Faculty of Media Engineering and Technology
German University in Cairo
Dr. Nourhan Ehab

CSEN 903: Advanced Computer Lab, Winter 2025
Lab 7 Manual: Introduction to Neuro-Symbolic AI

Part 1: Read and Watch Before the Lab

In this lab, you will build upon what you learned in **Lab 6 (Knowledge Graphs)** and explore how to combine structured knowledge with machine learning models for **Natural Language Processing (NLP)**.

Traditional NLP models process raw text data but often struggle with contextual reasoning or understanding relationships between entities. By integrating **Knowledge Graphs (KGs)**, we can provide these models with external, structured information.

You will learn how to:

- Extract entities from text using **spaCy**.
- Build a **Neo4j Knowledge Graph** from these entities.
- Engineer **knowledge-based features** that enrich your dataset.
- Compare a **baseline text classifier** with a **knowledge-augmented model**.

Required Read List:

1. The Future of AI: Machine Learning and Knowledge Graphs
2. Build a Knowledge Graph in NLP - GeeksforGeeks
3. Knowledge Graphs With Machine Learning [Guide]
4. Integrating LLM with Knowledge Graph | by Hakeem Abbas | Medium

Required Watch List:

1. Training Series - Create a Knowledge Graph: A Simple ML Approach

Part 2: In-Lab Task

We will use the IMDb movie review dataset. You will find the notebook ready on Kaggle

First, we might need to install the requirements

```
!pip install pandas spacy neo4j networkx scikit-learn
!python -m spacy download en_core_web_sm
!pip install --upgrade --force-reinstall numpy==1.26.4 scikit-learn==1.3.2
!pip install --force-reinstall numpy==1.26.4
!pip install --force-reinstall scipy==1.10.1
!pip install --force-reinstall scikit-learn==1.3.2
```

Then we import our libraries

```python
import pandas as pd
import spacy
from neo4j import GraphDatabase
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, accuracy_score,
precision_score, recall_score, f1_score
from sklearn.model_selection import train_test_split
from scipy.sparse import hstack
import numpy as np
from collections import Counter, defaultdict
```

Step 1: Load and Preprocess the Dataset

In most NLP workflows, we start with unstructured text data. To connect this data with a Knowledge Graph, we must first extract meaningful entities and clean the text so that both symbolic and statistical representations align. This prepares the dataset for hybrid learning, where structured knowledge complements statistical patterns in the text.

First, we'll load the IMDB dataset and create a balanced subset for our experiments. This ensures we have equal representation of positive and negative reviews.

```
df = pd.read_csv('/kaggle/input/imdb-dataset-of-50k-movie-reviews/IMDB
Dataset.csv')
df_neg = df[df['sentiment'] == 'negative'].sample(n=100, random_state=42)
df_pos = df[df['sentiment'] == 'positive'].sample(n=100, random_state=42)
df = pd.concat([df_neg,
df_pos]).sample(frac=1,random_state=42).reset_index(drop=True)
df.head()
```

Step 2: Extract Named Entities Using spaCy

Named Entity Recognition (NER) acts as the bridge between text and the Knowledge Graph. Each detected entity (person, movie, organization) becomes a node candidate in the KG. This step shows how unstructured linguistic data can be transformed into structured, machine-readable knowledge that can be queried and reasoned about.

We'll use spaCy's Named Entity Recognition (NER) to extract entities from each review. We focus on three entity types:

- PERSON: People mentioned (actors, directors, etc.)

- WORK_OF_ART: Creative works (movies, books, etc.)

- ORG: Organizations (studios, companies, etc.)

These entities will later be used to build our Knowledge Graph.

```
nlp = spacy.load("en_core_web_sm")
def extract_entities(text):
    """
    Returns: List of (entity_text, entity_label) tuples
    - PERSON: People (actors, directors, etc.)
    - WORK_OF_ART: Creative works (movies, books, etc.)
    - ORG: Organizations (studios, companies, etc.)
    """
    doc = nlp(text)
    return [(ent.text.strip(), ent.label_) for ent in doc.ents
            if ent.label_ in ["PERSON", "WORK_OF_ART", "ORG"]]
# Extract entities from all reviews
df["entities"] = df["review"].apply(extract_entities)
```

Step 3: Connect to Neo4j Database

Neo4j is used as the underlying engine to represent and query the Knowledge Graph. Connecting to Neo4j allows us to insert, query, and retrieve symbolic knowledge that can later enrich the machine learning model. This connection enables the hybrid architecture where graph queries provide structured context to complement statistical learning.

