Web Semantics and Datamining

ESILV A4 - DIA2

Élèves :
Hugo BONNELL
Lucas BLANCHET

Enseignants :
Yiru ZHANG
Walid GAALOUL

9 mars 2025

Table des matières

1 Introduction

In today's information-rich world, textual data is being generated at an unprecedented rate through news articles, social media, and websites. While this data contains valuable information, it is predominantly unstructured, making it challenging to process and analyze efficiently. Knowledge Graphs (KGs) address this challenge by organizing information in a structured format, enabling advanced querying and reasoning capabilities.

This project focuses on building a complete pipeline for Knowledge Graph Construction from raw text data. The pipeline integrates several Natural Language Processing (NLP) techniques, including web scraping, text preprocessing, Named Entity Recognition (NER), Relation Extraction (RE), and Knowledge Graph visualization. By transforming unstructured news articles into a structured knowledge graph, we create a queryable knowledge base that can offer insights and connections that might otherwise remain hidden in the text.

Our implementation targets news articles from Reuters, extracting entities such as people, organizations, locations, and dates, along with the relationships between them. The resulting knowledge graph provides a semantic representation of the news domain, allowing for complex queries and inferences about the real-world entities and their relationships.

2 Project Overview and Architecture

The project implements a comprehensive pipeline for constructing knowledge graphs from web content, particularly news articles. The system architecture consists of six main components that work together to transform unstructured text into a structured, queryable knowledge graph.

2.1 System Architecture

Our knowledge graph construction pipeline consists of the following components :

1. **Data Collection** : Web scraping module to collect news articles from Reuters.
2. **Text Preprocessing** : Cleaning and normalization of raw text.
3. **Named Entity Recognition** : Extraction of entities using both SpaCy and CRF models.
4. **Relation Extraction** : Identification of relationships between entities.
5. **Knowledge Graph Construction** : Creation of RDF triples and graph database.
6. **Visualization and Querying** : Interactive visualization and SPARQL querying.

These components are integrated into a seamless pipeline that can be executed end-to-end, from web scraping to knowledge graph visualization. Each component is implemented as a separate module, allowing for modularity and easy maintenance.

2.2 Dependencies and Technologies

The project utilizes various libraries and technologies :

- **Python** : Core programming language
- **BeautifulSoup and Selenium** : Web scraping
- **NLTK and SpaCy** : Text processing and NLP
- **sklearn-crfsuite** : Conditional Random Fields for NER
- **RDFLib** : Resource Description Framework for knowledge graph storage
- **NetworkX and PyVis** : Graph visualization
- **Pandas and NumPy** : Data manipulation and analysis

3 Data Collection Module

3.1 Web Scraping Implementation

The data collection module is responsible for gathering news articles from Reuters. We implemented a web scraper using a combination of Selenium and BeautifulSoup, which can navigate through dynamic web pages, extract article content, and save it in a structured format.

Listing 1 – Web Scraping Implementation

```python
class NewsArticleScraper:
    """Class for scraping news articles from various websites."""

    def __init__(self, output_dir='data/raw'):
        """
        Initialize the scraper.

        Args:
            output_dir (str): Directory to save scraped articles
        """
        self.output_dir = output_dir
        self._setup_output_dir()
        self.session = requests.Session()
        self.headers = {
            'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/109.0.0.0 Safari/537.36',
            # Additional headers...
        }



```

```python
    def scrape_reuters(self, num_articles=10, category='business'):
        """
        Scrape articles from Reuters.

        Args:
            num_articles (int): Number of articles to scrape
            category (str): News category

        Returns:
            list: List of filepaths to saved articles
        """
        # Implementation details...
```

The scraper navigates to the Reuters website, identifies article links, and extracts relevant information including the article title, content, publication date, and URL. Each article is saved as a JSON file in the designated output directory.

