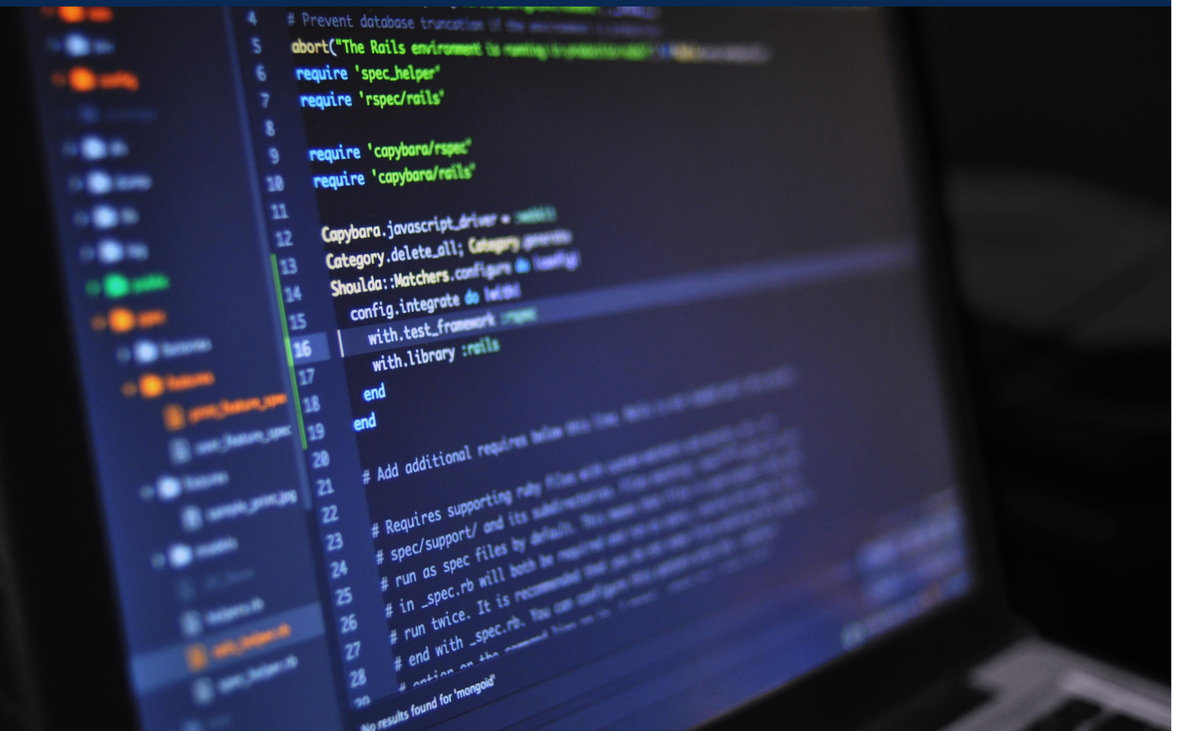
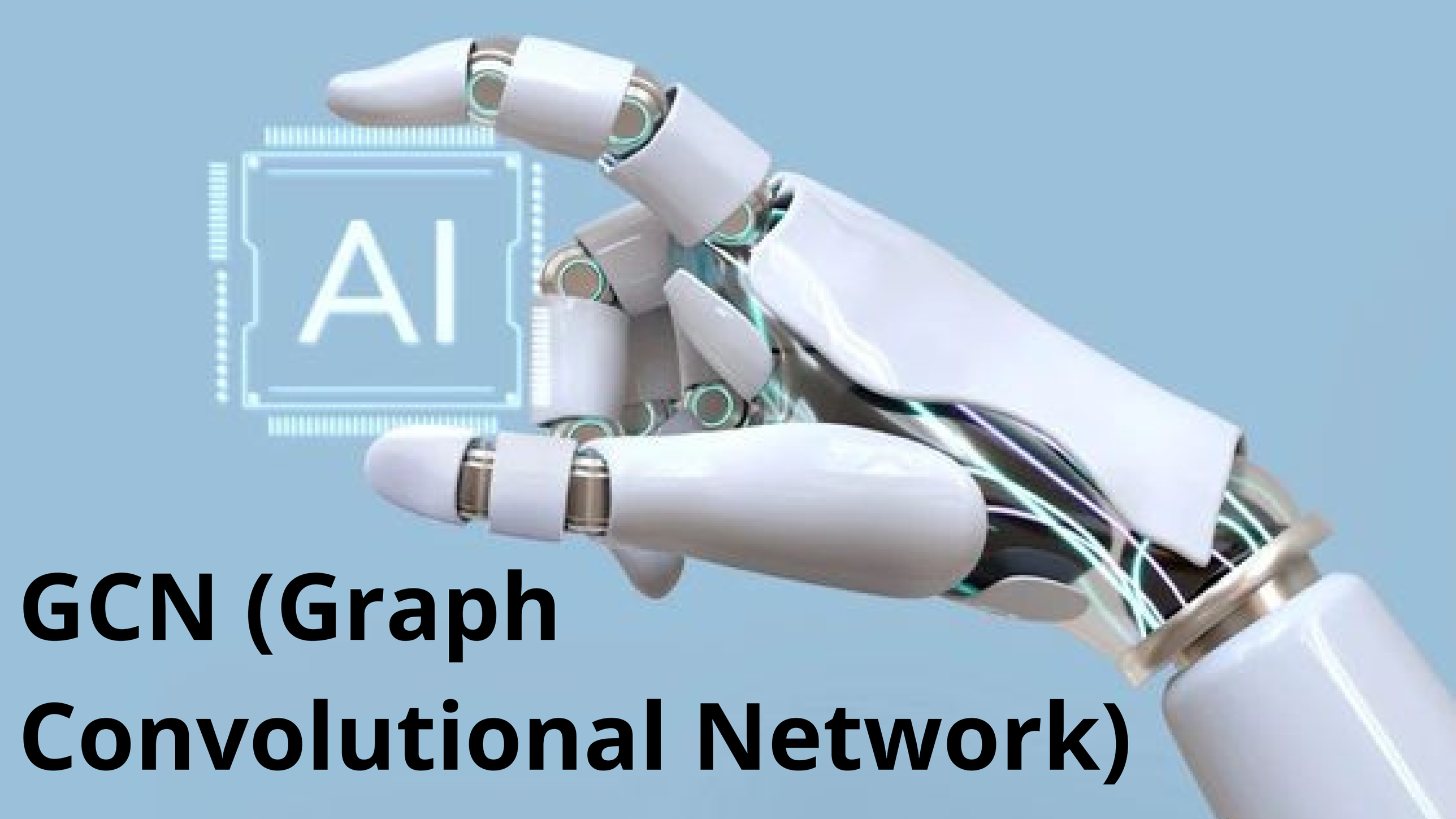
APRENDIZAJE ARTIFICIAL RELACIONAL