We'll use the Neo4j Python driver to connect to our Knowledge Graph database. For this lab, we'll use an Aura (cloud) Neo4j instance, but you can also use a local Neo4j instance by changing the URI.

```
uri = '<uri>'
user = "neo4j"
password = '<password>'
driver = GraphDatabase.driver(uri, auth=(user, password))
print("Connected to Neo4j database")
```

Step 4: Build Knowledge Graph from Extracted Entities

Here, we move from raw entities to a structured network of relationships. This step demonstrates how context, such as which actors co-occur with which movies, can be formalized as graph relationships. Such connections capture semantics that purely text-based models usually ignore. The graph structure enables reasoning about relationships and patterns that emerge from the data.

Now we'll construct a Knowledge Graph from the entities we extracted. The graph structure will be:

- WORK_OF_ART entities → Movie nodes

- PERSON entities → Actor nodes

- Relationships created based on co-occurrence in reviews

We'll analyze the entities and build the graph structure in Neo4j based on co-occurrence patterns.

```python
# Analyze entities to understand what we have
person_entities = Counter()  # Count occurrences of PERSON entities
work_of_art_entities = Counter()  # Count occurrences of WORK_OF_ART entities
for idx, row in df.iterrows():
    entities = row['entities']

    for ent_text, ent_label in entities:
        if ent_label == "PERSON":
            person_entities[ent_text] += 1
        elif ent_label == "WORK_OF_ART":
            work_of_art_entities[ent_text] += 1
# Build co-occurrence relationships: which persons appear with which movies
# This creates HAS_ACTOR relationships in the KG
person_movie_cooccurrence = defaultdict(set)
for idx, row in df.iterrows():
    entities = row['entities']
    persons = [ent_text for ent_text, ent_label in entities if ent_label == 
"PERSON"]
    movies = [ent_text for ent_text, ent_label in entities if ent_label == 
"WORK_OF_ART"]
    # If a person and movie appear in the same review, create a relationship
    for person in persons:
        for movie in movies:
            person_movie_cooccurrence[person].add(movie)
```

Build the knowledge graph in Neo4j using *UNWIND* and *Batching*:

1. ***UNWIND***: This is a Cypher clause that takes a list/array and "unwinds" it into individual rows.

For example, if you have a list ["Movie1", "Movie2", "Movie3"], UNWIND converts it into:- Row 1: "Movie1"

- Row 2: "Movie2"

- Row 3: "Movie3"

This allows us to process multiple items in a single query instead of making separate queries for each item, which is much faster.

2. ***Batching***: Instead of processing all data at once (which could cause memory issues or timeouts), we split large operations into smaller "batches" (e.g., 1000 items at a time). This approach:

- Reduces memory usage

- Prevents query timeouts

- Allows for better error handling

- Improves overall performance

Example of *UNWIND* in action:

Instead of:

for name in movie_names:

tx.run("MERGE (m:Movie {name: $name})", name=name) # Slow: one query per movie

We use:

tx.run("UNWIND $names AS name MERGE (m:Movie {name: name})", names=movie_names)

```
def build_kg(tx):
    # Clear existing data
    tx.run("MATCH (n) DETACH DELETE n")
    # Create Movie nodes from WORK_OF_ART entities
    movie_names = list(work_of_art_entities.keys())
    if movie_names:
        tx.run(
            """
            UNWIND $names AS name
            MERGE (m:Movie {name: name})
            """,
            names=movie_names
        )
    # Create Actor nodes from PERSON entities
    actor_names = list(person_entities.keys())
    if actor_names:
        tx.run(
            """
            UNWIND $names AS name
            MERGE (a:Actor {name: name})
            """,
            names=actor_names
        )
    # Batch create HAS_ACTOR relationships based on co-occurrence
    # Prepare list of (movie, actor) pairs
    actor_relationships = []
    for person, movies in person_movie_cooccurrence.items():
        for movie in movies:
            actor_relationships.append({"movie": movie, "actor": person})

    relationship_count = len(actor_relationships)
    if actor_relationships:
        # Process in batches to avoid memory issues
        batch_size = 1000
        for i in range(0, len(actor_relationships), batch_size):
            batch = actor_relationships[i:i + batch_size]
            tx.run(
                """
                UNWIND $pairs AS pair
                MATCH (m:Movie {name: pair.movie})
                MATCH (a:Actor {name: pair.actor})
                MERGE (m)-[:HAS_ACTOR]->(a)
                """,
                pairs=batch
            )

    return relationship_count

with driver.session() as session:
    rel_count = session.execute_write(build_kg)
# Save intermediate results
df.to_csv('output_lab8.csv', index=False)
```

