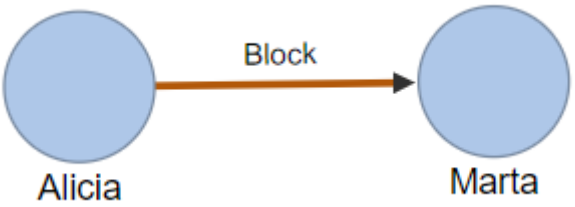
Actualmente, el sistema puede representar varios tipos de relaciones de usuario con estas dos aristas, para ello, se emplea la direccionalidad de los vectores como indicador de la reciprocidad de estas relaciones:

- Las relaciones de **amistad** se representarán mediante vectores *Relationship* bidireccionales, indicando que el aprecio es mutuo entre ambos usuarios.

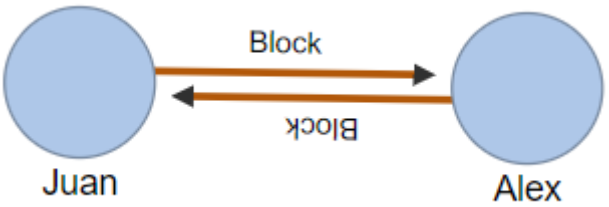


- Para el caso en el que un usuario **solicita ser amigo de otro** y este aún no ha aceptado o negado la petición el vector *Relationship* será unidireccional, indicando que el usuario desde donde se origina el vector es el autor de la petición, y que esta va dirigida al usuario en el destino de la arista.

- Para el caso en el que un usuario **quiera evitar a otro** y, por tanto, opte por **bloquearlo**, se representará esta situación mediante un vector *Block* originado desde el usuario autor del bloqueo y con el usuario a bloquear como destino.

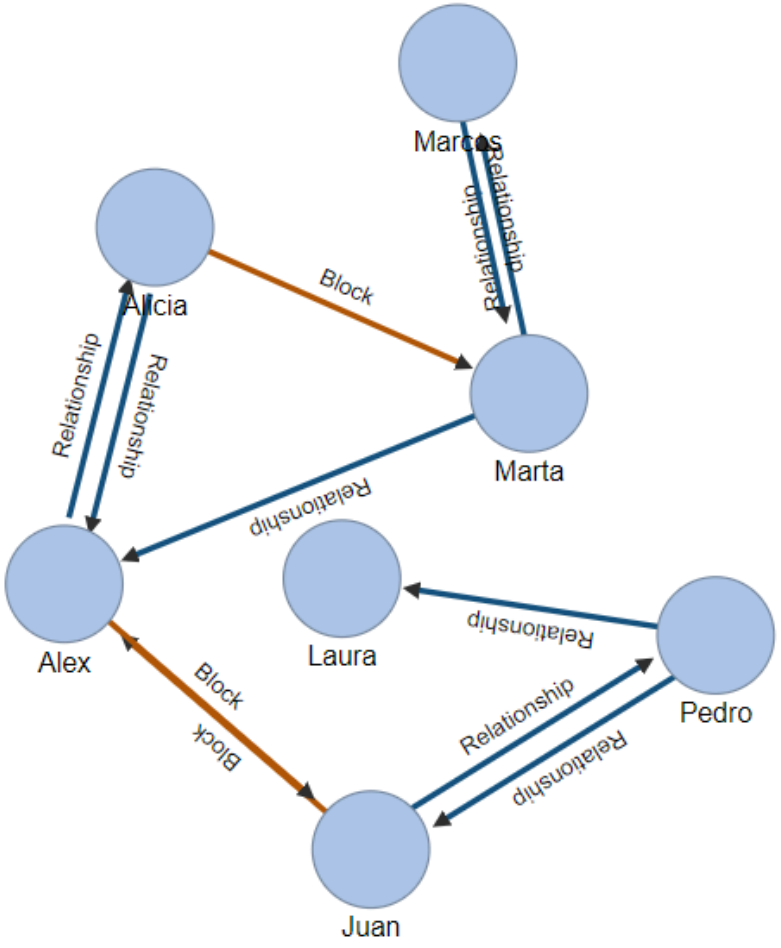


- De forma similar, si dos usuarios desean evitarse mutuamente, simplemente quedará representado como una relación *Block* bidireccional.



