GraphML and LLMs

The combination of Graph Machine Learning (GraphML) and Large Language Models (LLMs) is like stepping into a new world full of exciting opportunities!

https://github.com/PacktPublishing/Graph-Machine-Learning/tree/main/Chapter12

If you're curious about running an LLM on your own computer, using LM Studio to deploy it—check out the docs at https://lmstudio.ai/docs/api/server for more details!

When we need to tweak or train something, we'll be using the transformer Python module (https://pypi.org/project/transformers/). It's a great tool that lets us quickly grab, use, and even fine-tune models that are already trained, like LLMs.

Graph-Augmented Question Answering: Large language models (LLMs) can use knowledge graphs to answer questions about specific areas with real facts.

Node Embedding Generation: Cutting-edge tools like GraphGPT leverage large language models (LLMs) to create node embeddings right from text, making it easy to connect these embeddings with graph structures.

https://graphgpt.github.io

Building and improving knowledge graphs: Recent examples highlight how LLMs can boost knowledge graphs by pulling out the meaning behind words and identifying key points, which then helps fill in the gaps in the existing data.

LLMs as predictors

LLM operates as a tool to infer outcomes directly from graph data.

"Alice is linked with Bob"

```
import networkx as nx
# Create a directed graph
G = nx.DiGraph()
G.add_node(1, name="Alice", description="she is a software
engineer and she likes reading.")
G.add_node(2, name="Bob", description="he is a data scientist and
he likes writing books.")
G.add_node(3, name="Carl", description="he is a data scientist and
he likes swimming.")
G.add_edge(1, 3, relationship="is friend with")
```

```
# Function to convert network to text
def graph_to_text(graph, edge_type):
    descriptions = []
    # 1. describe the graph structure
    descriptions.append(f"Num nodes:
{graph.number_of_nodes()}.\n")
    for n in graph:
        descriptions.append(f"Node {n}: {graph.nodes[n]
['name']}\n")
    for u, v, data in graph.edges(data=True):
        node_u = graph.nodes[u]
        node_v = graph.nodes[v]
        descriptions.append(f"The person named '{node_u['name']}'
({node_u['description']}) {edge_type} '{node_v['name']}'
({node_v['description']}).")
    return " ".join(descriptions)
text_input = graph_to_text(G)
print("Social Network as text:\n", text_input)
```

```
# Create a prompt
prompt = f"Here is a social network: {text_input}\nBased on the
above, suggest any missing link and explain why they might be
relevant."
```

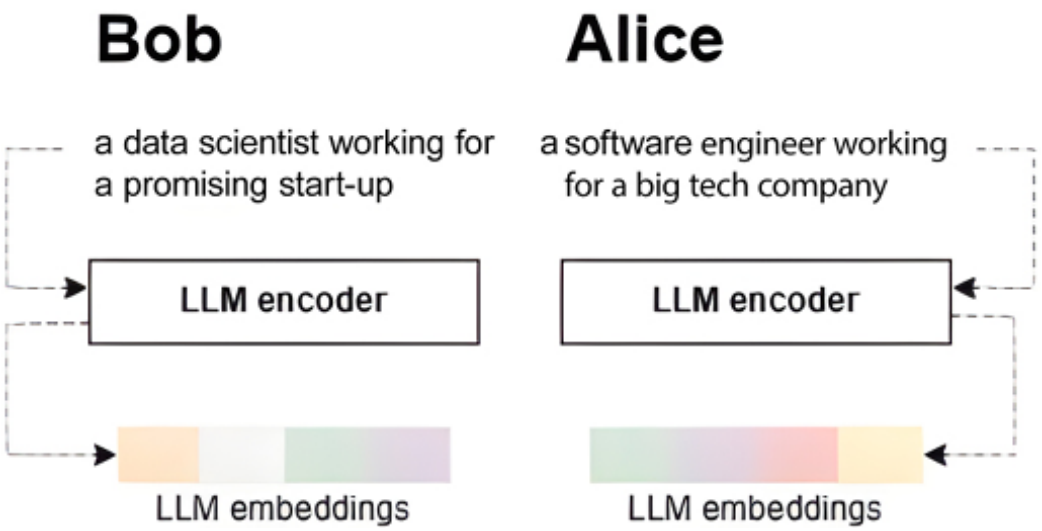
```
# Call the llm to generate a response
from openai import OpenAI
# Create a client for interacting with the LLM server. Here, we
are
# running LM Studio locally, therefore we use the localhost
address and
# "lm-studio" as api key. You can replace this line with a proper
api key
# to a remote LLM service if you have one.
client = OpenAI(base_url="http://localhost:1234/v1/", api_key="lm-
studio")
```

```
response = client.chat.completions.create(
    model="minicpm-llama3-v-2_5",
    messages=[{"role": "system", "content": "You are a helpful
assistant."},
              {"role": "user", "content": prompt}],
    max_tokens=300,
)
```

```
# Extract the generated text from the response
print(response.choices[0].message.content)
```

LLMs as encoders

When graphs are enriched with textual attributes, the LLM as encoder approach becomes particularly powerful.