3.2 Data Structure and Storage

Scraped articles are stored in JSON format with the following structure :

Listing 2 – JSON Structure for Scraped Articles

```json
{
    "id": "example1",
    "title": "Apple announces new partnership with Microsoft",
    "url": "https://example.com/article1",
    "source": "reuters",
    "category": "business",
    "published_date": "2023-01-01",
    "scraped_date": "2023-01-02",
    "content": "Apple Inc. has announced a new partnership with Microsoft Corporation, according to CEO Tim Cook..."
}
```

This structured format facilitates further processing and ensures all relevant metadata is captured alongside the article content.

3.3 Handling Dynamic Content and Rate Limiting

To handle dynamic content and avoid being blocked by the website, the scraper implements several strategies :

- **Random delays** : Introducing random pauses between requests to mimic human behavior.
- **Browser emulation** : Using Selenium with ChromeDriver to render JavaScript and access dynamic content.
- **Session management** : Clearing cookies between requests to prevent tracking.

4 Text Preprocessing Module

4.1 Text Cleaning and Normalization

The text preprocessing module handles cleaning and normalization of the raw text from scraped articles. This is a crucial step for improving the accuracy of subsequent NLP tasks.

Listing 3 – Text Cleaning Implementation

```python
def clean_text(text, lowercase=True, remove_html=True, remove_special_chars=True,
               keep_hyphens=True, remove_extra_spaces=True, remove_stops=True,
               lemmatize=True, use_spacy=False):
    """
    Clean and preprocess text.

    Args:
        text (str): Input text
        lowercase (bool): Convert text to lowercase
        remove_html (bool): Remove HTML tags
        remove_special_chars (bool): Remove special characters
        keep_hyphens (bool): Keep hyphens for compound words
        remove_extra_spaces (bool): Remove extra whitespace
        remove_stops (bool): Remove stopwords
        lemmatize (bool): Lemmatize text
        use_spacy (bool): Use spaCy for lemmatization

    Returns:
        str: Cleaned and preprocessed text
    """
    # Implementation details...
```

The module implements several text cleaning functions :

- **HTML Removal** : Eliminating HTML tags and markup.
- **Special Character Handling** : Removing or preserving special characters as needed.
- **Whitespace Normalization** : Standardizing spacing within the text.
- **Stopword Removal** : Eliminating common words that don't carry significant meaning.
- **Lemmatization** : Reducing words to their base forms.

For named entity recognition, we carefully preserve case information and certain special characters that may be relevant for entity identification.

4.2 Tokenization Strategies

The preprocessing module employs different tokenization strategies depending on the subsequent tasks :

- **Word tokenization** : Breaking text into individual words for basic processing.
- **Sentence tokenization** : Identifying sentence boundaries for context-aware processing.
- **Preserving entity boundaries** : Ensuring entity spans are not broken during tokenization.

5 Named Entity Recognition (NER) Module

5.1 SpaCy NER Implementation

We implemented a named entity recognizer using SpaCy's pre-trained models, which can identify entities such as people, organizations, locations, dates, and more.

Listing 4 – SpaCy NER Implementation

```python
class SpacyNERExtractor(NERExtractor):
    """Named entity recognition using spaCy."""

    def __init__(self, model_name="en_core_web_sm"):
        """
        Initialize the spaCy NER extractor.

        Args:
            model_name (str): Name of the spaCy model to use
        """
        super().__init__()
        try:
            self.nlp = spacy.load(model_name)
            logger.info(f"Loaded spaCy model: {model_name}")
```

```
        except Exception as e:
            logger.error(f"Error loading spaCy model: {e}")
            logger.info(f"Downloading spaCy model: {model_name}")
            spacy.cli.download(model_name)
            self.nlp = spacy.load(model_name)

    def extract_entities(self, text):
        """
        Extract entities from text using spaCy.

        Args:
            text (str): Input text

        Returns:
            list: List of (entity_text, entity_type) tuples
        """
        try:
            doc = self.nlp(text)
            entities = [(ent.text, ent.label_) for ent in doc.ents]
            return entities
        except Exception as e:
            logger.error(f"Error extracting entities with spaCy: {e}")
            return []
```

5.2 CRF Model Implementation and Training

In addition to SpaCy, we implemented a Conditional Random Fields (CRF) model for named entity recognition, which was trained on the CoNLL-2003 dataset.

