Natural Language Processing

An Introduction

Abdelbacet Mhamdi

2025-04-14

ISET Bizerte

Outline

1. Introduction to Regular Expressions (Regex)

1.1 What are Regular Expressions?

Regular expressions are powerful patterns used to match, search, and manipulate text strings. They provide a standardized way to describe search patterns in text, making them an essential tool in programming, text processing, and data validation.

1.2 Core Concepts

1.2.1 Pattern Matching

A regex pattern is a sequence of characters that defines a search pattern. These patterns can be:

- Literal characters that match themselves;
- Special characters (metacharacters) with special meanings;
- Combinations of both.

1.2 Core Concepts

1.2.2 Basic Metacharacters

Metacharacter	Description	Example
`.`	Matches any character except newline	`a.c` matches “abc”, “a1c”, “a@c”
`^`	Matches start of string	`^Hello` matches “Hello World”
`$`	Matches end of string	`world$` matches “Hello world”
`*`	Matches 0 or more occurrences	`ab*c` matches “ac”, “abc”, “abbc”
`+`	Matches 1 or more occurrences	`ab+c` matches “abc”, “abbc” but not “ac”
`?`	Matches 0 or 1 occurrence	`ab?c` matches “ac” and “abc”
`\`	Escapes special characters	`\.` matches literal dot

1.3 Common Use Cases

1. **Search Operations**
   - Advanced find/replace operations;
   - Pattern matching in large text files;
   - Content filtering.
2. **Text Processing**
   - Finding patterns in text;
   - Replacing specific text patterns;
   - Extracting information;
   - Parsing log files.
3. **Data Validation**
   - Email addresses;
   - Phone numbers;
   - Postal codes;
   - Passwords;
   - URLs.

1.4.1 Character Classes

```python
# Character class examples
pattern = r'[aeiou]'  # Matches any vowel
pattern = r'[0-9]'    # Matches any digit
pattern = r'[^0-9]'   # Matches any non-digit
```

1.4.2 Quantifiers and Groups

Python

```python
# Quantifiers
pattern = r'\d{3}'      # Exactly 3 digits
pattern = r'\d{2,4}'    # Between 2 and 4 digits
pattern = r'\d{2,}'     # 2 or more digits

# Groups
pattern = r'(\w+)\s+\1'  # Matches repeated words
```

1.4.3 Common Regex Functions in Python

Python

```python
import re

text = "The price is $19.99"

# Different matching functions
re.search(r'\$\d+\.\d+', text)   # Finds first match
re.findall(r'\$\d+\.\d+', text)  # Finds all matches
re.sub(r'\$(\d+\.\d+)', r'\1', text)  # Substitution

# Splitting text
re.split(r'\s+', text)  # Split on whitespace
```

1.5.1 Basic Pattern Matching

Python

```python
import re
# Simple pattern matching
text = "The quick brown fox jumps over the lazy dog"
pattern = r"fox"
# Search for pattern
match = re.search(pattern, text)
if match:
    print(f"Found '{pattern}' at position: {match.start()}-
    {match.end()}")
# Find all occurrences
words = re.findall(r"\w+", text)
print(f"All words: {words}")
```

1.5.2 Email Validation Example

Python

```python
def is_valid_email(email):
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return bool(re.match(pattern, email))
# Test cases
emails = [
    "user@example.com", # ✔
    "invalid.email@com", # ✗
    "user.name@bizerte.r-iset.tn", # ✔
    "@invalid.com" # ✗
]
for email in emails:
    print(f"{email} → {'Valid' if is_valid_email(email) else 'Invalid'}")
```

1.5 Python Implementation

1.5.3 Phone Number Formatting

Python

```python
def format_phone_number(phone):
    # Remove all non-digit characters
    digits = re.sub(r'\D', '', phone)

    # Format as (XXX) XXX-XXXX
    if len(digits) == 10:
        pattern = r'(\d{3})(\d{3})(\d{4})'
        formatted = re.sub(pattern, r'(\1) \2-\3', digits)
        return formatted

    return "Invalid phone number"
```

1.5 Python Implementation

Python

```python
# Test cases
numbers = [
        "1234567890",
        "123-456-7890",
        "(123) 456-7890",
        "12345"
]

for number in numbers:
        print(f"{number} → {format_phone_number(number)}")
```

1.6 Best Practices

1. **Use Raw Strings**
   - Always prefix regex patterns with `r` to avoid escape character issues

```python
pattern = r'\d+'  # Better than '\d+'
```

2. **Compile Frequently Used Patterns**

```python
email_pattern = re.compile(r'^[\w\.-]+@[\w\.-]+\.\w+$')
# Use multiple times
email_pattern.match(email1)
email_pattern.match(email2)
```

3. **Be Specific**
   - Make patterns as specific as possible to avoid false matches;
   - Use start (^) and end ($) anchors when matching whole strings.