1. INTRODUCCION
2. GCN (GRAPH CONVOLUTIONAL NETWORK)
3. GAT (GRAPH ATTENTION NETWORK)
4. NAIVE BAYES
5. KNN: (K-NEAREST NEIGHBORS)
6. METODOLOGIA
7. RESULTADOS

PROYECTO PRACTICO



GCN (Graph Convolutional Network)

In [2]:
```python
# Definir la clase del modelo GCN
class GCN(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(GCN, self).__init__()
        self.gc1 = GraphConvolution(input_dim, hidden_dim)
        self.gc2 = GraphConvolution(hidden_dim, output_dim)

    def forward(self, x, adj):
        x = F.relu(self.gc1(x, adj))
        x = self.gc2(x, adj)
        return F.log_softmax(x, dim=1)
```

In [3]:
```python
# Definir la clase para la capa de convolución de grafos
class GraphConvolution(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(GraphConvolution, self).__init__()
        self.weight = nn.Parameter(torch.FloatTensor(input_dim, output_dim))
        self.bias = nn.Parameter(torch.FloatTensor(output_dim))

    def forward(self, x, adj):
        x = torch.matmul(x, self.weight)
        ##print(x.shape)       para comprobar las dimensiones de x
        ##                      y ver que se puede mult con adj
        x = torch.matmul(adj, x)
        x = x + self.bias
        return x
```

