Natural Language Processing using Python Programming

Notebook 04.2: Dependency Parsing with SpaCy

Python 3.8+ NLTK Latest SpaCy Latest License MIT

Part of the comprehensive learning series: Natural Language Processing using Python Programming

Learning Objectives:

- Master dependency parsing concepts and grammatical relationships
- Implement dependency analysis using SpaCy's advanced linguistic models
- Visualize syntactic trees with SpaCy's displacy module
- Extract structured information using Subject-Verb-Object (SVO) patterns
- Build foundation for advanced information extraction techniques

---

- **Dependency Parsing** is a method of analyzing the grammatical structure of a sentence by defining the relationships between words.
- It structures a sentence as a tree, where the nodes are the words and the directed edges represent the grammatical relationships (dependencies) between a **head** word and its **dependent** word.
- This is essential for deep language understanding, such as **Information Extraction** (e.g., finding the subject of an action).

1. Setting up: Libraries and Sample Text

- SpaCy is the standard tool for dependency parsing due to its speed and high-quality, pre-trained models.

In [1]:

```python
# Import necessary libraries
import spacy

# Load the full English model (parser included)
nlp = spacy.load('en_core_web_sm')

sample_sentence = "Apple is looking to buy a German startup for $100 million."

# Process the sentence
doc = nlp(sample_sentence)
print(f"Sample Sentence: {sample_sentence}")
```

```
Sample Sentence: Apple is looking to buy a German startup for $100 million.
```

2. Analyzing Dependencies Token by Token

- For every token in the SpaCy `Doc` object, we can access three key dependency attributes:

    1. **`.dep_`** : The typed dependency relation (e.g., `nsubj` , `dobj` , `amod`).
    2. **`.head.text`** : The word this token modifies or depends on (the **head**).
    3. **`.children`** : An iterator of the tokens that depend on this token.

- The root of the sentence (usually the main verb) has itself as its head.

In [2]:
```python
print("TOKEN      | DEPENDENCY TYPE | HEAD (Parent Word) | POS Tag")
print("-----------|-----------------|--------------------|---------")

for token in doc:
    print(f"{token.text:<10} | {token.dep_:<15} | {token.head.text:<18} | {token.
```

```
TOKEN      | DEPENDENCY TYPE | HEAD (Parent Word) | POS Tag
-----------|-----------------|--------------------|---------
Apple      | nsubj           | looking            | PROPN
is         | aux             | looking            | AUX
looking    | ROOT            | looking            | VERB
to         | aux             | buy                | PART
buy        | xcomp           | looking            | VERB
a          | det             | startup            | DET
German     | amod            | startup            | ADJ
startup    | dobj            | buy                | NOUN
for        | prep            | buy                | ADP
$          | quantmod        | million            | SYM
100        | compound        | million            | NUM
million    | pobj            | for                | NUM
.          | punct           | looking            | PUNCT
```

> **Observation:** The verb **`looking`** is the **ROOT**. `Apple` is the **`nsubj`** (nominal subject) of `looking` . `startup` is the **`dobj`** (direct object) of `buy` . This structure provides grammatical context.

Common Dependency Relations (Subsets of Universal Dependencies):

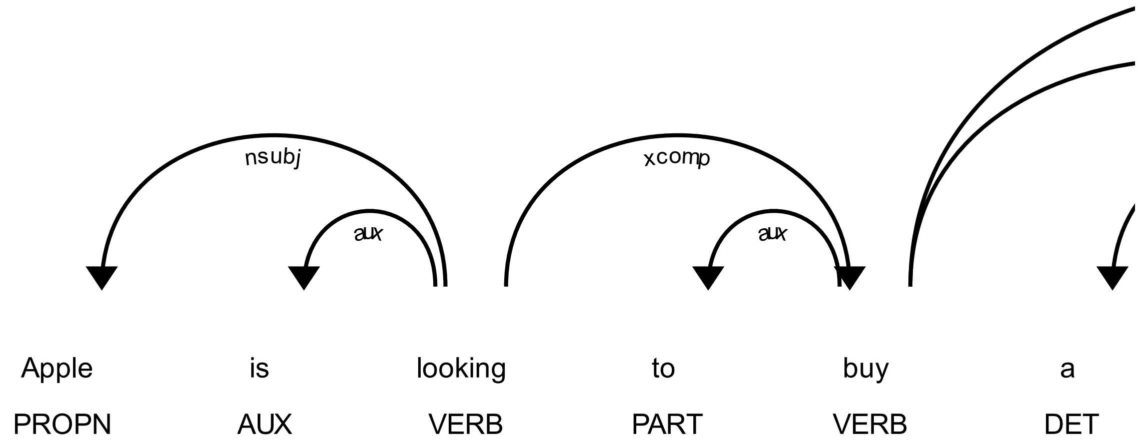
- **`nsubj`** : Nominal Subject (The agent of the verb).
- **`dobj`** : Direct Object (The recipient of the verb's action).
- **`amod`** : Adjectival Modifier (An adjective modifying a noun).
- **`attr`** : Attribute (The complement of a copular verb like 'is' or 'was').