- Finalmente, aquellos usuarios que no guarden ningún tipo de relación entre ellos aparecerán sin conexiones entre los nodos.

Empleando estas reglas, es posible establecer redes más complejas de usuarios y comprender las distintas relaciones entre ellos:



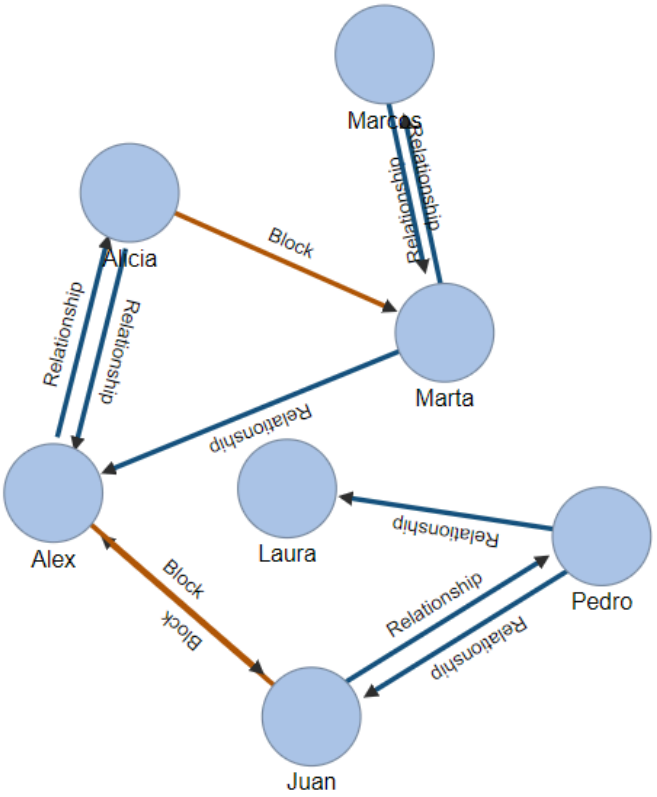
Ya establecida la forma de representación de la información, podemos explorar distintas formas de extraer y representar información de este tipo de redes mediante la representación de grafos. Para extraer la información y graficarla, se empleará el editor visual por defecto de OrientDB: *OrientDB Studio*.

Visualización de una malla completa de usuarios

Para la visualización de una malla completa de usuarios podemos emplear una consulta simple en OrientDB desde el graficador de *OrientDB Studio* tal como:

```
SELECT * FROM User
```

Lo que resulta en una visualización con todos los nodos existentes de la base de datos y sus relaciones completas, tal como se muestra en la siguiente imagen:

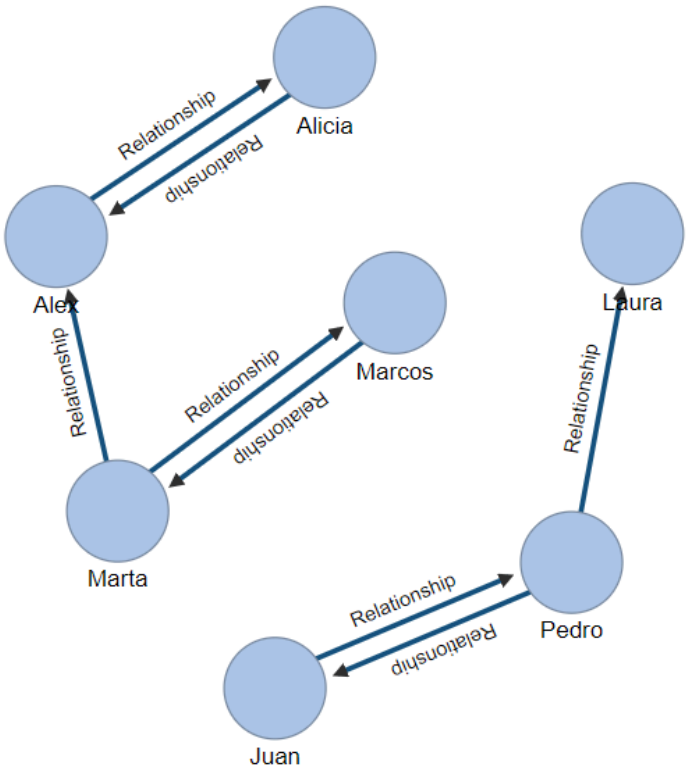


Visualización de las interacciones de amistad

Mediante una secuencia SQL, podemos obtener una visual rápidamente de todas las interacciones amistosas:

```
SELECT FROM Relationship
```

Esto devuelve un gráfico con todas las amistades y solicitudes de amistad alojadas en la base de datos:

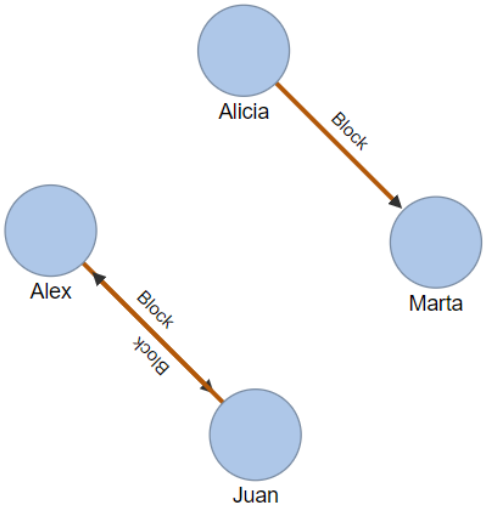


Representación de las interacciones de bloqueo entre usuarios

De forma similar a las amistades, podemos visualizar los bloqueos entre usuarios mediante una consulta simple:

```
SELECT FROM Block
```

Resultando en una representación similar a la siguiente:



En este caso, tan solo existen dos relaciones de bloqueo entre usuarios: *Alicia* tiene bloqueada a *Marta*, mientras que *Alex* y *Juan* se bloquean mutuamente. Esto permite tener una idea general rápidamente de los usuarios que prefieren no tener ningún tipo de relación.

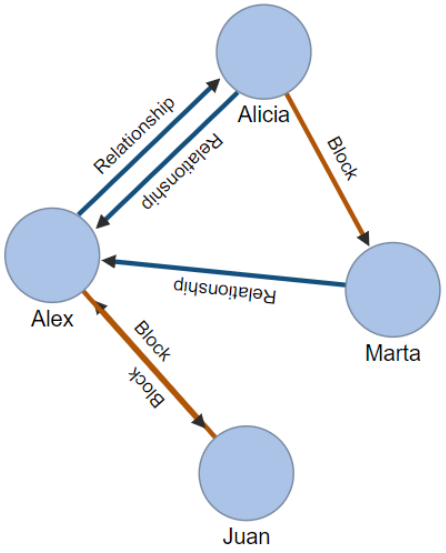
Visualización de círculos de amistades

La visualización en grafo de las relaciones permite también analizar a usuarios concretos de forma más detallada, permitiendo aislar grupos de amistades y relaciones.

Tomemos al usuario *'Alex'* como ejemplo, podemos ilustrar su círculo de interacciones completo mediante la consulta:

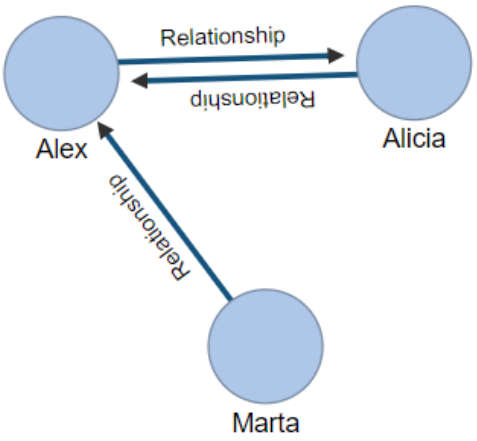
```
SELECT FROM User
WHERE @rid IN (
       SELECT expand(both()) FROM User
       WHERE name = 'Alex'
) OR name='Alex'
```