It is worth noticing that the process of using LLMs as encoders typically involves fine-tuning the LLM on domain-specific textual data to ensure that the embeddings accurately reflect the requirements of the task.

```
import networkx as nx
from openai import OpenAI
# Let's create a toy movie graph
G = nx.Graph()
G.add_node(1, title="Inception", description="A mind-bending
thriller about dreams within dreams.")
G.add_node(2, title="The Matrix", description="A hacker discovers
the shocking truth about reality.")
G.add_node(3, title="Interstellar", description="A team travels
through a wormhole to save humanity.")
G.add_edge(1, 2, similarity=0.8)
G.add_edge(1, 3, similarity=0.9)
```

```
# Intialize the client
client = OpenAI(base_url="http://localhost:1234/v1/", api_key="lm-
studio")
```

```
def encode_text(text):
    # Prepare the query for the LLM
    response = client.embeddings.create(
        input=text,
        model="text-embedding-nomic-embed-text-v1.5-embedding"
    )
    # Get 768-dimensional embedding
    embedding = response.data[0].embedding
    return embedding
```

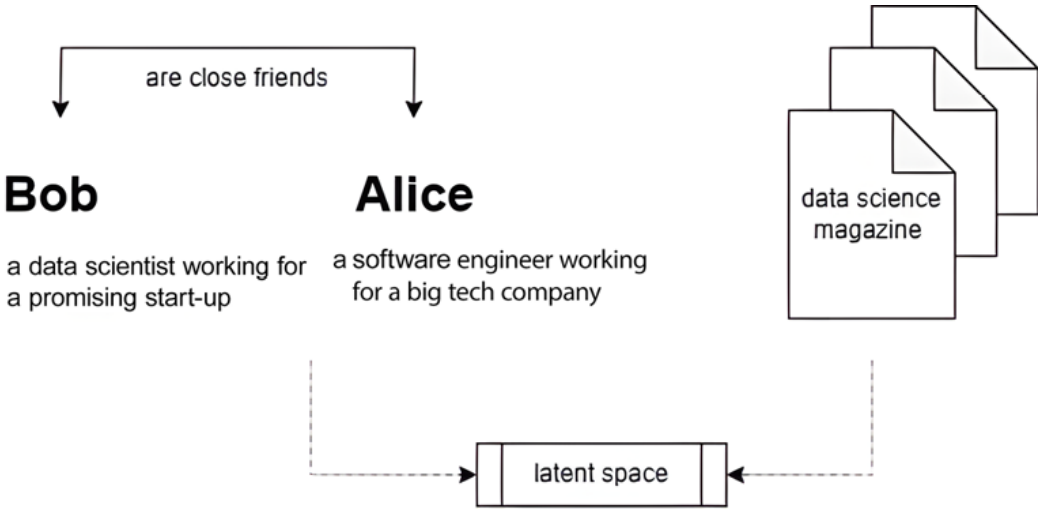
```
# Encode movie descriptions and add embeddings to the graph
for node in G.nodes(data=True):
    description = node[1]['description']
    embedding = encode_text(description)
    node[1]['embedding'] = embedding
```

```
import numpy as np
# Combine embeddings with structural features
for node in G.nodes(data=True):
    # We are using degree as a sample feature
    structural_features = np.array([G.degree[node[0]]])
    node[1]['combined_features'] = np.concatenate((node[1]
['embedding'], structural_features),axis=None)
```

```
from sklearn.metrics.pairwise import cosine_similarity
# Compute similarity between nodes based on combined features
node_features = [node[1]['combined_features'] for node in
```

```
G.nodes(data=True)]
similarity_matrix = cosine_similarity(node_features)
# Example: Find movies similar to 'Inception' (node 1)
movie_index = 0  # Index of the movie 'Inception'
# Let's take the top 2 similar
similar_movies = np.argsort(-similarity_matrix[movie_index])[1:3]
print("Movies similar to Inception:", similar_movies)
```