1.6 Best Practices

4. **Test Thoroughly**
   - Test with both valid and invalid inputs
   - Include edge cases in your tests

Python

```python
def test_pattern(pattern, test_cases):
    regex = re.compile(pattern)
    for test, expected in test_cases:
        result = bool(regex.match(test))
        print(f"'{test}': {'✓' if result == expected else '✗'}")
```

1.7 Common Pitfalls

“Some people, when confronted with a problem, think ‘I know, I’ll use regular expressions.‘ Now they have two problems.“ - Jamie Zawinski

1. **Greedy vs. Non-Greedy Matching**

```python
# Greedy (default)
re.findall(r'<.*>', '<tag>text</tag>')  # ['<tag>text</tag>']

# Non-greedy: Add (lazy) `?`
re.findall(r'<.*?>', '<tag>text</tag>')  # ['<tag>', '</tag>']
```

2. **Performance Considerations**
   - Avoid excessive backtracking *(recursion)*;
   - Be careful with nested quantifiers;
   - Use more specific patterns when possible.

1.8 Applications

1. **Basic Pattern Matching**

Python

```python
# Write a pattern to match dates in format DD/MM/YYYY
date_pattern = r'\d{2}/\d{2}/\d{4}'
```

2. **Data Extraction**

Python

```python
# Extract all email addresses from text
text = "Contact us at support@example.com or sales@example.com"
emails = re.findall(r'[\w\.-]+@[\w\.-]+\.\w+', text)
```

1.8 Applications

3. **Password Validation**

```python
def is_strong_password(password):
    # At least 8 chars, 1 upper, 1 lower, 1 digit, 1 special
    pattern = r'^(?=.*[A-Z])(?=.*[a-z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$' # Positive Lookahead
    return bool(re.match(pattern, password))
```