Lo que nos muestra el círculo de relaciones de *Alex*:



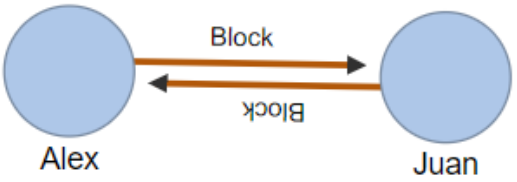
De forma similar, podemos obtener únicamente las interacciones amistosas de *Alex* empleando la consulta:

```
SELECT FROM Relationship
       WHERE in.name ='Alex'
       OR out.name = 'Alex'
```



Similarmente, podemos representar los bloqueos con la misma consulta:

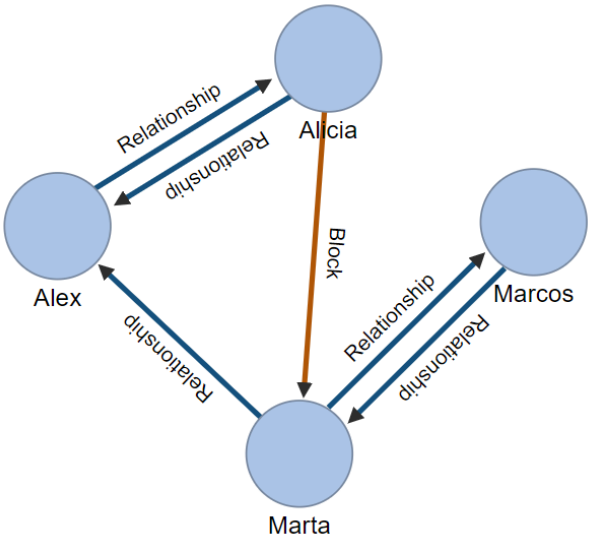
```
SELECT FROM Block
       WHERE in.name ='Alex'
       OR out.name = 'Alex'
```



Dibujo de círculos de amistad ampliados

Finalmente, podemos trazar lo que podría ser una forma básica de crear recomendaciones de amistad para usuarios, mediante la extracción de los *"amigos de mis amigos"*. Mediante esta representación se puede visualizar las potenciales amistades de un usuario.

```
SELECT EXPAND(
  bothE('Relationship').bothV()
  .bothE('Relationship').bothV()
) FROM User WHERE name='Alex'
```



En este caso, podemos ver que *Marcos* es el único *amigo de los amigos* de *Alex*.

Algoritmos aplicables

El problema que se trata de abordar en el siguiente apartado es la creación de un sistema de recomendación de amistades centrado en el usuario y que le permita conocer gente con la cual le sería interesante relacionarse.

Como requisitos del sistema, se deberá tener en cuenta que los únicos datos que tendremos en cuenta, a priori, son las relaciones interpersonales. Únicamente funcionaremos con las relaciones de quien es amigo de quien. No se dispondrán datos sobre interacciones, visualizaciones o temas de interés para el usuario.

Para dicho problema, se nos presentan 2 posibles soluciones; una clásica basada en algoritmos escritos directamente por el desarrollador, y otra más dinámica el aprendizaje máquina.

Solución clásica: El amigo de mi amigo es mi amigo.

Una forma sencilla de abordar el problema planteado sería recomendar como amistades a los amigos de tus amigos, siendo los primeros en esta lista en aparecer los que más amigos en común se tengan y, para lo demás, ordenando de forma aleatoria.

Esta solución ofrecerá siempre una misma solución dado un estado en la base de datos, lo que haría que la tasa de descubrimiento de personas de un usuario fuese relativamente baja, ya que solo le aparecerán personas distintas en caso de que las relaciones entre sus usuarios relacionados cambian.

En el mundo del “engagement” y la recolección de datos, este sistema sería rápido, pero insuficiente de cara a ofrecer una mayor variabilidad en cuanto las recomendaciones.