Step 5: Extract Knowledge Graph Features

Integrating KGs into ML means converting relational data into features. The "has_popular_actor" feature illustrates this process: symbolic relationships become numerical signals that machine learning models can understand. This transformation allows us to combine the interpretability of symbolic knowledge with the learning power of statistical models.

For each review, we'll extract 1 binary feature from the KG:

1. has_popular_actor: 1 if review mentions any actor connected to >3 movies

This feature captures structured knowledge not directly present in the text. The key idea is to query the Knowledge Graph to check if any entity mentioned in a review corresponds to a popular actor (one connected to more than 3 movies).

```
def query_neo4j(query, parameters={}):
    """Execute a Cypher query and return results as a list of dictionaries."""
    with driver.session() as session:
        result = session.run(query, parameters)
        records = [record.data() for record in result]
        return records
```

Extract KG features for a single entity. This function checks if an entity is a popular actor by querying the Knowledge Graph. An actor is considered "popular" if they are connected to more than 3 movies.

```python
def get_kg_features_for_entity(entity_name, entity_label):
    features = {
        'has_popular_actor': 0
    }
    # Feature: has_popular_actor
    # Check if this entity is an Actor connected to more than 3 movies
    # We use OPTIONAL MATCH to handle cases where the actor doesn't exist in the KG
    actor_query = """
    OPTIONAL MATCH (a:Actor {name: $entity_name})
    OPTIONAL MATCH (m:Movie)-[:HAS_ACTOR]->(a)
    WITH a, COUNT(DISTINCT m) as movie_count
    WHERE a IS NOT NULL AND movie_count > 3
    RETURN a.name AS actor
    """
    try:
        result = query_neo4j(actor_query, {"entity_name": entity_name})
        if result and len(result) > 0 and result[0].get('actor'):
            features['has_popular_actor'] = 1
    except (Exception, IndexError, KeyError):
        pass

    return features
```

Aggregate KG features across all entities in a review.

Uses OR logic: if ANY entity in the review has a feature, the review gets it. For example: if any actor mentioned is popular, has_popular_actor = 1

This function loops through all entities in a review and checks each one.

If any entity is a popular actor, the review gets has_popular_actor = 1.

```python
def get_kg_features_for_review(review_entities):
    all_features = {
        'has_popular_actor': 0
    }
    if not review_entities or len(review_entities) == 0:
        return all_features

    # Loop through all entities in the review and check their features
    for ent_text, ent_label in review_entities:
        entity_features = get_kg_features_for_entity(ent_text, ent_label)
        # Use OR logic: if any entity has the feature, set it to 1
        # max() implements OR logic for binary features (0 or 1)
        all_features['has_popular_actor'] = max(all_features['has_popular_actor'],
                                                 entity_features['has_popular_actor'])

    return all_features
```

Extract KG features for all reviews

```
kg_features_list = []
for idx, entities in enumerate(df["entities"]):
    if (idx + 1) % 100 == 0:
        print(f"  Processed {idx + 1}/{len(df)} reviews...")
    features = get_kg_features_for_review(entities)
    kg_features_list.append(features)
# Convert to DataFrame
kg_features_df = pd.DataFrame(kg_features_list)
kg_features_df.head()
```

Saving the results to be able to recreate the same experiment

```
df_with_kg = df.copy()
df_with_kg['has_popular_actor'] = kg_features_df['has_popular_actor']
df_with_kg.to_csv('output_lab8.csv', index=False)
```

Step 6: Train Classification Models

This step tests the value of symbolic knowledge. By comparing models with and without KG-derived features, we can evaluate whether structured reasoning helps improve predictive performance, a key question in knowledge-infused learning. This comparison demonstrates the practical benefits of combining symbolic and statistical approaches.