LLMs as aligners



Prediction alignment

1. The LLM analyzes text data and generates node labels, which will serve as pseudo-labels for the GNN.
2. The GNN then processes the graph structure and produces node labels based on connectivity and relationships, which are then fed back to the LLM.
3. The process is repeated with each model refining its prediction based on insights from the other.

```
from torch_geometric.data import Data
# Assume a toy dataset with 3 papers (nodes), edges, and labels
data = Data(
    x=torch.rand(3, 10),  # let's use random features for
```

```
simplicity
    edge_index=torch.tensor([[0, 1], [1, 2]], dtype=torch.long),
# Edges
    y=torch.tensor([0, 1, 2], dtype=torch.long),  # True labels
    text=["Paper A abstract", "Paper B abstract", "Paper C
abstract"],
    # Text data
)
```

```
# 1. Define the Graph Neural Network (GNN)
class GNN(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(GNN, self).__init__()
        self.conv1 = GCNConv(input_dim, hidden_dim)
        self.conv2 = GCNConv(hidden_dim, output_dim)
    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index).relu()
        x = self.conv2(x, edge_index)
        return x
# 2. Define the LLM (e.g., BERT for text encoding)
class TextEncoder(torch.nn.Module):
    def __init__(self, model_name="bert-base-uncased",
output_dim=128):
        super(TextEncoder, self).__init__()
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model = AutoModel.from_pretrained(model_name)
        self.fc = torch.nn.Linear(self.model.config.hidden_size,
output_dim)
    def forward(self, texts):
        # Tokenize and encode text data
        inputs = self.tokenizer(texts, return_tensors="pt",
padding=True, truncation=True)
        outputs = self.model(**inputs)
        cls_embedding = outputs.last_hidden_state[:, 0, :]
        # [CLS] token embedding
        return self.fc(cls_embedding)
```

```
# 3. Training Loop with Pseudo-label Exchange
def train_prediction_alignment(data, gnn, text_encoder,
num_iterations=5):
    optimizer_gnn = torch.optim.Adam(gnn.parameters(), lr=0.01)
    optimizer_text = torch.optim.Adam(text_encoder.parameters(),
lr=0.0001)

    # Initialize with true labels for first iteration
    gnn_pseudo_labels = data.y.clone()
    llm_pseudo_labels = data.y.clone()

    for iteration in range(num_iterations):
        # Train GNN using LLM pseudo-labels from previous
iteration
        gnn.train()
        optimizer_gnn.zero_grad()
        gnn_logits = gnn(data.x, data.edge_index)
        gnn_loss = torch.nn.CrossEntropyLoss()(gnn_logits,
llm_pseudo_labels)
        gnn_loss.backward()
        optimizer_gnn.step()

        # Generate new GNN pseudo-labels
        with torch.no_grad():
            gnn_pseudo_labels = torch.argmax(gnn_logits, dim=1)

        # Train Text Encoder using GNN pseudo-labels
        text_encoder.train()
        optimizer_text.zero_grad()
        text_logits = text_encoder(data.text)
        llm_loss = torch.nn.CrossEntropyLoss()(text_logits,
gnn_pseudo_labels)
        llm_loss.backward()
        optimizer_text.step()

        # Generate new LLM pseudo-labels for next iteration
        with torch.no_grad():
            llm_pseudo_labels = torch.argmax(text_logits, dim=1)

        print(f"Iteration {iteration+1}: GNN Loss =
{gnn_loss.item():.4f}, LLM Loss = {llm_loss.item():.4f}")
```

```
        print(f"  GNN predictions: {gnn_pseudo_labels.tolist()}")
        print(f"  LLM predictions: {llm_pseudo_labels.tolist()}")
```