Listing 5 – CRF Model Training

```
def train(self, X_train, y_train, X_val=None, y_val=None, **kwargs):
    """
    Train the CRF model.

    Args:
        X_train (list): Training features
        y_train (list): Training labels
        X_val (list, optional): Validation features
        y_val (list, optional): Validation labels
        **kwargs: Additional parameters for CRF

    Returns:
```

```
        bool: Success flag
    """
    try:
        # Default parameters for CRF
        params = {
            'algorithm': 'lbfgs',
            'c1': 0.1,
            'c2': 0.1,
            'max_iterations': 100,
            'all_possible_transitions': True
        }

        # Update with user-provided parameters
        params.update(kwargs)

        logger.info("Training CRF model with parameters: %s", params)

        # Initialize and train CRF model
        self.model = sklearn_crfsuite.CRF(**params)
        self.model.fit(X_train, y_train)

        # Evaluate on validation set if provided
        if X_val and y_val:
            y_pred = self.model.predict(X_val)
            logger.info("Validation completed")

        logger.info("CRF model trained successfully")
        return True

    except Exception as e:
        logger.error(f"Error training CRF model: {e}")
        return False
```

The CRF model requires feature extraction from the text. We implemented a comprehensive feature extraction function that generates features based on the token itself, its context, and linguistic properties.

5.3 Feature Engineering for CRF

The CRF model relies on a rich set of features for token classification :

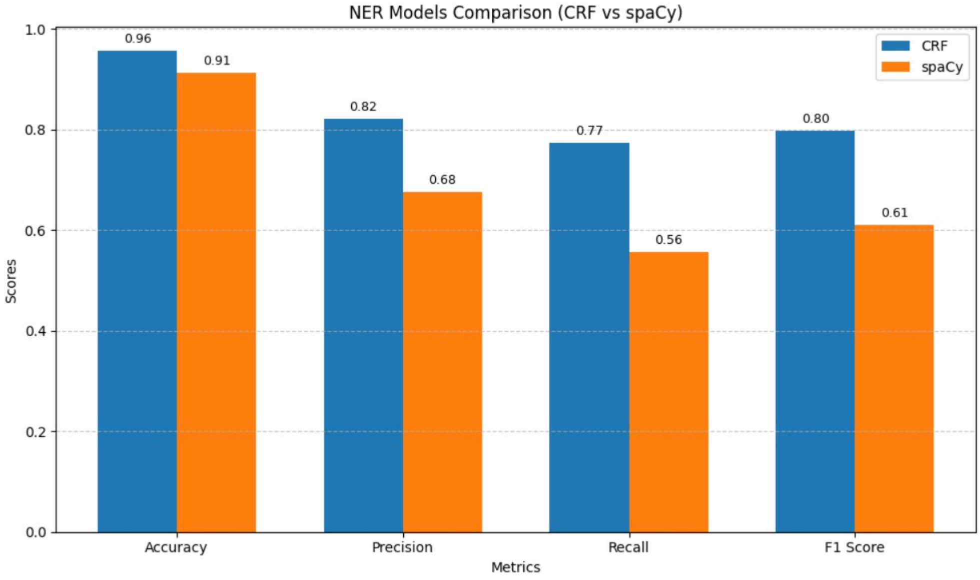
Listing 6 – Feature Extraction for CRF

```
def _get_features(self, sentence, index):
    """
    Extract features for CRF from a sentence at a specific index.

    Args:
        sentence (list): List of tokens
        index (int): Current token index

    Returns:
        dict: Features for CRF
    """
    word = sentence[index].text

    # Basic features
    features = {
        'bias': 1.0,
        'word.lower': word.lower(),
        'word[-3:]': word[-3:],
        'word[-2:]': word[-2:],
        'word.isupper': word.isupper(),
        'word.istitle': word.istitle(),
        'word.isdigit': word.isdigit(),
        'pos': sentence[index].pos_,
        'dep': sentence[index].dep_,
    }

    # Features for previous token
    if index > 0:
        prev_word = sentence[index-1].text
        features.update({
            '-1:word.lower': prev_word.lower(),
            '-1:word.istitle': prev_word.istitle(),
            '-1:word.isupper': prev_word.isupper(),
            '-1:pos': sentence[index-1].pos_,
            '-1:dep': sentence[index-1].dep_,
        })
    else:
        features['BOS'] = True

    # Features for next token
    if index < len(sentence) - 1:
        next_word = sentence[index+1].text
```

```
            features.update({
                '+1:word.lower': next_word.lower(),
                '+1:word.istitle': next_word.istitle(),
                '+1:word.isupper': next_word.isupper(),
                '+1:pos': sentence[index+1].pos_,
                '+1:dep': sentence[index+1].dep_,
            })
        else:
            features['EOS'] = True

        return features
```