1.8 Applications

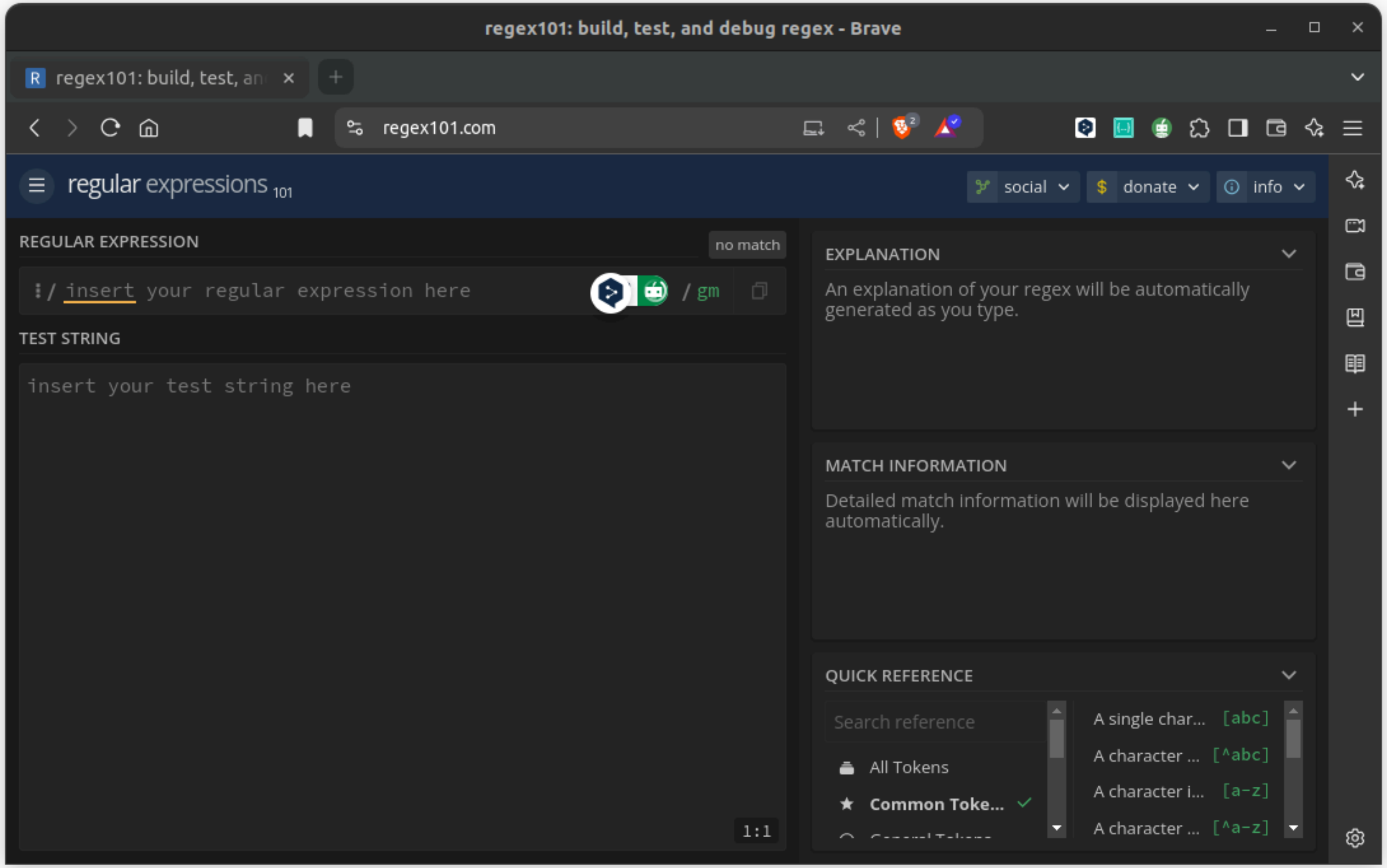

regex101: build, test, and debug regex - Brave
regex101: build, test, an
regex101.com
regular expressions 101
social
donate
info
REGULAR EXPRESSION
no match
/ insert your regular expression here / gm
TEST STRING
insert your test string here
1:1
EXPLANATION
An explanation of your regex will be automatically generated as you type.
MATCH INFORMATION
Detailed match information will be displayed here automatically.
QUICK REFERENCE
Search reference
All Tokens
Common Toke...
A single char... [abc]
A character ... [^abc]
A character i... [a-z]
A character ... [^a-z]

Figure 1: Build, test and debug regex patterns.

1.8 Applications

Task 1:

Write a regex pattern to match all occurrences of the word “python” (case-insensitive) in a string.

```
pattern = re.compile(r“python”, re.IGNORECASE)
```

1.8 Applications

Task 2:

Write a regex pattern to validate email addresses.

```
pattern = re.compile(r“^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+.[a-zA-Z]{2,}$")
```

1.8 Applications

Task 3:

Extract all phone numbers from a text where numbers can be in format XXX-XXX-XXXX or (XXX) XXX-XXXX.

```
pattern = re.compile(r"(\d{3}-\d{3}-\d{4}|\(\d{3}\)\s\d{3}-\d{4})")
```

1.8 Applications

Task 4:

Extract the username and domain from email addresses.

```
pattern = re.compile(r“^([a-zA-Z0-9._%+-]+)@([a-zA-Z0-9.-]+\.[a-zA-Z]{2,})$”)
```

1.8 Applications

Task 5:

Match whole words “code” and “coding” but not words that contain them like “encoder” or “decode”.

```
pattern = re.compile(r“\b(code|coding)\b”)
```

2. Text Tokenization

2.1 Introduction to Tokenization

Tokenization is the process of breaking down text into smaller units called tokens. These tokens can be words, characters, subwords, or phrases depending on the specific requirements of the NLP task.