We'll train two models to compare:

- Model A (Baseline): Uses only text features (TF-IDF)

- Model B (KG-Augmented): Uses text features + 1 KG feature (has_popular_actor)

```
X_text = df['review']
y = df['sentiment'].map({'positive': 1, 'negative': 0})
vectorizer = TfidfVectorizer(max_features=3000)
X_vect = vectorizer.fit_transform(X_text)
```

Model A: Baseline (Text Only)

```
X_train, X_test, y_train, y_test = train_test_split(X_vect, y, test_size=0.2,
random_state=42)

model_text = LogisticRegression(max_iter=1000, random_state=42)
model_text.fit(X_train, y_train)
y_pred_text = model_text.predict(X_test)

baseline_accuracy = accuracy_score(y_test, y_pred_text)
baseline_precision = precision_score(y_test, y_pred_text, average='weighted')
baseline_recall = recall_score(y_test, y_pred_text, average='weighted')
print(f"\nBaseline Model - Accuracy: {baseline_accuracy:.4f}, Precision:
{baseline_precision:.4f}, Recall: {baseline_recall:.4f}")
```

Model B: Knowledge-Augmented (Text + KG Features)

```
kg_array = kg_features_df.values
X_with_kg = hstack([X_vect, kg_array])

X_train_kg, X_test_kg, y_train_kg, y_test_kg = train_test_split(X_with_kg, y,
test_size=0.2, random_state=42)
model_kg = LogisticRegression(max_iter=1000, random_state=42)
```

```
model_kg.fit(X_train_kg, y_train_kg)
y_pred_kg = model_kg.predict(X_test_kg)

kg_accuracy = accuracy_score(y_test_kg, y_pred_kg)
kg_precision = precision_score(y_test_kg, y_pred_kg, average='weighted')
kg_recall = recall_score(y_test_kg, y_pred_kg, average='weighted')
print(f"\nKG-Augmented Model - Accuracy: {kg_accuracy:.4f}, Precision:
{kg_precision:.4f}, Recall: {kg_recall:.4f}")
```

Step 7: Compare Model Performance

Finally, we'll compare the performance of both models across multiple metrics to see if the KG features provide any improvement. The final comparison illustrates how integrating structured knowledge enhances interpretability and may lead to performance gains. In real-world NLP applications, such hybrid architectures are at the core of retrieval-augmented generation (RAG) and neurosymbolic learning systems. This lab demonstrates the fundamental principle: combining symbolic knowledge with statistical learning can create more robust and interpretable AI systems.

Finally, we close the connection.

```
driver.close()
```

In this lab, you have implemented an end-to-end **knowledge-augmented sentiment classification pipeline**. You learned how to extract entities from text, construct a Knowledge Graph in Neo4j, and use graph-derived features to enhance a machine learning model.

This exercise demonstrates how **structured knowledge can be fused with machine learning** to achieve better reasoning and contextual understanding; a foundation for modern **neurosymbolic AI**, **graph neural networks (GNNs)**, and **knowledge-grounded NLP systems**.

Faculty of Media Engineering and Technology
German University in Cairo
Dr. Nourhan Ehab

<u>Part 3: Graded Task</u>

Task Description

Using the IMDb review subset and the provided Neo4j Knowledge Graph, perform the following tasks:

1. Load the dataset and extract named entities using spaCy (you can reuse the code from Part 2 or build the KG if needed).
2. Feature Engineering: Extract the following knowledge-based features from the KG for each review:
    a. has_popular_actor: 1 if the review mentions any actor connected to more than 3 movies, 0 otherwise.
    b. actor_movie_count: The maximum number of movies that any actor mentioned in the review is connected to.
        i. If no actors are mentioned, set to 0
        ii. If multiple actors are mentioned, use the maximum count
        iii. Example: If review mentions Actor A (connected to 5 movies) and Actor B (connected to 2 movies), then actor_movie_count = 5
    c. mentioned_movie_count: The number of unique movies (WORK_OF_ART entities) mentioned in the review that exist in the KG.
        i. Count how many WORK_OF_ART entities from the review are present as Movie nodes in the KG
        ii. This feature captures how "recognizable" the movies in the review are
3. Train and compare TWO classification models:
    a. Model A (Single KG Feature): Uses text features + has_popular_actor
    b. Model B (All KG Features): Uses text features + all three KG features (has_popular_actor, actor_movie_count, mentioned_movie_count)
4. Performance Analysis:
    a. Compare the two models on accuracy, precision, and recall
    b. Create a comparison table showing all metrics
    c. Report your findings: Which model performs best? Do additional KG features help?

Deliverables: Submit an .ipynb notebook with all the completed tasks above to the form by the end of your lab.