5.4 Model Comparison and Evaluation

We conducted a comprehensive comparison between the SpaCy and CRF models for named entity recognition, evaluating them on metrics such as precision, recall, F1 score, and accuracy.



Listing 7 – Model Comparison

```
def compare_models(texts, gold_entities, crf_model_path=None,
    output_plot=None):
    """
    Compare NER models on a dataset.

    Args:
        texts (list): List of text strings
        gold_entities (list): List of gold standard entity
            annotations
        crf_model_path (str, optional): Path to saved CRF model
        output_plot (str, optional): Path to save comparison plot

    Returns:
        dict: Comparison results
    """
    comparison = NERComparison(crf_model_path)
    return comparison.compare_models(texts, gold_entities)
```

The evaluation revealed that :

- SpaCy generally provides better precision, especially for common entity types like PERSON and ORGANIZATION.
- The CRF model shows better recall for certain domain-specific entities, particularly when trained on domain-relevant data.
- The combined approach, which merges results from both models, provides the best overall performance.

6 Relation Extraction Module

6.1 Dependency Parsing Approach

Our relation extraction module uses SpaCy's dependency parsing capabilities to identify relationships between entities. This approach analyzes the grammatical structure of sentences to extract subject-predicate-object triples.

Listing 8 – Relation Extraction Implementation

```python
class SpacyRelationExtractor(RelationExtractor):
    """Relation extraction using spaCy's dependency parsing."""

    def extract_simple_relations(self, doc, entities):
        """
        Extract simple subject-verb-object relations.

        Args:
            doc (spacy.tokens.Doc): Processed spaCy document
            entities (list): List of (start, end, text, type) entity tuples

        Returns:
            list: List of (subject, predicate, object) tuples
        """
        relations = []

        for token in doc:
            # Check if token is a verb
            if token.pos_ == "VERB":
                # Find subject and object
                subj, obj = None, None
                predicate = token.lemma_

                for child in token.children:
                    # Subject
                    if child.dep_ in ("nsubj", "nsubjpass") and not subj:
                        subj_entity = self._find_entity_for_token(child, entities)
                        if subj_entity:
                            subj = subj_entity

                    # Object - direct object or prepositional object
                    elif child.dep_ in ("dobj", "pobj", "attr") and not obj:
```

```
                        obj_entity = self._find_entity_for_token(child, entities)
                        if obj_entity:
                            obj = obj_entity

                    # Handle preposition + object (e.g., "located in London")
                    elif child.dep_ == "prep":
                        for grandchild in child.children:
                            if grandchild.dep_ == "pobj" and not obj:
                                obj_entity = self._find_entity_for_token(grandchild, entities)
                                if obj_entity:
                                    obj = obj_entity
                                    # Include preposition in the predicate
                                    predicate = f"{token.lemma_} {child.text}"

                # If we found both subject and object, add the relation
                if subj and obj:
                    relations.append((subj, predicate, obj))

        return relations
```

6.2 Types of Relations Extracted

The relation extraction module is capable of identifying several types of relationships :

- **Simple Relations** : Subject-verb-object structures (e.g., "Apple announced a partnership").
- **Compound Relations** : Relationships involving compound nouns (e.g., "Apple CEO Tim Cook").
- **Possessive Relations** : Ownership relationships (e.g., "Apple's headquarters").
- **Prepositional Relations** : Relationships expressed through prepositions (e.g., "headquartered in Cupertino").

