Natural Language Processing using Python Programming

Notebook 05.2: Named Entity Recognition (NER) 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 production-ready NER using SpaCy's state-of-the-art models
- Implement high-accuracy entity extraction with pre-trained models
- Visualize entity recognition results using SpaCy's displacy module
- Create custom entity patterns with SpaCy's rule-based Matcher
- Build foundation for domain-specific entity recognition systems

---

- While NLTK provides a conceptual introduction to NER, **SpaCy** is the industry standard for production-ready NER due to its high accuracy, speed, and integrated architecture.
- This notebook focuses on harnessing SpaCy's power and exploring methods for customizing it for domain-specific tasks.

1. Using SpaCy's Pre-trained NER Model

- SpaCy's large English models (`en_core_web_lg` or `md`) recognize a wide range of entity types (typically 18).
- We'll use the small model (`sm`) for speed, which still provides robust results.

In [1]:
```
# Import necessary libraries
import spacy
from spacy import displacy # for visualizing entities

# Load the small English model
nlp = spacy.load('en_core_web_sm')

sample_text = "The research team at Google DeepMind announced a breakthrough in Lo

# Process the text
doc = nlp(sample_text)
print(f"Sample Text: {sample_text}")
```

```
Sample Text: The research team at Google DeepMind announced a breakthrough in Londo
n on October 5, 2025. The project cost $50 million.
```

1.1 Extracting Entities Programmatically

- Entities are stored in the `doc.ents` property, which is a tuple of `Span` objects.
- Each `Span` has text and an entity label (`.label_`).

In [2]:
```python
print("Extracted Entities (SpaCy):")
print("ENTITY TEXT       | LABEL (TYPE)")
print("------------------|--------------")
for ent in doc.ents:
    print(f"{ent.text:<18}| {ent.label_}")
```

```
Extracted Entities (SpaCy):
ENTITY TEXT       | LABEL (TYPE)
------------------|--------------
Google DeepMind   | ORG
London            | GPE
October 5, 2025   | DATE
$50 million       | MONEY
```

> **Observation:** SpaCy correctly groups multi-word entities (like `Google DeepMind`) and accurately labels types like `ORG` , `GPE` , `DATE` , and `MONEY` .

1.2 Visualizing Entities with `displacy`

- Visualization is key for quickly verifying NER output and presenting results.

In [3]:
```python
# Render the entities in the notebook
displacy.render(doc, style="ent", jupyter=True)
```

The research team at Google DeepMind **ORG** announced a breakthrough in London **GPE** on October 5, 2025 **DATE** . The project cost $50 million **MONEY** .

---

2. SpaCy's Entity Labels (Quick Reference)

- The small model recognizes these common labels.
- You can look up the full definition using `spacy.explain()` :

In [4]:
```python
print("Explanation for GPE:")
print(spacy.explain('GPE'))

print("\nExplanation for MONEY:")
print(spacy.explain('MONEY'))
```

```
Explanation for GPE:
Countries, cities, states

Explanation for MONEY:
Monetary values, including unit
```

---

3. Customizing NER: The SpaCy `Matcher` (Rule-Based)

- Pre-trained models fail on domain-specific entities (e.g., product codes, proprietary job titles).
- We can use SpaCy's **Rule-Based Matcher** to define custom patterns.

Example 1: Identifying Custom Product Codes

- Imagine we need to identify internal product codes that follow the pattern: `[Capital Letter]-[Digit][Digit][Digit]` (e.g., `P-304`, `Z-999`).