In [ ]:

```
# Preparar los datos en tensores de PyTorch
features = torch.FloatTensor(features)
labels = torch.LongTensor(labels)
adj = nx.adjacency_matrix(graph)
print(adj.shape)
adj = torch.FloatTensor(adj.todense())

train_features = features[train_indices]
train_labels = labels[train_indices]
train_adj = adj[train_indices, :][:, train_indices]

val_features = features[val_indices]
val_labels = labels[val_indices]
val_adj = adj[val_indices, :][:, val_indices]

test_features = features[test_indices]
test_labels = labels[test_indices]
test_adj = adj[test_indices, :][:, test_indices]
```

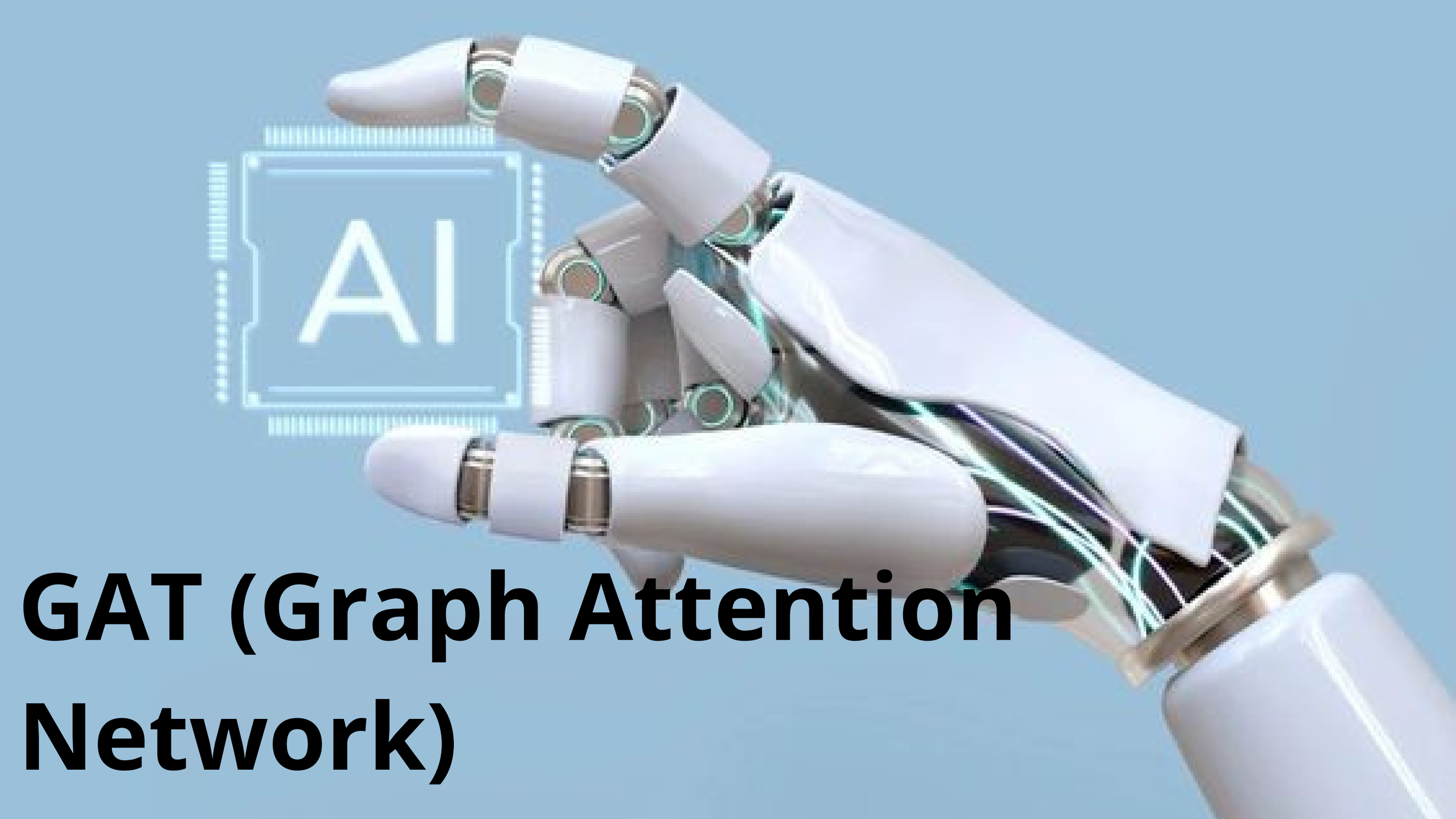
In [ ]:
```python
# Crear el modelo GCN
model = GCN(input_dim, hidden_dim, output_dim)
```

