**Phase 1: Define Project Scope and Prepare the Environment**

**1. Define Project Objectives**

* **Objective**: Automate the extraction of entities and relationships from SOPs to build a structured knowledge graph.
* **Outcomes**:
  + Reduce hallucinations in the LLM.
  + Enable structured and consistent information for querying.
  + Facilitate automation for future SOPs.

**2. Identify Key Entities and Relationships**

* **Entities**: Roles (e.g., "Loan Officer"), Procedures (e.g., "Loan Application Process"), Documents (e.g., "Loan Application Form"), Regulations (e.g., "AML Regulation").
* **Relationships**: "is responsible for", "must comply with", "requires", "approves", "follows", etc.

**3. Set Up the Development Environment**

* **Tools**: Install Python, required libraries (spaCy, Hugging Face, Neo4j, etc.).
* **Graph Database**: Set up Neo4j or GraphDB.
* **IDE**: Use Visual Studio Code, Jupyter Notebook, or any preferred Python IDE.

**Phase 2: Text Preprocessing**

**1. Text Cleaning**

* Remove irrelevant characters, unnecessary white spaces, footers, headers, or special symbols from SOPs.

**Tools**: Python’s re (regular expressions) or spaCy

!pip install PyPDF2

import re

import PyPDF2

def clean\_text(text):

    # Remove unwanted characters

    text = re.sub(r'\s+', ' ', text)  # Replace multiple spaces with a single space

    text = re.sub(r'\n', ' ', text)  # Remove newline characters

    text = re.sub(r'[^A-Za-z0-9\s.,;!?]', '', text)  # Remove special characters

    return text

def read\_pdf(file\_path):

    # Open the PDF file

    with open(file\_path, 'rb') as file:

        reader = PyPDF2.PdfReader(file)

        # Extract text from each page

        text = ''

        for page in reader.pages:

            text += page.extract\_text()

    # Clean the extracted text

    cleaned\_text = clean\_text(text)

    return cleaned\_text

# Example usage

file\_path = 'C1-Annex-2.pdf'  # Replace with your PDF file path

cleaned\_text = read\_pdf(file\_path)

print(cleaned\_text)

**2. Tokenization and Sentence Segmentation**

* Break down the cleaned text into individual tokens (words) and sentences for further processing.

**Tools**: spaCy (for tokenization), nltk (for sentence segmentation)

import spacy

nlp = spacy.load("en\_core\_web\_sm")

def tokenize\_text(text):

    doc = nlp(text)

    tokens = [token.text.lower() for token in doc if not token.is\_stop and not token.is\_punct]

    return tokens

print(tokenize\_text(cleaned\_text))

**Phase 3: Named Entity Recognition (NER)**

**1. Extract Entities Using Pre-Trained Models**

* Use spaCy’s pre-trained NER model to extract common entities like people, organizations, and dates.

**Example**:

def extract\_entities(text):

doc = nlp(text)

entities = [(ent.text, ent.label\_) for ent in doc.ents]

return entities

# Example usage

text = "The Loan Officer is responsible for verifying the Loan Application Form."

entities = extract\_entities(text)

print(entities)

**2. Custom NER Model for Domain-Specific Entities**

* Extend NER to recognize domain-specific entities such as roles (e.g., "Loan Officer"), procedures (e.g., "Loan Application Process"), and regulations (e.g., "AML").
* **Train a Custom NER Model** using annotated data (using spaCy’s custom training pipeline).

**Training Process**: Label a set of SOPs with entities, then fine-tune spaCy's model.

**spaCy Custom NER Training**:

* [spaCy Custom NER Tutorial](https://spacy.io/usage/training)
* <https://www.youtube.com/watch?v=p_7hJvl7P2A>
* def extract\_entities(text):
* doc = nlp(text)
* entities = [(ent.text, ent.label\_) for ent in doc.ents]
* return entities
* print(extract\_entities(cleaned\_text))
* import spacy
* from spacy.tokens import DocBin
* from tqdm import tqdm
* import json
* # Load a new blank spaCy model
* nlp = spacy.blank("en")
* # Create a DocBin object
* db = DocBin()
* # Load training data from JSON file
* f = open('annotations.json')
* TRAIN\_DATA = json.load(f)
* # Process training data and convert it to spaCy format
* for text, annot in tqdm(TRAIN\_DATA['annotations']):
* doc = nlp.make\_doc(text)
* ents = []
* for start, end, label in annot["entities"]:
* span = doc.char\_span(start, end, label=label, alignment\_mode="contract")
* if span is None:
* print("Skipping entity")
* else:
* ents.append(span)
* doc.ents = ents
* db.add(doc)
* # Save the processed data as a spaCy binary file
* db.to\_disk("./pvr\_training\_data.spacy")

!python -m spacy init config config.cfg --lang en --pipeline ner --optimize efficiency

!python -m spacy train config.cfg --output ./model-best --paths.train ./pvr\_training\_data.spacy --paths.dev ./pvr\_training\_data.spacy

nlp\_nor = spacy.load("./model-best/model-best") # Include "model-best" folder within the path.

doc = nlp\_nor(cleaned\_text)  # Input sample text

spacy.displacy.render(doc, style="ent", jupyter=True)  # display in Jupyter

**3. Handle Non-Standard Entities:**

* Manually define rules (using spaCy or regular expressions) to capture terms that aren’t identified by default NER models, such as specific policies, forms, or roles.