In [5]:
```python
# Custom NER with SpaCy's Matcher
from spacy.matcher import Matcher

text_with_code = "We need to process orders for product P-304 and R-007 immediatel
doc_custom = nlp(text_with_code)

matcher = Matcher(nlp.vocab)

# Define the pattern for a custom product code:
# SpaCy tokenizes "P-304" as a single token with shape "X-ddd"
# So we need to match against the complete token pattern
pattern = [
    {"SHAPE": "X-ddd"}  # Matches single tokens like "P-304", "R-007"
]

matcher.add("PRODUCT_CODE", [pattern])

# Apply the matcher to the document
matches = matcher(doc_custom)

# Note: SpaCy tokenizes "P-304" as a single token, not as separate "P", "-", "304'

print(f"Text to analyze: {text_with_code}")
print("\nFound Custom Matches:")
for match_id, start, end in matches:
    span = doc_custom[start:end]
    print(f" - Entity: {span.text:<10} | Start: {start} | End: {end}")
```

```
Text to analyze: We need to process orders for product P-304 and R-007 immediately.

Found Custom Matches:
 - Entity: P-304      | Start: 7 | End: 8
 - Entity: R-007      | Start: 9 | End: 10
```

Example 2: Multi-Token Custom Patterns

- Let's create a more complex example that matches patterns spanning **multiple tokens**.
- We'll identify email addresses and phone number patterns that SpaCy tokenizes as separate tokens.

In [6]:

```
# Multi-token pattern matching example
text_complex = "Contact John Smith at john.smith@company.com or call (555) 123-456
doc_multi = nlp(text_complex)

# First, let's see how SpaCy tokenizes this text
print("Tokenization Analysis:")
for i, token in enumerate(doc_multi):
    print(f"Token {i}: '{token.text}' | Shape: {token.shape_}")

print(f"\nText to analyze: {text_complex}")
```

```
Tokenization Analysis:
Token 0: 'Contact' | Shape: Xxxxx
Token 1: 'John' | Shape: Xxxx
Token 2: 'Smith' | Shape: Xxxxx
Token 3: 'at' | Shape: xx
Token 4: 'john.smith@company.com' | Shape: xxxx.xxxx@xxxx.xxx
Token 5: 'or' | Shape: xx
Token 6: 'call' | Shape: xxxx
Token 7: '(' | Shape: (
Token 8: '555' | Shape: ddd
Token 9: ')' | Shape: )
Token 10: '123' | Shape: ddd
Token 11: '-' | Shape: -
Token 12: '4567' | Shape: dddd
Token 13: 'for' | Shape: xxx
Token 14: 'more' | Shape: xxxx
Token 15: 'details' | Shape: xxxx
Token 16: '.' | Shape: .

Text to analyze: Contact John Smith at john.smith@company.com or call (555) 123-456
7 for more details.
```

In [7]:

```
# Create a new matcher for multiple patterns
matcher_multi = Matcher(nlp.vocab)

# Pattern 1: Email addresses (4 tokens: name.name@domain.com)
email_pattern = [
    {"LIKE_EMAIL": True}  # SpaCy's built-in email detection
]

# Pattern 2: Phone numbers like (555) 123-4567 (6 tokens: ( 555 ) 123 - 4567)
phone_pattern = [
    {"TEXT": "("},                      # (
    {"SHAPE": "ddd"},                   # 555
    {"TEXT": ")"},                      # )
    {"SHAPE": "ddd"},                   # 123
    {"TEXT": "-"},                      # -
    {"SHAPE": "dddd"}                   # 4567
]

# Pattern 3: Full names (2 tokens: First Last) - excludes "Contact"
```

```
name_pattern = [
    {"POS": "PROPN", "IS_TITLE": True, "TEXT": {"NOT_IN": ["Contact"]}},  # First
    {"POS": "PROPN", "IS_TITLE": True}   # Last name
]

# Add all patterns to matcher
matcher_multi.add("EMAIL", [email_pattern])
matcher_multi.add("PHONE", [phone_pattern])
matcher_multi.add("FULL_NAME", [name_pattern])

# Apply matcher
matches_multi = matcher_multi(doc_multi)

print(f"\nText to analyze: {text_complex}")
print(f"\nNumber of multi-token matches found: {len(matches_multi)}")
print("\nFound Multi-Token Custom Matches:")
for match_id, start, end in matches_multi:
    span = doc_multi[start:end]
    label = nlp.vocab.strings[match_id]  # Convert match_id back to string
    print(f" - Type: {label:<10} | Entity: '{span.text:<22}' | Tokens: {end-start
```

```
Text to analyze: Contact John Smith at john.smith@company.com or call (555) 123-456
7 for more details.

Number of multi-token matches found: 3

Found Multi-Token Custom Matches:
 - Type: FULL_NAME  | Entity: 'John Smith            ' | Tokens: 2 | Start: 1 | En
d: 3
 - Type: EMAIL      | Entity: 'john.smith@company.com' | Tokens: 1 | Start: 4 | En
d: 5
 - Type: PHONE      | Entity: '(555) 123-4567        ' | Tokens: 6 | Start: 7 | En
d: 13
```