In [ ]:
```python
# Definir la función de pérdida y el optimizador
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=lr)
```

In [ ]:
```python
# Función de entrenamiento
def train(model, features, adj, labels, train_indices, epochs, lr):
    optimizer = optim.Adam(model.parameters(), lr=lr)

    model.train()
    for epoch in range(epochs):
        optimizer.zero_grad()
        output = model(features, adj)
        loss = F.cross_entropy(output[train_indices], labels[train_indices])
        loss.backward()
        optimizer.step()

        print('Epoch: {:04d} | Loss: {:.4f}'.format(epoch+1, loss.item()))
```

In [ ]:
```python
train(model, features, adj, labels, train_indices, epochs, lr)
```

In [ ]:
```python
## Función de predicción
def predict(model, features, adj, test_indices):
    model.eval()
    with torch.no_grad():
        output = model(features, adj)
        predictions = output[test_indices].argmax(dim=1)
    return predictions
```



GAT (Graph Attention Network)

[ ]:
```python
class GAT(torch.nn.Module):
        def __init__(self, n_features, n_classes, n_heads, dropout, alpha):
            super(GAT, self).__init__()
            self.dropout = dropout
            self.attentions = []
            self.out_att = GraphAttentionLayer(n_features, n_classes, dropout=dropout, alpha=alpha, concat=True)

            for _ in range(n_heads-1):
                self.attentions.append(GraphAttentionLayer(n_features, n_classes, dropout=dropout, alpha=alpha, concat=True))

        def forward(self, x, adj):
            x = F.dropout(x, self.dropout, training=self.training)
            x = torch.cat([att(x, adj) for att in self.attentions], dim=1)
            x = F.dropout(x, self.dropout, training=self.training)
            print(x.shape)
            print(adj)
            x = F.elu(self.out_att(x, adj))
            return x
```