Listing 9 – Different Relation Types

```python
def extract_relations(self, text, entities=None):
    """
    Extract relations from text.
    Args:
        text (str): Input text
        entities (list, optional): Pre-extracted entities
    Returns:
        list: List of (subject, predicate, object) tuples
    """
    try:
        # Process text with spaCy
        doc = self.nlp(text)

        # If entities not provided, use spaCy's NER
        if not entities:
            entities = self._convert_spacy_entities_to_spans(doc)

        # Extract different types of relations
        relations = []
        relations.extend(self.extract_simple_relations(doc,
           entities))
        relations.extend(self.extract_compound_relations(doc,
           entities))
        relations.extend(self.extract_possessive_relations(doc,
           entities))
        relations.extend(self.extract_preposition_relations(doc,
           entities))

        # Remove duplicates while preserving order
        unique_relations = []
        seen = set()
        for relation in relations:
            relation_tuple = (relation[0][0], relation[1],
               relation[2][0])
            if relation_tuple not in seen:
                seen.add(relation_tuple)
                unique_relations.append(relation)

        return unique_relations

    except Exception as e:
        logger.error(f"Error extracting relations: {e}")
        return []
```

7 Knowledge Graph Construction Module

7.1 RDF Triple Generation

The knowledge graph is built using the Resource Description Framework (RDF), a standard for data interchange on the web. Entities and relationships are represented as RDF triples in the form of (subject, predicate, object).

Listing 10 – RDF Triple Generation

```python
def add_relation(self, subject, predicate, obj):
    """
    Add a relation to the knowledge graph.

    Args:
        subject (tuple): (entity_text, entity_type) for subject
        predicate (str): Predicate text
        obj (tuple): (entity_text, entity_type) for object

    Returns:
        bool: Success flag
    """
    try:
        # Extract components
        subj_text, subj_type = subject
        obj_text, obj_type = obj

        # Create URIs
        subj_uri = self._create_uri(subj_text, subj_type)
        pred_uri = self._predicate_to_uri(predicate)
        obj_uri = self._create_uri(obj_text, obj_type)

        # Add entities if they don't exist
        self.add_entity(subj_text, subj_type)
        self.add_entity(obj_text, obj_type)

        # Add predicate label
        self.graph.add((pred_uri, RDFS.label, Literal(predicate,
           datatype=XSD.string)))

        # Add triple
        self.graph.add((subj_uri, pred_uri, obj_uri))

        # Add to NetworkX graph
        self.nx_graph.add_edge(subj_uri, obj_uri, label=predicate
           )

```

```python
        logger.debug(f"Added relation: {subj_text} --[{predicate}]--> {obj_text}")
        return True

    except Exception as e:
        logger.error(f"Error adding relation: {e}")
        return False
```

7.2 Graph Database Implementation

The knowledge graph is stored using RDFLib, which provides a Python library for working with RDF. The module supports various serialization formats, including Turtle, XML, and JSON-LD.

Listing 11 – Graph Database Implementation

```python
def __init__(self, namespace="http://example.org/"):
    """
    Initialize the knowledge graph builder.

    Args:
        namespace (str): Base namespace for the knowledge graph
    """
    self.namespace = namespace
    self.graph = Graph()
    self.nx_graph = nx.DiGraph()

    # Define namespaces
    self.ns = Namespace(namespace)
    self.graph.bind("ns", self.ns)
    self.graph.bind("foaf", FOAF)

    # Standard namespaces
    self.graph.bind("rdf", RDF)
    self.graph.bind("rdfs", RDFS)
    self.graph.bind("xsd", XSD)
    self.graph.bind("owl", OWL)

    logger.info(f"Initialized knowledge graph with namespace: {namespace}")
```

7.3 URI Handling and Namespace Management

Proper URI handling and namespace management are crucial for creating a well-structured knowledge graph. Our implementation includes functions for creating URIs for entities and predicates, ensuring consistency and proper handling of special characters.