2.2 Basic Regex-based Tokenization

2.2.1 Simple Word Tokenization

Python

```python
import re

def simple_word_tokenize(text):
    # Split on whitespace and punctuation
    tokens = re.findall(r'\b\w+\b', text)
    return tokens

text = "Hello, world! This is a simple example."
tokens = simple_word_tokenize(text)
print(tokens)
```

```
['Hello', 'world', 'This', 'is', 'a', 'simple', 'example']
```

2.2.2 Advanced Regex Tokenization

Python

```python
def advanced_tokenize(text):
    pattern = r"""
        [a-zA-Z]+ | (?<=\$)\d+(?:\.\d+)? | \d+(?:\.\d+)?(?=%)
    """
    tokens = re.findall(pattern, text, re.VERBOSE)
    return tokens

text = "The price is $19.99, and the discount is 15%!"
print(advanced_tokenize(text))
```

```
['The', 'price', 'is', '19.99', 'and', 'the', 'discount', 'is', '15']
```

2.3 NLTK Tokenization

NLTK *(Natural Language Toolkit)* is a Python library offering tools for tokenization, stemming, part-of-speech tagging, and more, making it ideal for text analysis and linguistic research.

While it's widely used for education and prototyping, **NLTK** is less efficient for large-scale applications compared to modern libraries like **spaCy** or **Hugging Face Transformers**.

2.3.1 Installation and Usage

Python

```python
import nltk
nltk.download('punkt_tab')  # Required for word and sentence tokenization
```

2.3 NLTK Tokenization

2.3.2 Word Tokenization

```python
from nltk.tokenize import word_tokenize, TreebankWordTokenizer
text = "Don't hesitate to use NLTK's tokenizer."
tokens = word_tokenize(text)
print(tokens)
# Output: ['Do', "n't", 'hesitate', 'to', 'use', 'NLTK', "'s",
'tokenizer', '.']
treebank = TreebankWordTokenizer()
tokens = treebank.tokenize(text)
print(tokens)
```

```
['Do', "n't", 'hesitate', 'to', 'use', 'NLTK', "'s", 'tokenizer', '.']
```

2.3 NLTK Tokenization

2.3.3 Sentence Tokenization

Python

```python
from nltk.tokenize import sent_tokenize
text = """Mr. Smith bought a car. He loves driving it!
          What will he buy next? Only time will tell."""
sentences = sent_tokenize(text)
print(sentences)
```

```
['Mr. Smith bought a car.', 'He loves driving it!', 'What will he buy next?', 'Only time will tell.']
```

2.3 NLTK Tokenization

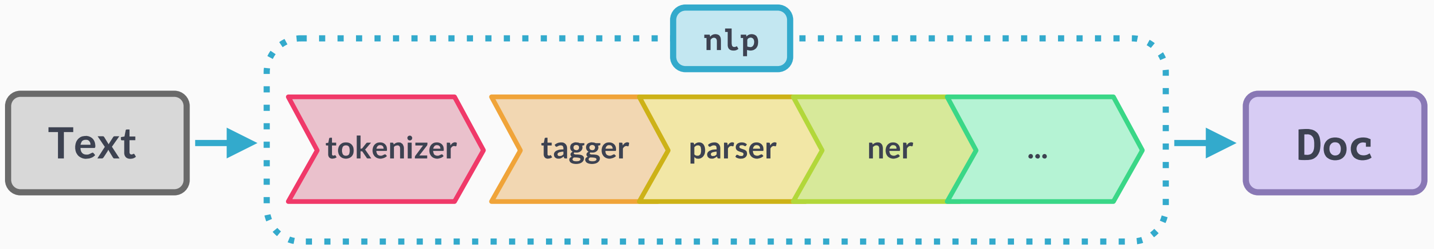
2.3.4 Regular Expression Tokenizer

```python
from nltk.tokenize import RegexpTokenizer
tokenizer = RegexpTokenizer(r'\w+|[^\w\s]+')
text = "Hello, World! How's it going?"
tokens = tokenizer.tokenize(text)
print(tokens)
```

```
[‘Hello’, ‘,’, ‘World’, ‘!’, ‘How’, “‘“, ‘s’, ‘it’, ‘going’, ‘?’]
```

2.4 spaCy Tokenization

spaCy[1] offers faster, more efficient tokenization with built-in linguistic annotations, making it better suited for production pipelines than rule-based alternatives.