**Phase 4: Relationship Extraction**

**1. Dependency Parsing for Relationship Identification**

* Use **dependency parsing** to identify syntactic relationships between entities (e.g., subject-object relationships).
* import spacy
* from spacy import displacy
* nlp = spacy.load("en\_core\_web\_sm")
* # Assuming 'cleaned\_text' is the variable holding the output of read\_pdf function
* sentence = cleaned\_text.split('.')
* for i in sentence:
* doc = nlp(i)
* print(f"{'Node (from)-->':<15} {'Relation':^10} {'-->Node (to)':>15}\n")
* for token in doc:
* print("{:<15} {:^10} {:>15}".format(str(token.head.text), str(token.dep\_), str(token.text)))
* displacy.render(doc, style='dep')
* **2. Rule-Based Relationship Extraction**
* Develop regular expressions (or rule-based logic) to extract common relationships such as "is responsible for", "complies with", "requires", etc.

**Example Relationship Extraction Rules**:

!pip install stanza

import stanza

stanza.download('en')

nlp = stanza.Pipeline(lang='en', processors='tokenize,mwt,pos,lemma,depparse')

# Call the clean\_text function and pass its output to the pipeline

# The variable name was incorrect. Changed 'clean\_text' to 'cleaned\_text'

doc = nlp(cleaned\_text)  # Assuming 'cleaned\_text' is the variable containing the raw text

for sent in doc.sentences:

    for word in sent.words:

        print(f'id:{word.id}\nword: {word.text}\nhead id: {word.head}\nhead: {sent.words[word.head-1].text if word.head > 0 else "root"}\tdeprel: {word.deprel} \n --------------\n', sep='.')

**3. Train Relationship Extraction Models**

* Fine-tune a pre-trained model (e.g., BERT) to detect specific relationships between entities in your domain.

**Tools**: Hugging Face Transformers, spaCy

**Phase 5: Construct Knowledge Graph**

**1. Design the Knowledge Graph Schema**

* Define the **nodes** (e.g., roles, procedures, regulations) and **relationships** (e.g., "responsible\_for", "complies\_with").
* Nodes are the entities you identified.
* Relationships are the links between entities.

**2. Set Up a Graph Database**

* **Install and Configure Neo4j** (or GraphDB).
* **Use the Neo4j Python driver** (py2neo) to insert nodes and relationships into the graph.

**Neo4j Example**:

from py2neo import Graph, Node, Relationship

# Connect to Neo4j

graph = Graph("bolt://localhost:7687", auth=("neo4j", "password"))

# Create Nodes

loan\_officer = Node("Role", name="Loan Officer")

loan\_application\_form = Node("Document", name="Loan Application Form")

# Create Relationship

responsible\_for = Relationship(loan\_officer, "RESPONSIBLE\_FOR", loan\_application\_form)

# Create Nodes and Relationship in the Database

graph.create(responsible\_for)

**3. Populate the Graph with Extracted Data**

* Using the entities and relationships you extracted, programmatically populate the graph with data.

**Example Process**:

* Extract entities using NER.
* Extract relationships based on dependency parsing or regex.
* Insert entities as nodes and relationships as edges into the graph database.

**4. Optimizing Graph Structure**

* Use labels and indexes in Neo4j to improve query performance.
* Example: Add labels like Role, Procedure, Regulation to differentiate types of entities.

**Phase 6: Querying and Integrating with LLM**

**1. Create Queries for Information Retrieval**

* Use **Cypher** (Neo4j query language) to query the knowledge graph for relevant data.

**Example Query**:

MATCH (r:Role)-[:RESPONSIBLE\_FOR]->(p:Procedure)

WHERE r.name = "Loan Officer"

RETURN p.name

**2. Integrate Knowledge Graph with LLM**

* **Pass queries to the LLM**: Whenever the LLM requires information from the SOPs, it queries the knowledge graph and retrieves structured information.
* **Reduce Hallucination**: By querying the graph, the LLM can provide factual information based on the knowledge graph instead of generating responses from unreliable sources.

**3. Automate Querying for Real-Time Access**

* Create an **API** or service that the LLM can call to retrieve the graph information, ensuring dynamic interaction with the graph as new SOPs are processed.

**Phase 7: Continuous Improvement and Monitoring**

**1. Evaluate and Improve Accuracy**

* Periodically review and fine-tune the NER and relationship extraction models.
* Add new SOPs, update existing ones, and retrain models as necessary.

**2. Feedback Loop**

* Collect feedback from the LLM’s output to identify areas where hallucination is still occurring.
* Refine extraction methods, update graph data, and improve knowledge graph accuracy.

**Tools and Technologies for Automation**

* **spaCy**: NER, dependency parsing, tokenization
* **Hugging Face Transformers**: Advanced NLP tasks, fine-tuning models
* **Neo4j**: Graph database for knowledge graph
* **py2neo**: Python client for interacting with Neo4j
* **regex**: Relationship extraction rules
* **Flask/FastAPI**: Build an API for querying the graph