Natural Language Processing using Python Programming

Notebook 02.2: Tokenization with NLTK and 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 word-level and sentence-level tokenization techniques
- Compare NLTK and SpaCy tokenization approaches and capabilities
- Handle complex text cases like contractions, abbreviations, and punctuation
- Implement custom tokenization rules for domain-specific requirements
- Build robust tokenization pipelines for real-world text processing

---

- **Tokenization** is the initial, mandatory step in the NLP pipeline where a text sequence is broken down into smaller components called **tokens**.

- In this notebook, we'll master two main forms—word and sentence tokenization—and explore how NLTK and SpaCy handle complex, real-world text.

1. Setting up: Libraries and Sample Text

- We use a slightly complex sample text to test how well each library handles punctuation, contractions, and abbreviations.

In [1]:
```python
# Import necessary libraries
import nltk
import spacy

# Ensure NLTK's punkt (tokenizer model) is downloaded
nltk.download('punkt', quiet=True)

# Load SpaCy model
nlp = spacy.load('en_core_web_sm')
```

In [2]:
```python
sample_text = "The U.S. doesn't have a perfect solution. Dr. Smith said, 'They're
print(f"Sample Text:\n{sample_text}")
```

```
Sample Text:
The U.S. doesn't have a perfect solution. Dr. Smith said, 'They're looking at a $1.
2M investment.' What's next?
```

---

2. Word-Level Tokenization

- This is the most common form of tokenization, where each word and punctuation mark becomes a separate token.

2.1 Word Tokenization with NLTK

- NLTK's `word_tokenize` is a standard, rule-based tokenizer.
- It often splits contractions into two or three separate tokens.

In [3]:
```python
# Import word_tokenize from NLTK
# Contraction handling means "do", "n't" are split
from nltk.tokenize import word_tokenize

nltk_tokens = word_tokenize(sample_text)

print(f"Total Tokens (NLTK): {len(nltk_tokens)}")
print("Tokens:")
print(nltk_tokens)
```

```
Total Tokens (NLTK): 27
Tokens:
['The', 'U.S.', 'does', "n't", 'have', 'a', 'perfect', 'solution', '.', 'Dr.', 'Smi
th', 'said', ',', "'They", "'re", 'looking', 'at', 'a', '$', '1.2M', 'investment',
'.', "'", 'What', "'s", 'next', '?']
```

> **Observation:** NLTK successfully handles `doesn't` (*ightarrow* `does`, `n't`), but note how abbreviations like `U.S.` are tokenized as `U.S.`. Also, the single quote in `They're` is split.

2.2 Word Tokenization with SpaCy

- SpaCy's tokenizer is non-destructive and is built with a statistical model and specific language rules (like contraction handling) in mind.
- It's often more accurate for production.

In [6]:
```python
# Processing text with SpaCy creates a Doc object
# document object handles contractions as single tokens and punctuation properly
doc = nlp(sample_text)

# Extract tokens from the Doc object
spacy_tokens = [token.text for token in doc]

print(f"Total Tokens (SpaCy): {len(spacy_tokens)}")
print("Tokens:")
print(spacy_tokens)
```

```
Total Tokens (SpaCy): 29
Tokens:
['The', 'U.S.', 'does', "n't", 'have', 'a', 'perfect', 'solution', '.', 'Dr.', 'Smi
th', 'said', ',', "'", 'They', "'re", 'looking', 'at', 'a', '$', '1.2', 'M', 'inves
tment', '.', "'", 'What', "'s", 'next', '?']
```

> **Observation:** SpaCy keeps `U.S.` as a single token, which preserves the abbreviation's meaning. It correctly separates the `$` symbol from the number `1.2` and handles contractions like `doesn't` by splitting into `does` and `n't`. SpaCy's approach is generally more linguistically informed.

---

3. Sentence-Level Tokenization

- **Sentence Tokenization** (or Sentence Segmentation) is vital for tasks like text summarization, machine translation, and question answering.
- It's challenging because a period (`.`) can mark the end of a sentence OR an abbreviation (e.g., `Dr.`, `U.S.`).

3.1 Sentence Tokenization with NLTK

- NLTK uses a pre-trained model (trained on the *Punkt* corpus) to recognize sentence boundaries.

In [7]:
```python
# Import sent_tokenize from NLTK
from nltk.tokenize import sent_tokenize

nltk_sentences = sent_tokenize(sample_text)

print(f"Total Sentences (NLTK): {len(nltk_sentences)}")
for i, sent in enumerate(nltk_sentences):
    print(f"Sentence {i+1}: {sent}")
```

```
Total Sentences (NLTK): 3
Sentence 1: The U.S. doesn't have a perfect solution.
Sentence 2: Dr. Smith said, 'They're looking at a $1.2M investment.'
Sentence 3: What's next?
```

> **Observation:** NLTK correctly identifies three sentences, managing the `Dr.` abbreviation and the complex punctuation. It's highly effective for sentence segmentation.