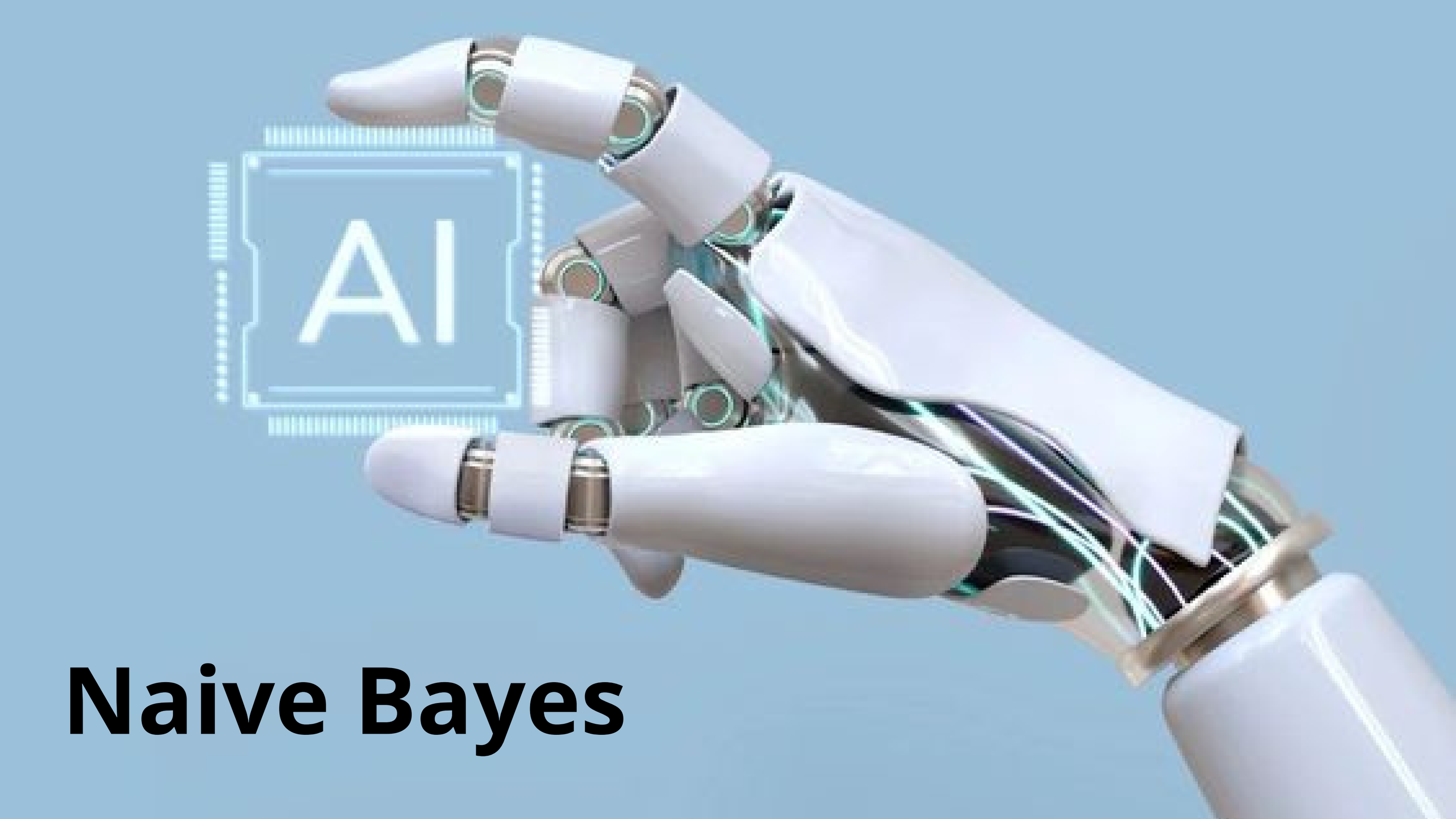
[ ]:
```python
model = GAT(n_features=n_features, n_classes=n_classes, n_heads=n_heads, dropout=dropout, alpha=alpha)
```

```python
# Definir la función de pérdida y el optimizador
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=weight_decay)
```

```python
# Entrenamiento del modelo
model.train()
for epoch in range(num_epochs):
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data, adj)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

    # Validación durante el entrenamiento
    model.eval()
    with torch.no_grad():
        val_loss = 0
        correct = 0
        total = 0
        for data, target in val_loader:
            output = model(data, adj)
            val_loss += criterion(output, target).item()
            _, predicted = output.max(1)
            total += target.size(0)
            correct += predicted.eq(target).sum().item()

        val_loss /= len(val_loader)
        val_accuracy = 100 * correct / total

        print('Epoch: {}, Val Loss: {:.4f}, Val Accuracy: {:.2f}%'.format(epoch+1, val_loss, val_accuracy))

    model.train()
```

In [ ]:
```python
def predict(model, features, adj):
    model.eval()
    with torch.no_grad():
        output = model(features, adj)
        probabilities = F.softmax(output, dim=1)
        _, predicted_labels = torch.max(probabilities, dim=1)
    return predicted_labels
```

In [ ]:
```python
predict(model, test_dataset, adj)
```



Naive Bayes

In [ ]:
```
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score
```

In [ ]:
```
##Crear una instancia del clasificador Naive Bayes
model = GaussianNB()
```

In [ ]:
```
##Ajustar el modelo utilizando los datos de entrenamiento
model.fit(features[train_indices], labels[train_indices])
```

In [ ]:
```
##Realizar predicciones utilizando los datos de prueba
predictions = model.predict(features[test_indices])
```

In [ ]:
```
##Calcular la precisión del modelo
accuracy = accuracy_score(labels[test_indices], predictions)
```

kNN:
(k-Nearest Neighbors)

In [ ]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

In [ ]:
```
X_train, X_test, y_train, y_test = train_test_split(atributos_prueba, objetivo_prueva, test_size=0.2, random_state=42)
```

In [ ]:
```
knn = KNeighborsClassifier(n_neighbors=3)  # Puedes ajustar el valor de K según tus necesidades
knn.fit(X_train, y_train)
```

In [ ]:
```
y_pred = knn.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
```