Solución avanzada: Graph Convolutional Networks (GCN)

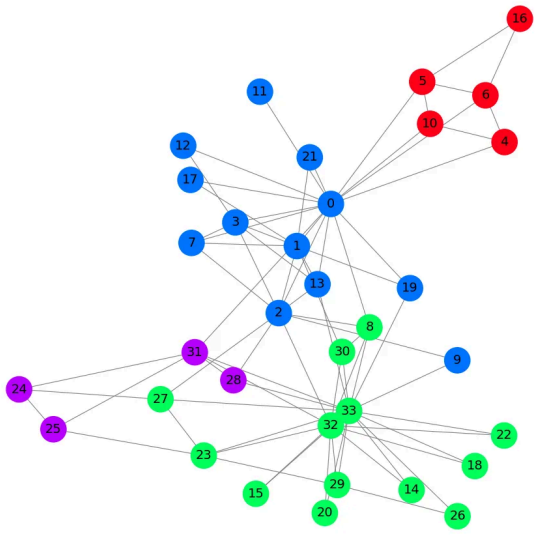
Para ayudar a flexibilizar la sugerencia de amistades, se propone hacer uso de un algoritmo de Machine Learning capaz de dividir el grafo en un subconjunto de grafos semi-separados, permitiendo dar unas recomendaciones de amistad basadas

en la sub-división en grandes grupos de la población de la red. Esto permitirá generar un subconjunto mayor de posibles amistades, las cuales no tienen porqué tener una relación directa con las amistades del usuario objetivo al mismo tiempo que siguen siendo usuarios “cercanos” entre sí.

¿Que es una Red Convolucional de Grafos (GNC)?

> *A graph neural network (GNN) belongs to a class of artificial neural networks for processing data that can be represented as graphs*
>
> *Graph Neural Network - Wikipedia*

A grandes rasgos, una GCN es un algoritmo el cual permite, dado un grafo no espacial con aristas no ponderadas, dividirlo en subgrupos cuyos integrantes están separados o semi-separados del resto.



Este tipo de algoritmo permite la división de la red en subredes de elementos altamente cohesionados entre sí, permitiendo hacer una separación de la red sin tener en cuenta la distancia entre usuarios, sino subgrupos más cerrados de usuarios.

El objetivo de esta clasificación es la de ampliar el grupo de usuarios que podrían ser seleccionados como recomendación, pudiendo hacer una selección aleatoria entre usuarios del mismo grupo con los cuales no exista relación actualmente.

Aplicando una GNC

Para la aplicación de la GNC, se hará uso de un cuaderno Jupyter junto con las librerias pytorch y torch_geometric.

```
%pip install pytorch torch_geometric
```

Una vez instaladas las librerias necesarias, se importarán para su posterior uso.

```
import networkx as nx
import matplotlib.pyplot as plt
```

Importaremos a su vez el dataset “KarateClub”, el cual es un grafo de relaciones entre usuarios, con aristas no ponderadas y sin ubicación espacial de los usuarios.

```
from torch_geometric.datasets import KarateClub
```

Una vez hechas todas las importaciones, pasaremos a mostrar los datos del dataset para hacernos una idea de su forma.

```
# Import dataset from PyTorch Geometric
dataset = KarateClub()
# Print information
print(dataset)
print('------------')
print(f'Number of graphs: {len(dataset)}')
print(f'Number of features: {dataset.num_features}')
print(f'Number of classes: {dataset.num_classes}')
```

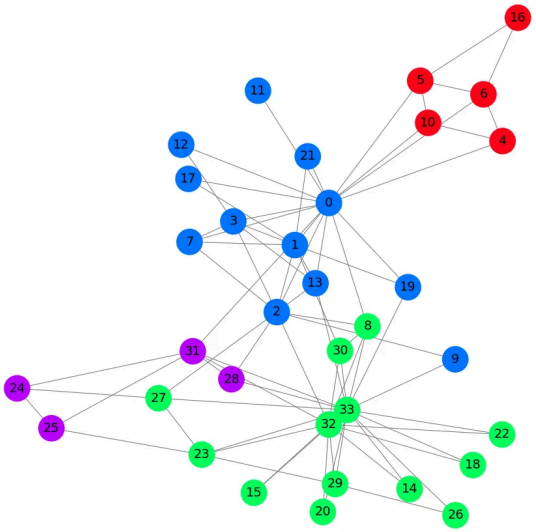
Salida:

```
KarateClub()
------------
Number of graphs: 1
Number of features: 34
Number of classes: 4
```