3.2 Sentence Tokenization with SpaCy

- In SpaCy, sentence boundaries are marked by the dependency parser and are accessible via the `doc.sents` iterator.

In [8]:
```python
# Processing text with SpaCy creates a Doc object
# document object handles sentence segmentation
# doc.sents is an iterator of sentence spans
spacy_sentences = list(doc.sents)

print(f"Total Sentences (SpaCy): {len(spacy_sentences)}")
```

```
for i, sent in enumerate(spacy_sentences):
    print(f"Sentence {i+1}: {sent.text}")
```

```
Total Sentences (SpaCy): 3
Sentence 1: The U.S. doesn't have a perfect solution.
Sentence 2: Dr. Smith said, 'They're looking at a $1.2M investment.'
Sentence 3: What's next?
```

> **Observation:** SpaCy also correctly identifies the three sentences. Its segmentation relies on its powerful pipeline (POS, dependencies), which makes it very robust to complex syntax.

---

4. Handling Edge Cases: Custom Tokenization (Conceptual)

- Sometimes, standard tokenizers don't work for specific domains (e.g., gene names in biology, product codes in e-commerce).
- Both NLTK and SpaCy allow for customization.

4.1 Customizing NLTK with RegexpTokenizer

- NLTK allows for simple, rule-based tokenizers using regular expressions.
- This gives you absolute control, though it requires knowledge of regex.

In [9]:
```
# Import RegexpTokenizer from NLTK for custom tokenization
# Example: Tokenizer that only captures words (alphanumeric sequences)
from nltk.tokenize import RegexpTokenizer

# Tokenizer that only finds words (alphanumeric sequences)
tokenizer = RegexpTokenizer(r'\w+')
custom_tokens = tokenizer.tokenize(sample_text)

print(f"Regex Tokens (only words):\n{custom_tokens}")

# Note: All punctuation, including the dollar sign, is stripped out.
```

```
Regex Tokens (only words):
['The', 'U', 'S', 'doesn', 't', 'have', 'a', 'perfect', 'solution', 'Dr', 'Smith',
'said', 'They', 're', 'looking', 'at', 'a', '1', '2M', 'investment', 'What', 's',
'next']
```

4.2 Understanding SpaCy's Default Tokenization Behavior

- SpaCy's tokenizer has sophisticated built-in rules for handling different types of hyphenated expressions.
- Some hyphens are kept together (like product codes), while others are split for linguistic accuracy.

In [12]:
```python
# Import re for regex operations
import re

# Create a simple example showing the concept of custom tokenization
# Since modifying SpaCy's tokenizer is complex, we'll show a conceptual difference
# nlp() creates a Doc object with default tokenization rules

# Default SpaCy tokenization
text_product = "The new product, X-3000, is fantastic."
doc_default = nlp(text_product)

# For demonstration, let's show what happens with a hyphenated word that SpaCy doe
text_with_hyphen = "The state-of-the-art technology is amazing."
doc_hyphen = nlp(text_with_hyphen)

print("Default Tokens (text with X-3000):", [t.text for t in doc_default])
print("Default Tokens (splits 'state-of-the-art'):", [t.text for t in doc_hyphen])

# Note: SpaCy keeps 'X-3000' as one token by default (which is good!)
# But it splits 'state-of-the-art' into multiple tokens
print("\nObservation: SpaCy's default tokenizer already handles 'X-3000' well,")
print("but splits compound words like 'state-of-the-art' for linguistic accuracy.'
```

```
Default Tokens (text with X-3000): ['The', 'new', 'product', ',', 'X-3000', ',', 'i
s', 'fantastic', '.']
Default Tokens (splits 'state-of-the-art'): ['The', 'state', '-', 'of', '-', 'the',
'-', 'art', 'technology', 'is', 'amazing', '.']

Observation: SpaCy's default tokenizer already handles 'X-3000' well,
but splits compound words like 'state-of-the-art' for linguistic accuracy.
```

5. Summary and Next Steps

- Tokenization is more nuanced than simple splitting by whitespace.
- Both NLTK and SpaCy offer robust solutions, with SpaCy generally providing better accuracy due to its integrated model.
- We have now covered the entire **Text Preprocessing** stage (Normalization and Tokenization).
- In **Chapter 3**, we will apply these skills to large, real-world text collections (Corpora and Datasets) to begin exploring language patterns.

Key Takeaways

- **Tokenization Mastery:** We explored both word-level and sentence-level tokenization using NLTK and SpaCy, understanding their strengths and differences.
- **Complex Text Handling:** We learned how both libraries manage challenging cases like contractions, abbreviations, and special punctuation marks.
- **Customization Capabilities:** We discovered how to create custom tokenization rules for domain-specific requirements using RegexpTokenizer and SpaCy's flexible architecture.

Next Notebook Preview

- With preprocessing and tokenization mastered, we're ready to work with **real-world text collections and corpora**.
- The next chapter will explore **loading and analyzing large text datasets**, applying our preprocessing skills to discover language patterns and insights.

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