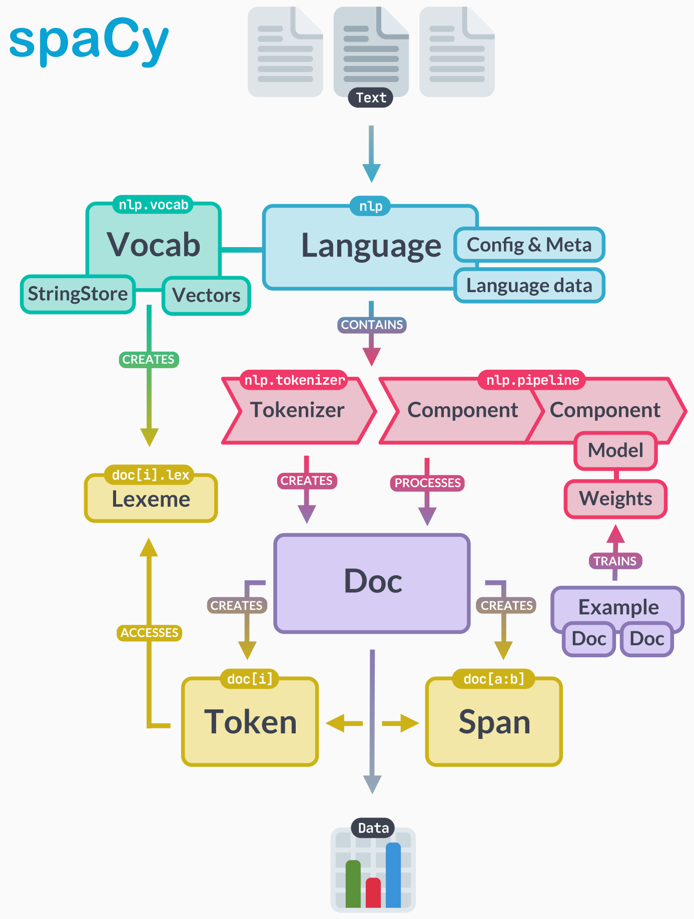
2.4.1 Installation and Usage

```shell
python -m spacy download en_core_web_sm
```

```python
import spacy
nlp = spacy.load('en_core_web_sm') # Load the English model
```

[1] https://spacy.io/

2.4 spaCy Tokenization



2.4 spaCy Tokenization

```python
def spacy_tokenize(text):
	doc = nlp(text)
	return [token.text for token in doc]

text = "spaCy's tokenizer is industrial-strength!"
tokens = spacy_tokenize(text)
print(tokens)
```

[“spaCy”, “‘s“, “tokenizer”, “is”, “industrial”, “-”, “strength”, “!”]

2.4.2 Tokenization with Linguistic Features

Python

```python
text = "Apple's stock price rose 5.2% to $200 in 2024. Wow!"

# Tokenize and analyze the text
doc = nlp(text) # Transform the text into a Doc object
# Extract tokens with linguistic features
for token in doc:
    print(
        f"Token: {token.text:<10}",
        f"POS: {token.pos_:<8}",
        f"Lemma: {token.lemma_:<10}",
        f"Is punctuation? {token.is_punct}"
    )
```

2.4 spaCy Tokenization

```
"""
Token: Apple       POS: PROPN    Lemma: Apple     Is punctuation? False
Token: 's          POS: PART     Lemma: 's        Is punctuation? False
Token: stock       POS: NOUN     Lemma: stock     Is punctuation? False
Token: price       POS: NOUN     Lemma: price     Is punctuation? False
Token: rose        POS: VERB     Lemma: rise      Is punctuation? False
Token: 5.2         POS: NUM      Lemma: 5.2       Is punctuation? False
Token: %           POS: NOUN     Lemma: %         Is punctuation? True
Token: to          POS: ADP      Lemma: to        Is punctuation? False
Token: $           POS: SYM      Lemma: $         Is punctuation? False
Token: 200         POS: NUM      Lemma: 200       Is punctuation? False
...
"""
```

2.4 spaCy Tokenization

Python

```python
def analyze_tokens(text):
    """
    Analyze the tokens in a text using spaCy.
    """
    doc = nlp(text)
    for token in doc:
        print(f"""
        Text: {token.text}
        Lemma: {token.lemma_}
        POS: {token.pos_}
        Is stop word: {token.is_stop}
        """)

```

2.4 spaCy Tokenization

```
text = "Running quickly through the forest"
analyze_tokens(text)
"""
Text: Running
Lemma: run
POS: VERB
Is stop word: False

Text: quickly
Lemma: quickly
POS: ADV
Is stop word: False
"""
```

2.5 Polyglot Tokenization

Polyglot is an open-source NLP library designed for multilingual text processing, supporting over 130 languages for tasks like tokenization, named entity recognition, and part-of-speech tagging. Unlike **spaCy** or **NLTK**, it specializes in low-resource languages, making it useful for linguistic diversity but less optimized for speed or deep learning integration.