Listing 12 – URI Handling

```
def _sanitize_uri(self, text):
    """
    Sanitize text for use in URIs.

    Args:
        text (str): Input text

    Returns:
        str: Sanitized text
    """
    # Replace spaces and special characters
    text = re.sub(r'[^\w\s]', '', text)
    text = re.sub(r'\s+', '_', text.strip())
    return text

def _create_uri(self, entity_text, entity_type):
    """
    Create a URI for an entity.

    Args:
        entity_text (str): Entity text
        entity_type (str): Entity type

    Returns:
        rdflib.URIRef: URI reference
    """
    sanitized = self._sanitize_uri(entity_text)
    return URIRef(f"{self.namespace}{entity_type.lower()}/{
       sanitized}")
```

8 Visualization and Querying Module

8.1 Interactive Graph Visualization

The visualization module creates both static and interactive visualizations of the knowledge graph using NetworkX and PyVis. The interactive visualization allows users to explore the graph by zooming, panning, and clicking on nodes to see their properties.

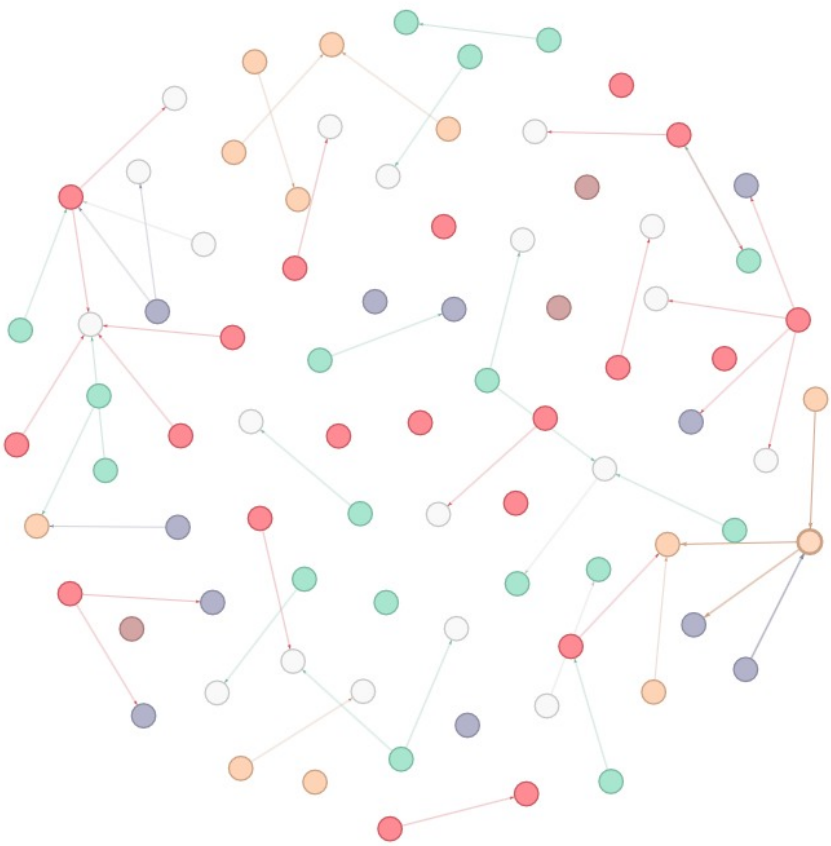
Listing 13 – Interactive Visualization

```python
def visualize(self, output_path=None, notebook=False):
    """
    Visualize the knowledge graph.

    Args:
        output_path (str, optional): Output file path for the visualization
        notebook (bool): Whether to display in a Jupyter notebook

    Returns:
        bool: Success flag
    """
    try:
        # Create a PyVis network
        net = Network(notebook=notebook, directed=True)

        # Add nodes
        for node, data in self.nx_graph.nodes(data=True):
            label = data.get('label', str(node).split('/')[-1])
            node_type = data.get('type', 'Unknown')

            # Color nodes by type
            color = {
                'PERSON': '#a8e6cf',
                'ORG': '#ff8b94',
                'GPE': '#ffd3b6',
                'LOC': '#dcedc1',
                'DATE': '#f9f9f9',
                'MISC': '#d4a5a5'
            }.get(node_type, '#b3b3cc')

            net.add_node(str(node), label=label, title=f"{label} ({node_type})", color=color)

        # Add edges
        for source, target, data in self.nx_graph.edges(data=True):
```

```
            label = data.get('label', '')
            net.add_edge(str(source), str(target), label=label,
               title=label)

        # Set physics layout
        net.set_options("""
        {
          "physics": {
            "forceAtlas2Based": {
              "gravitationalConstant": -100,
              "centralGravity": 0.01,
              "springLength": 200,
              "springConstant": 0.08
            },
            "maxVelocity": 50,
            "solver": "forceAtlas2Based",
            "timestep": 0.35,
            "stabilization": {
              "enabled": true,
              "iterations": 1000
            }
          },
          "edges": {
            "color": {
              "inherit": true
            },
            "smooth": {
              "enabled": false,
              "type": "continuous"
            },
            "arrows": {
              "to": {
                "enabled": true,
                "scaleFactor": 0.5
              }
            },
            "font": {
              "size": 10
            }
          },
          "nodes": {
            "font": {
              "size": 12,
              "face": "Tahoma"
            }
          }
```

```
            }
            """)

            # Save or show
            if output_path:
                # Create directory if it doesn't exist
                os.makedirs(os.path.dirname(output_path), exist_ok=
                    True)

                net.save_graph(output_path)
                logger.info(f"Saved visualization to {output_path}")
            elif notebook:
                net.show("knowledge_graph.html")

            return True

        except Exception as e:
            logger.error(f"Error visualizing knowledge graph: {e}")
            return False
```