A continuación mostraremos la solución final dada a el presente problema con los datos requeridos:

```
data = dataset[0]
from torch_geometric.utils import to_networkx
G = to_networkx(data, to_undirected=True)
plt.figure(figsize=(12,12))
plt.axis('off')
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=0),
                 with_labels=True,
```

```
                node_size=800,
                node_color=data.y,
                cmap="hsv",
                vmin=-2,
                vmax=3,
                width=0.8,
                edge_color="grey",
                font_size=14
                )
plt.show()
```



Implementando una GCN

En primer lugar, se declara el modelo a utilizar, utilizando parámetros que podremos modificar para afinar el modelo. En dicho modelo se utiliza una capa oculta de 3 dimensiones, la cual es la GCN. En caso de que se añadiese una segunda capa, nuestro modelo sería capaz de tener en cuenta no solo a los vecinos de un nodo, sino a los vecinos de sus vecinos, pudiendo ampliar la red.

```
from torch.nn import Linear
from torch_geometric.nn import GCNConv

class GCN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.gcn = GCNConv(dataset.num_features, 3)
        self.out = Linear(3, dataset.num_classes)

    def forward(self, x, edge_index):
        h = self.gcn(x, edge_index).relu()
        z = self.out(h)
        return h, z

model = GCN()
```

```
print(model)
```

A continuación se inicia el entrenamiento de la red. En este caso, el criterio de cálculo de la red es la pérdida total de entropía y el optimizador es un optimizador Adam. Para su posterior análisis, también se guardarán todos los datos asociados a cada iteración en un vector. Finalmente, cada 10 iteraciones se mostrará su pérdida y eficacia.

```
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.02)

# Calculate accuracy
def accuracy(pred_y, y):
    return (pred_y == y).sum() / len(y)
# Data for animations
embeddings = []
losses = []
accuracies = []
outputs = []
# Training loop
for epoch in range(201):
    # Clear gradients
    optimizer.zero_grad()
    # Forward pass
    h, z = model(data.x, data.edge_index)
    # Calculate loss function
    loss = criterion(z, data.y)
    # Calculate accuracy
    acc = accuracy(z.argmax(dim=1), data.y)
    # Compute gradients
    loss.backward()
    # Tune parameters
    optimizer.step()
    # Store data for animations
    embeddings.append(h)
    losses.append(loss)
    accuracies.append(acc)
    outputs.append(z.argmax(dim=1))
    # Print metrics every 10 epochs
    if epoch % 10 == 0:
        print(f'Epoch {epoch:>3} | Loss: {loss:.2f} | Acc: {acc*100:.2f}%')
```

Salida:

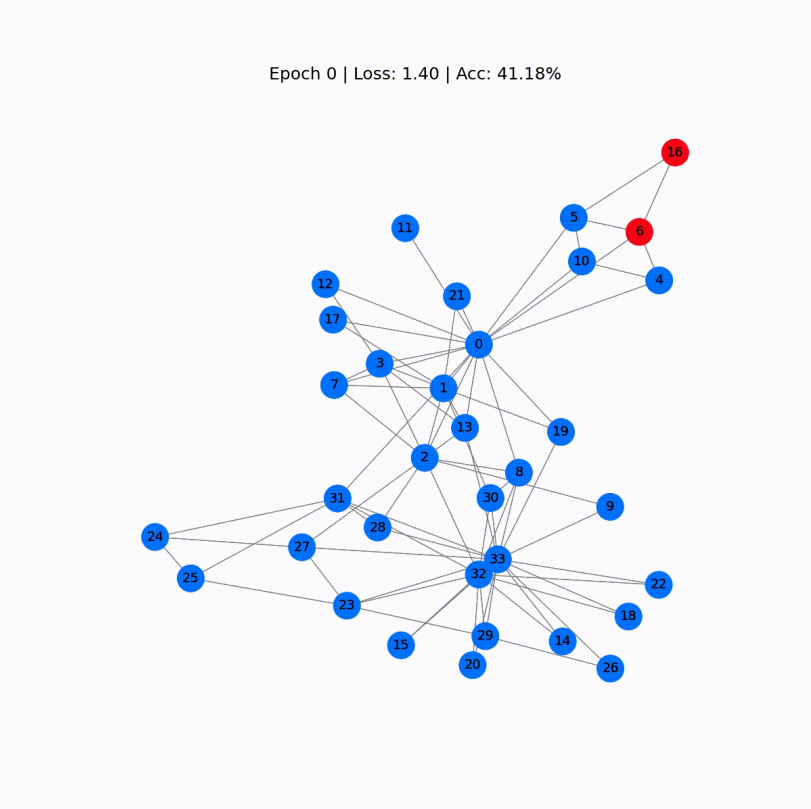
```
Epoch   0 | Loss: 1.58 | Acc: 11.76%
Epoch  10 | Loss: 1.39 | Acc: 17.65%
Epoch  20 | Loss: 1.15 | Acc: 55.88%
Epoch  30 | Loss: 0.87 | Acc: 79.41%
Epoch  40 | Loss: 0.63 | Acc: 79.41%
Epoch  50 | Loss: 0.45 | Acc: 85.29%
Epoch  60 | Loss: 0.31 | Acc: 88.24%
Epoch  70 | Loss: 0.20 | Acc: 94.12%
Epoch  80 | Loss: 0.13 | Acc: 97.06%
Epoch  90 | Loss: 0.08 | Acc: 100.00%
Epoch 100 | Loss: 0.05 | Acc: 100.00%
Epoch 110 | Loss: 0.04 | Acc: 100.00%
Epoch 120 | Loss: 0.03 | Acc: 100.00%
```

```
Epoch 130 | Loss: 0.02 | Acc: 100.00%
Epoch 140 | Loss: 0.02 | Acc: 100.00%
Epoch 150 | Loss: 0.01 | Acc: 100.00%
Epoch 160 | Loss: 0.01 | Acc: 100.00%
Epoch 170 | Loss: 0.01 | Acc: 100.00%
Epoch 180 | Loss: 0.01 | Acc: 100.00%
Epoch 190 | Loss: 0.01 | Acc: 100.00%
Epoch 200 | Loss: 0.01 | Acc: 100.00%
```

Adicionalmente y de forma meramente informativa, puede configurarse de tal manera que se vaya mostrando cada paso por cada una de las iteraciones que se hacen sobre los datos, mostrando el paso a paso de la conclusión final de red a la que se ha llegado.

```
%%capture
from IPython.display import HTML
from matplotlib import animation
plt.rcParams["animation.bitrate"] = 3000

def animate(i):
    G = to_networkx(data, to_undirected=True)
    nx.draw_networkx(G,
                     pos=nx.spring_layout(G, seed=0),
                     with_labels=True,
                     node_size=800,
                     node_color=outputs[i],
                     cmap="hsv",
                     vmin=-2,
                     vmax=3,
                     width=0.8,
                     edge_color="grey",
                     font_size=14
                     )
    plt.title(f'Epoch {i} | Loss: {losses[i]:.2f} | Acc: {accuracies[i]*100:.2f}%',
              fontsize=18, pad=20)
fig = plt.figure(figsize=(12, 12))
plt.axis('off')
anim = animation.FuncAnimation(fig, animate, \
            np.arange(0, 200, 10), interval=500, repeat=True)
html = HTML(anim.to_html5_video())
display(html)
```


Epoch 0 | Loss: 1.40 | Acc: 41.18%
16
5
6
11
10
4
12
21
17
0
3
7
1
13
19
2
8
31
30
9
28
24
27
33
25
32
22
23
18
29
14
15
20
26