2.5.1 Installation and Usage

```shell
pip install polyglot
```

Shell

```python
from polyglot.text import Text
```

Python

2.5.2 Basic Usage

Python

```python
def polyglot_tokenize(text):
    text = Text(text)
    return list(text.words)

english_text = "Hello, world!"
spanish_text = "¡Hola, mundo!"
arabic_text = "مرحبا أيها العالم!"

for sample in [english_text, spanish_text, arabic_text]:
    tokens = polyglot_tokenize(sample)
    print(f"Original: {sample}")
    print(f"Tokens: {tokens}\n")
```

2.6 Comparison of Different Tokenizers

2.6.1 Regex vs. NLTK vs. spaCy vs. Polyglot

Python

```python
text = "Don't forget those in need!"

print("Regex:", simple_word_tokenize(text))
# Regex: ['Don', 't', 'forget', 'those', 'in', 'need']
print("NLTK:", word_tokenize(text))
# NLTK: ['Do', "n't", 'forget', 'those', 'in', 'need', '!']
print("spaCy:", spacy_tokenize(text))
# spaCy: ['Do', "n't", 'forget', 'those', 'in', 'need', '!']
print("Polyglot:", polyglot_tokenize(text))
# Polyglot: ["Don't", 'forget', 'those', 'in', 'need', '!']
```

2.6 Comparison of Different Tokenizers

2.6.2 Strengths and Use Cases

1. **Regex-based Tokenization**
   - 🛑 Simple, custom tokenization rules
   - 👍 Fast, flexible, easy to modify
   - 👎 Can't handle complex linguistic cases
2. **NLTK**
   - 🛑 Academic and research projects
   - 👍 Rich features, well-documented
   - 👎 Slower than some alternatives

2.6 Comparison of Different Tokenizers

3. **spaCy**
    - 🛑 Production environments
    - 👍 Fast, modern, good defaults
    - 👎 Larger memory footprint

4. **Polyglot**
    - 🛑 Multilingual projects
    - 👍 Excellent language coverage
    - 👎 Can be slower, fewer features

1. **Choose the Right Tokenizer**

```python
def select_tokenizer(text, language='en', needs_speed=False):
    if needs_speed and language == 'en':
        return spacy_tokenize(text)
    elif language != 'en':
        return polyglot_tokenize(text)
    else:
        return word_tokenize(text)
```

2.7 Best Practices

```python
# Sample texts for testing different tokenization paths
test_texts = {
    "english_basic": """This sentence contains simple punctuation, numbers (123), and some special characters !@#$.""",

    "english_complex": """Mr. Smith's car broke down at 3:30 P.M. "This is terrible," he thought.""",

    "multilingual": """Hello in French is Bonjour. In Spanish, hello is ¡Hola!""",

    "technical": """Python3 supports utf-8 encoding. Variables use snake_case by convention."""
}
```

2.7 Best Practices

Test case 1: Basic English, no speed requirement

Python
```python
print("Test 1: Basic English (default settings)")
result1 = select_tokenizer(test_texts["english_basic"])
print(f"Tokens: {result1}\n")
```

['This', 'sentence', 'contains', 'simple', 'punctuation', ',', 'numbers', '(', '123', ')', ',', 'and', 'some', 'special', 'characters', '!', '@', '#', '$', '.']

2.7 Best Practices

Test case 2: Non-English text

```python
print("Test 2: Multilingual text")
result2 = select_tokenizer(test_texts["multilingual"], language="fr")
print(f"Tokens: {result2}\n")
```

```
['Hello', 'in', 'French', 'is', 'Bonjour', '.', 'In', 'Spanish', ',', 'hello', 'is', '¡', 'Hola', '!']
```

2.7 Best Practices

Test case 3: English with speed requirement

```python
print("Test 3: English with speed optimization")
result3 = select_tokenizer(test_texts["english_complex"], needs_speed=True)
print(f"Tokens: {result3}\n")
```

```
[‘Mr.’, ‘Smith’, “‘s“, ‘car’, ‘broke’, ‘down’, ‘at’, ‘3:30’, ‘P.M.’, ‘““, ‘This’, ‘is’, ‘terrible’, ‘,’, ‘““, ‘he’, ‘thought’, ‘.’]
```

Test case 4: Technical text with special characters

```python
print("Test 4: Technical text")
result4 = select_tokenizer(test_texts["technical"])
print(f"Tokens: {result4}")
```

['Python3', 'supports', 'utf-8', 'encoding', '.', 'Variables', 'use', 'snake_case', 'by', 'convention', '.']