In [ ]:

METODOLOGÍA

3/ METODOLOGÍA

LECTURA->

```
In [4]: # Cargar los datos de vértices y aristas desde los archivos CSV
        vertices_data = pd.read_csv('trabajo raiz/large_twitch_features.csv')[:100]
        aristas_data = pd.read_csv('trabajo raiz/large_twitch_edges.csv')[:99]
        print(vertices_data)
        print(aristas_data)
```

CODIFICACION->

```
]: # Obtener las características y etiquetas de los vértices
   labels = vertices_data['affiliate'].values
   print(labels.size)
```

```
]: ##Codificacion de 'created_at', 'updated_at' y 'language'
   from sklearn.preprocessing import OneHotEncoder

   # Seleccionar las columnas de características categóricas
   categorical_columns = ['created_at', 'updated_at', 'language']

   # Codificar las columnas categóricas utilizando one-hot encoding
   encoder = OneHotEncoder(sparse=False)
   encoded_features = encoder.fit_transform(vertices_data[categorical_columns])

   # Obtener las columnas de características numéricas
   numeric_columns = ['views', 'mature', 'life_time', 'numeric_id', 'dead_account']

   # Combinar las características numéricas y codificadas en un solo array
   features = np.concatenate([vertices_data[numeric_columns].values, encoded_features], axis=1)
```

DIVISION->

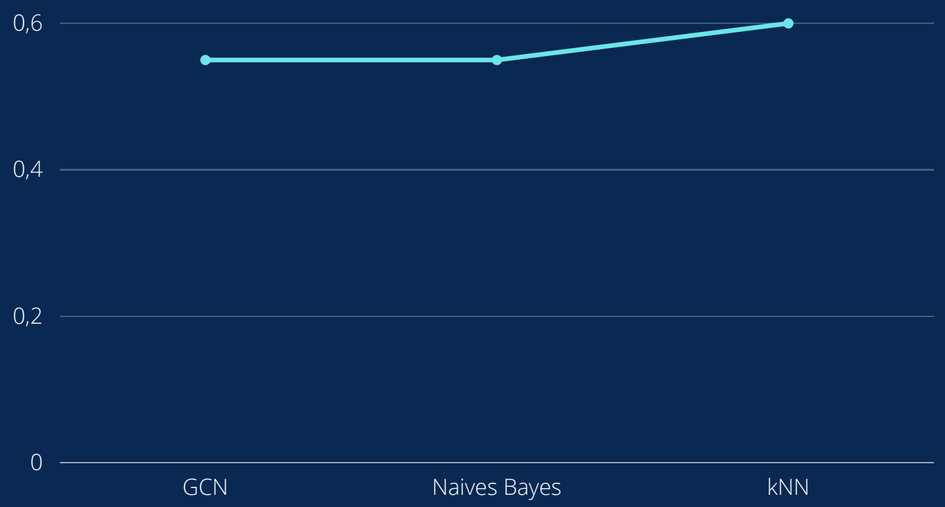
```
# Dividir los datos en conjuntos de entrenamiento, validación y prueba
train_indices, test_indices = train_test_split(range(len(features)), test_size=0.2, random_state=42)
train_indices, val_indices = train_test_split(train_indices, test_size=0.2, random_state=42)
```

CREACION DEL GRAFO->

```
In [ ]: # Construir el grafo a partir de los datos de vértices y aristas
        graph = nx.from_pandas_edgelist(aristas_data, 'numeric_id_1', 'numeric_id_2', create_using=nx.Graph())
        print(graph)
```

RESULTADOS

RESULTADOS




```
La predicción para el modelo por el método GCN será:
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
El accuracy conseguido mediante este método será: 0.55
```

```
La predicción para el método Naive Bayes será:
[1 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0]
El accuracy para el método Naive Bayes será: 0.55
```

```
Las predicciones mediante el modelo kNN con k = 3 son:
[1 0 1 1 1 1 1 0 0 1 0 0 1 0 0 0 0 0 1 0]
El accuracy mediante el modelo kNN con k = 3 es: 0.6
```

MUCHAS GRACIAS
POR SU ATENCION

GRUPO: TRABAJO JUNIO: RELACIONAL 11
CURSO 2022-2023