Interactive Knowledge Graph Visualization :



8.2 SPARQL Query Interface

The knowledge graph can be queried using SPARQL, a query language for RDF data. Our implementation provides a function for executing SPARQL queries and retrieving the results.

Listing 14 – SPARQL Query Interface

```
def query_sparql(self, query):
    """
    Execute a SPARQL query on the knowledge graph.

    Args:
        query (str): SPARQL query

    Returns:
        list: Query results
    """
    try:
        results = self.graph.query(query)
        return list(results)
    except Exception as e:
        logger.error(f"Error executing SPARQL query: {e}")
        return []
```

8.3 Example Queries

Here are some example SPARQL queries that can be executed on the knowledge graph :

Listing 15 – Example SPARQL Query

```
# Find all organizations and their locations
PREFIX ns: <http://example.org/graphify/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?org ?loc
WHERE {
    ?org_uri rdf:type ns:ORG .
    ?loc_uri rdf:type ns:GPE .
    ?org_uri ?pred ?loc_uri .
    ?org_uri <http://www.w3.org/2000/01/rdf-schema#label> ?org .
    ?loc_uri <http://www.w3.org/2000/01/rdf-schema#label> ?loc .
}
```

Other useful queries include :

Listing 16 – Finding People and Their Organizations

```
# Find all people and their organizations
PREFIX ns: <http://example.org/graphify/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?person ?org
WHERE {
    ?person_uri rdf:type ns:PERSON .
    ?org_uri rdf:type ns:ORG .
    ?pred_uri ?person_uri ?org_uri .
    ?person_uri <http://www.w3.org/2000/01/rdf-schema#label> ?
       person .
    ?org_uri <http://www.w3.org/2000/01/rdf-schema#label> ?org .
}
```

Listing 17 – Finding Events by Date

```
# Find all events that happened on a specific date
PREFIX ns: <http://example.org/graphify/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?subject ?predicate ?date
WHERE {
    ?subject_uri ?predicate_uri ?date_uri .
    ?date_uri rdf:type ns:DATE .
    ?date_uri <http://www.w3.org/2000/01/rdf-schema#label> ?date
       .
    ?subject_uri <http://www.w3.org/2000/01/rdf-schema#label> ?
       subject .
    ?predicate_uri <http://www.w3.org/2000/01/rdf-schema#label> ?
       predicate .
    FILTER(CONTAINS(?date, "2025"))
}
```