2. **Pre-processing**

Python

```python
def preprocess_text(text):
        # Convert to lowercase
        text = text.lower()
        # Remove extra whitespace
        text = re.sub(r'\s+', ' ', text).strip()
        # Remove special characters (if needed)
        text = re.sub(r'[^\w\s]', '', text)
        return text
```

' Hello WORLD '	'!@#$%^&*()_+'	'Hello\n\tWorld\n Python'
'hello world'	'_'	'hello world python'

3. **Handling Special Cases**

```python
def handle_special_cases(tokens):
    """Expand contractions in a token list."""
    contractions = {"n't": "not", "'s": "is", "'re": "are"}  # Added common cases
    expanded_tokens = []
    for token in tokens:
        expanded_tokens.append(contractions.get(token, token)) # dict.get(key, default)
    return expanded_tokens
```

[‘They’, ‘ca’, “n’t”, ‘believe’, ‘it’, “‘s“, ‘already’, ‘Firday’]

[‘They’, ‘ca’, ‘not’, ‘believe’, ‘it’, ‘is’, ‘already’, ‘Firday’]

3. Text Processing and Visualization

3.1.1 What is Text Preprocessing?

Text preprocessing is the process of cleaning and transforming raw text into a format that's more suitable for analysis.

Think of it as preparing ingredients before cooking - just as you wash and chop vegetables before cooking, you clean and standardize text before analysis.

1. Raw text: *"RT @username: Check out our new product!!! It's AMAZING... 😊 www.example.com #awesome"*
2. Preprocessed text: *"check out our new product it is amazing"*

3.1 Fundamental Concepts and Definitions

3.1.2 Key Preprocessing Steps

3.1.2.1 Tokenization

The process of breaking down text into individual units (tokens), typically words or subwords.

1. Sentence: *“I love natural language processing!”*
2. Tokens: *[“I”, “love”, “natural”, “language”, “processing”, “!”]*

1. Sentence: *“Bizerte City is beautiful”*
2. Tokens: *[“Bizerte”, “City”, “is”, “beautiful”]*

3.1 Fundamental Concepts and Definitions

3.1.2.2 Stop Word Removal

Eliminating common words that typically don’t carry significant meaning.

Common stop words in English: “the”, “is”, “at”, “which”, “on”, etc.

1. Original: *“The cat is on the mat”*
2. After stop word removal: *“cat mat”*

3.1 Fundamental Concepts and Definitions

3.1.2.3 Lemmatization

Reducing words to their base or dictionary form (lemma).

```
am, are, is → be
running, ran, runs → run
better, best → good
wolves → wolf
```

3.1 Fundamental Concepts and Definitions

3.1.2.4 Stemming

Reducing words to their root form by removing affixes, often resulting in non-dictionary words.

```
running → run
fishing → fish
completely → complet
authentication → authent
```

3.1.3 Bag of Words (BoW)

A text representation method that describes the occurrence of words within a document. It creates a vocabulary of unique words and represents each document as a vector of word frequencies.

Documents:

1. "The cat likes milk"
2. "The dog hates milk"

Vocabulary:

["the", "cat", "dog", "likes", "hates", "milk"]

BoW representations:

- Doc 1: [1, 1, 0, 1, 0, 1]
- Doc 2: [1, 0, 1, 0, 1, 1]

3.1 Fundamental Concepts and Definitions

- **Document 1**: "I love machine learning"
- **Document 2**: "I love deep learning"
- **Document 3**: "Deep learning is a subset of machine learning"

3.1.3.1 Vocabulary

The unique words across all documents: "I", "love", "machine", "learning", "deep", "is", "a", "subset", "of"

3.1.3.2 Document Vectors

Document 1: [1, 1, 1, 1, 0, 0, 0, 0, 0]

Document 2: [1, 1, 0, 1, 1, 0, 0, 0, 0]