Key Observations from Multi-Token Matching:

- **Email Pattern**: Uses SpaCy's built-in `LIKE_EMAIL` attribute (1 token - SpaCy keeps emails together)
- **Phone Pattern**: Matches exactly 6 tokens: `(`, `555`, `)`, `123`, `-`, `4567`
- **Name Pattern**: Identifies 2 consecutive proper nouns, excluding common words like "Contact"
- **Token Count**: Shows how many tokens each match spans (crucial for understanding SpaCy's tokenization)

Important: Always analyze tokenization first! SpaCy might split text differently than you expect, which affects pattern design.

Advanced Tips: Beyond Rule-Based Matching

1. Pattern Complexity Considerations

- **Single-token patterns** (like `P-304`) are fast and reliable when SpaCy tokenizes entities as expected

- **Multi-token patterns** (like phone numbers) require careful alignment with SpaCy's tokenization behavior

- **Always test tokenization first** using `[token.text for token in doc]` before designing patterns

2. Training Custom NER Models (Conceptual)

- For maximum accuracy on completely new entity types (e.g., proprietary legal document tags), you must provide hundreds or thousands of **labeled examples** and fine-tune the SpaCy model itself.
- This is often done using the BILOU scheme (Chapter 5.1) and involves:
  - Collecting annotated training data
  - Using SpaCy's training pipeline
  - Iterative model refinement and evaluation
- The concept is to **teach** the model new patterns through machine learning, not just define rules.

4. Summary and Next Steps

- **Production NER**: Implemented SpaCy's high-accuracy, pre-trained models for standard entity recognition
- **Entity Visualization**: Used `displacy` for clear, interactive entity displays
- **Single-Token Matching**: Created patterns for entities tokenized as single units (product codes)
- **Multi-Token Matching**: Built complex patterns spanning multiple tokens (emails, phones, names)
- **Pattern Design**: Learned to analyze tokenization behavior before designing custom patterns

Custom Entity Strategy:

For domain-specific entities, we demonstrated two approaches:

1. **Rule-Based Matching** (SpaCy Matcher) - Fast, effective for well-defined patterns
2. **Machine Learning Training** (Conceptual) - For complex, nuanced entity types requiring labeled data

Course Transition:

We have now completed the **linguistic analysis foundation** of NLP (Chapters 1-5):

- Text preprocessing and tokenization
- Part-of-speech tagging and dependency parsing
- Named entity recognition and custom pattern matching

Next: We transition to the **machine learning core** where we bridge human language and algorithms.

In **Chapter 6**, we will learn **Text Vectorization** - converting text into numerical representations that machine learning models can process.

Key Takeaways

- **Production NER Mastery:** We successfully implemented industry-standard Named Entity Recognition using SpaCy's high-accuracy, pre-trained models.
- **Entity Visualization:** We mastered SpaCy's displacy module for creating clear, interactive visualizations of entity recognition results.
- **Custom Entity Recognition:** We implemented comprehensive rule-based entity matching using SpaCy's Matcher, covering both single-token and multi-token patterns for domain-specific entity extraction.
- **Pattern Design Mastery:** We learned to analyze SpaCy's tokenization behavior and design patterns accordingly, from simple product codes to complex phone numbers and email addresses.
- **Advanced NER Understanding:** We explored the conceptual foundation for training custom NER models with labeled examples and BILOU tagging.

Next Notebook Preview

- With linguistic analysis mastered (Chapters 1-5), we're ready to bridge the gap between human language and machine learning.
- The next notebook will dive into **Text Vectorization**, converting text into numerical representations that machine learning algorithms can process.

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