3. Dependency Visualization with `displacy`

- SpaCy's built-in visualizer, **`displacy`**, is invaluable for understanding the dependency tree visually.

In [3]:
```
# Import displacy for visualization
from spacy import displacy

# Render the dependency tree inline in the notebook
displacy.render(doc, style="dep", jupyter=True, options={'distance': 100})
```



---

4. Practical Application: Simple Information Extraction

- We can use dependency parsing to extract simple **Subject-Verb-Object (SVO)** triplets from a sentence, which is a common task in Information Extraction (IE).

In [4]:
```
# Function to extract SVO triplets
def extract_svo(doc):
    """Extracts simple Subject-Verb-Object triplets from a SpaCy Doc."""
    triplets = []
    for token in doc:
        # Look for the main verb (ROOT) or primary clausal verbs
        if token.dep_ in ("ROOT", "advcl", "relcl") or token.pos_ == "VERB":
            subject = ""
            direct_object = ""

            # Find subject and direct object among the children
            for child in token.children:
                if child.dep_ == "nsubj": # Nominal Subject
                    subject = child.text
                elif child.dep_ == "dobj": # Direct Object
                    direct_object = child.text
```

```
            # Capture the full S-V-O if found
            if subject and direct_object:
                triplets.append((subject, token.text, direct_object))

    return triplets

ie_sentence = "Microsoft is designing a new cloud server, which analysts love."
ie_doc = nlp(ie_sentence)
extracted_triplets = extract_svo(ie_doc)

print(f"Sentence: {ie_sentence}")
print(f"Extracted S-V-O Triplet(s): {extracted_triplets}")
```

```
Sentence: Microsoft is designing a new cloud server, which analysts love.
Extracted S-V-O Triplet(s): [('Microsoft', 'designing', 'server'), ('analysts', 'l
ove', 'which')]
```

> **Observation:** The extraction successfully identifies: `('Microsoft', 'designing', 'server')`. This shows how structural analysis is directly used to pull information.

5. Summary and Next Steps

- Dependency parsing is the foundation of high-level language understanding.
- By understanding the head-dependent relationships, we gain syntactic context far beyond simple word lists.
- With Chapters 1-4 complete, we have covered all the fundamental preprocessing and linguistic analysis steps.
- In **Chapter 5**, we will move to the high-value task of **Named Entity Recognition (NER)**, which often relies on POS tags and dependency relations for maximum accuracy.

Key Takeaways

- **Dependency Parsing Mastery:** We successfully implemented dependency parsing using SpaCy, learning to analyze grammatical relationships between words in sentences.
- **Syntactic Tree Understanding:** We mastered the concept of head-dependent relationships and how sentences form hierarchical grammatical structures.
- **Visualization Skills:** We utilized SpaCy's displacy module to create clear visual representations of dependency trees for better understanding.
- **Information Extraction Foundation:** We implemented practical Subject-Verb-Object (SVO) extraction, demonstrating how syntactic analysis enables structured information retrieval.

Next Notebook Preview

- With dependency parsing mastered, we're ready to explore **advanced linguistic analysis**.
- The next notebook will dive into **Named Entity Recognition (NER)**, which leverages POS tags and dependency relations for accurate entity identification and classification.

About This Project

This notebook is part of the **Natural Language Processing using Python Programming for Beginners** repository - a comprehensive, beginner-friendly guide for mastering NLP using Python, NLTK, and SpaCy.

Repository: `NLP`

Author

Prakash Ukhalkar
GitHub prakash-ukhalkar

Built with ❤️ for the Python community