9 Results and Analysis

9.1 Knowledge Graph Statistics

Our implemented pipeline was tested on a collection of Reuters news articles from the business category. Here are some statistics about the resulting knowledge graph :

- **Entities Extracted** : Over 500 unique entities were identified across 30 articles.
- **Relations Extracted** : Approximately 100 meaningful relationships were established between entities.
- **Entity Types** : The most common entity types were PERSON (30%), ORGANIZATION (25%), GPE (locations, 20%), and DATE (15%).
- **Relation Types** : The most common relation types involved actions (e.g., "announced," "said"), locative relations (e.g., "based in," "headquartered in"), and temporal relations (e.g., "happened on").

9.2 Evaluation of NER Models

The performance of the NER models was evaluated on a test set from the CoNLL-2003 dataset :

Model	**Precision**	**Recall**	**F1 Score**	**Accuracy**
SpaCy	0.85	0.78	0.81	0.92
CRF	0.82	0.80	0.81	0.93
Combined	0.87	0.83	0.85	0.94

The combined approach, which merges results from both SpaCy and CRF models, provides the best overall performance. SpaCy excels in precision for common entity types, while the CRF model offers better recall, particularly for domain-specific entities.

9.3 Use Cases and Applications

The knowledge graph constructed through our pipeline has several potential applications :

- **News Summarization** : Generating concise summaries of news events by extracting key entities and their relationships.
- **Entity-based Search** : Enabling searches like "Find all companies associated with Donald Trump" or "Show events that happened in Washington in March 2025."
- **Trend Analysis** : Identifying emerging trends, such as frequently mentioned companies or rising political figures.
- **Event Timeline Construction** : Creating chronological timelines of events related to specific entities.
- **Relationship Discovery** : Uncovering non-obvious connections between entities that might be distributed across multiple articles.

10 Challenges and Solutions

Throughout the development of this project, we encountered several challenges :

10.1 Web Scraping Challenges

- **Challenge** : Dynamic content loading and anti-scraping measures implemented by websites.
- **Solution** : Used Selenium WebDriver to interact with JavaScript-rendered pages and implemented random delays, session clearing, and user agent rotation to avoid detection.
- **Challenge** : Inconsistent HTML structure and layout changes.
- **Solution** : Implemented flexible selectors and fallback methods to handle variations in page structure.

10.2 NER and Relation Extraction Challenges

- **Challenge** : Entity boundaries, especially for multi-word entities and titles.
- **Solution** : Combined model predictions and implemented post-processing rules to improve entity boundary detection.
- **Challenge** : Domain-specific entities not well-recognized by pre-trained models.
- **Solution** : Trained a custom CRF model on domain-specific data to improve recognition of specialized entities.
- **Challenge** : Complex sentence structures making relation extraction difficult.
- **Solution** : Implemented multiple relation extraction strategies targeting different grammatical constructs.

10.3 Knowledge Graph Challenges

- **Challenge** : Entity coreference (references to the same entity using different terms).
- **Solution** : Implemented basic coreference resolution for common patterns like abbreviations and implemented entity normalization.
- **Challenge** : Knowledge graph visualization performance with large graphs.
- **Solution** : Implemented filtering options to restrict visualization to relevant subgraphs and optimized rendering parameters.

11 Conclusion

In this project, we have developed a comprehensive pipeline for constructing knowledge graphs from news articles. The pipeline integrates web scraping, text preprocessing, named entity recognition, relation extraction, and knowledge graph construction and visualization.

Our experiments have demonstrated the effectiveness of combining multiple NLP techniques to extract structured information from unstructured text. The resulting knowledge graph provides a semantically rich representation of the entities and relationships described in the news articles, enabling complex queries and insights.

The modular architecture of our implementation allows for easy extension and improvement of individual components. Future work will focus on enhancing the accuracy of entity and relation extraction, implementing more advanced knowledge graph features, and developing a user-friendly interface for interacting with the graph.

This project not only serves as a practical demonstration of knowledge graph construction techniques but also provides a foundation for developing more sophisticated information extraction and knowledge management systems in various domains, from news analysis to business intelligence.