Latent space alignment

1. Text Encoding: Use an LLM to encode the node descriptions into a latent vector.
2. Graph Encoding: Use a GraphML model (e.g., GNN) to encode the graph structure around each node into latent vectors.
3. Contrastive learning: Use contrastive learning to maximize the similarity between the text and graph encoding for the same node or neighbor nodes, while minimizing the similarity between unrelated nodes.

```
# Toy data with 3 products and their relationships
data = Data(
    x=torch.rand(3, 10),  # Node features
    edge_index=torch.tensor([[0, 1], [1, 2]], dtype=torch.long),
# Edges
    text=["Product A description", "Product B description",
"Product C description"],  # Text data
)
```

```
# Contrastive Learning Objective
def contrastive_loss(graph_emb, text_emb, tau=0.1):
    sim = F.cosine_similarity(graph_emb, text_emb)
    labels = torch.arange(sim.size(0)).to(sim.device)
    loss = F.cross_entropy(sim / tau, labels)
    return loss
```

```
# Training Loop for Latent Space Alignment
def train_latent_alignment(data, gnn, text_encoder, epochs=10):
    optimizer = torch.optim.Adam(list(gnn.parameters()) +
list(text_encoder.parameters()), lr=0.001)
    for epoch in range(epochs):
        optimizer.zero_grad()
```

```
        # Encode graph and text
        graph_emb = gnn(data.x, data.edge_index)  # Graph
embeddings
        text_emb = text_encoder(data.text)  # Text embeddings
        # Compute contrastive loss
        loss = contrastive_loss(graph_emb, text_emb)
        loss.backward()
        optimizer.step()
        print(f"Epoch {epoch+1}: Loss = {loss.item()}")
```

Building knowledge graphs from text

```
text = """
Marie Curie, born in 1867, was a Polish and naturalized-French
physicist and chemist who conducted pioneering research on
radioactivity.
She was the first woman to win a Nobel Prize, the first person to
win a Nobel Prize twice, and the only person to win a Nobel Prize
in two scientific fields.
"""
```

```
from langchain_experimental.graph_transformers.llm import
LLMGraphTransformer
from langchain_openai import ChatOpenAI
from langchain_core.documents import Document
llm = ChatOpenAI(temperature=0, model_name="minicpm-llama3-v-2_5",
base_url="http://localhost:1234/v1", api_key="lm-studio")
llm_transformer = LLMGraphTransformer(llm=llm)
documents = [Document(page_content=text)]
graph_documents =
llm_transformer.convert_to_graph_documents(documents)
print(f"Nodes: {graph_documents[0].nodes}")
print(f"Relationships: {graph_documents[0].relationships}")
```

GraphRAG

First of all, we need to start our Neo4j server. It will act as a backend for storing the KG and performing the RAG operations.

```
docker run --rm --detach --name neo4j \
              --publish=7474:7474 --publish=7687:7687 \
              --env NEO4J_AUTH=neo4j/defaultpass \
              --env NEO4J_PLUGINS='["apoc-extended"]' \
              --env NEO4J_apoc_export_file_enabled=true \
              --env NEO4J_apoc_import_file_enabled=true \
              --env NEO4J_apoc_import_file_use__neo4j_config=true
\
              neo4j:5.26.0
```

```
from neo4j import GraphDatabase
from langchain_neo4j import Neo4jGraph
NEO4J_URI = "bolt://localhost:7687"
NEO4J_USER = "neo4j"
NEO4J_PASSWORD = "your_password"
driver = GraphDatabase.driver(NEO4J_URI, auth=(NEO4J_USER,
NEO4J_PASSWORD))
graph = Neo4jGraph(url=NEO4J_URI, username=NEO4J_USER,
password=NEO4J_PASSWORD)
```

```
graph.add_graph_documents(graph_documents)
```

```
CYPHER_GENERATION_TEMPLATE = """You are a Neo4j expert. Generate a
Cypher query to answer the given question.
Database Schema:
- Nodes:
  * Person (properties: id)
  * Award (properties: id)
```

```
- Relationships:
  * (Person)-[:MARRIED_TO]-(Person)
  * (Person)-[:WON_NOBEL_PRIZE]->(Award)
Rules:
1. Always use explicit `MATCH` for relationships.
2. Never use `WHERE` for relationship matching.
3. Use `RETURN DISTINCT` when appropriate.
Example Queries:
1. Question: "Who won the Nobel Prize?"
   Cypher: MATCH (p:Person)-[:WON_NOBEL_PRIZE]->(:Award) RETURN
p.id AS winner
Question: {query}
Return only the Cypher query without any explanation or additional
text.
Cypher:"""
```

```
from langchain_neo4j import GraphCypherQAChain
from langchain_core.prompts import PromptTemplate
chain = GraphCypherQAChain.from_llm(
    llm=llm,
    graph=graph,
    verbose=True,
    cypher_prompt=PromptTemplate(
        input_variables=["query"],
        template=CYPHER_GENERATION_TEMPLATE
    ),
    allow_dangerous_requests=True
)
```

```
question = "Who married a Nobel Prize winner?"
print(f"\nQuestion: {question}")
response = chain.invoke(question)
print("Response:", response['result'])
# Close the driver
driver.close()
```

Reference: Graph Machine Learning, Aldo Marzullo, Enrico Deusebio, Claudio Stamile