Document 3: [0, 0, 1, 2, 1, 1, 1, 1, 1]

3.1.3.3 BoW Matrix

Document	I	love	machine	learning	deep	is	a	subset	of
Doc 1	1	1	1	1	0	0	0	0	0
Doc 2	1	1	0	1	1	0	0	0	0
Doc 3	0	0	1	2	1	1	1	1	1

3.1.3.4 Document Similarity Analysis

Let's verify document similarity by calculating dot products between document vectors:

$$\text{Doc 1} \cdot \text{Doc 2} = (1\text{x}1) + (1\text{x}1) + (1\text{x}0) + (1\text{x}1) + (0\text{x}1) + (0\text{x}0) + (0\text{x}0) + (0\text{x}0) + (0\text{x}0) = 3$$

$$\text{Doc 1} \cdot \text{Doc 3} = (1\text{x}0) + (1\text{x}0) + (1\text{x}1) + (1\text{x}2) + (0\text{x}1) + (0\text{x}1) + (0\text{x}1) + (0\text{x}0) + (0\text{x}1) = 3$$

$$\text{Doc 2} \cdot \text{Doc 3} = (1\text{x}0) + (1\text{x}0) + (0\text{x}1) + (1\text{x}2) + (1\text{x}1) + (0\text{x}1) + (0\text{x}1) + (0\text{x}1) + (0\text{x}1) = 3$$

3.1 Fundamental Concepts and Definitions

For proper similarity comparison, we should normalize using cosine similarity:

$$\cos(\theta) = \frac{(A) \cdot (B)}{\|(A)\| \times \|(B)\|}$$

Doc 1 magnitude: $\sqrt{1^2 + 1^2 + 1^2 + 1^2 + 0^2 + 0^2 + 0^2 + 0^2 + 0^2} = \sqrt{4} = 2$

Doc 2 magnitude: $\sqrt{1^2 + 1^2 + 0^2 + 1^2 + 1^2 + 0^2 + 0^2 + 0^2 + 0^2} = \sqrt{4} = 2$

Doc 3 magnitude: $\sqrt{0^2 + 0^2 + 1^2 + 2^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2} = \sqrt{10} \approx 3.16$

Cosine similarities:

Doc 1 & Doc 2	**Doc 1 & Doc 3**	**Doc 2 & Doc 3**
$\frac{3}{2\times2} = 0.75$	$\frac{3}{2\times3.16} \approx 0.47$	$\frac{3}{2\times3.16} \approx 0.47$

Doc 1 and **Doc 2** are more similar to each other (0.75) than either is to **Doc 3** (0.47).

3.1.4 Term Frequency-Inverse Document Frequency (TF-IDF)

A numerical statistic that reflects how important a word is to a document in a collection of documents.

Consider these news articles:

1. "The new iPhone features advanced AI capabilities"
2. "The new Android phone launches today"
3. "The weather is nice today"

The word "the" appears in all documents, so it gets a low IDF score.
The word "iPhone" appears in only one document, so it gets a high IDF score.

3.1 Fundamental Concepts and Definitions

3.1.4.1 TF-IDF Calculation

It is calculated by multiplying two metrics: Term Frequency (TF) and Inverse Document Frequency (IDF).

The TF-IDF score for a term t in document d from a document collection D is calculated as:

$$\text{TF-IDF}(t, d, D) = \text{TF}(t, d) \times \text{IDF}(t, D)$$

Where:

1. **Term Frequency (TF)** measures how frequently term t appears in document d:

$$\text{TF}(t, d) = \frac{\text{Number of times } t \text{ appears in } d}{\text{Total number of terms in } d}$$

2. **Inverse Document Frequency (IDF)** measures the importance of term t across all documents:

3.1 Fundamental Concepts and Definitions

$$\text{IDF}(t, D) = \log\left(\frac{N}{\text{DF}(t)}\right)$$

Where:

- N is the total number of documents in collection D
- $\text{DF}(t)$ is the number of documents containing term t

3.1.4.2 Computing Example

- **Document 1:** "This smartphone has a great camera and long battery life"
- **Document 2:** "The camera on this laptop is decent but the keyboard is excellent"
- **Document 3:** "Battery life is crucial for any smartphone"
- **Document 4:** "This laptop has a fast processor and excellent keyboard"

Calculate TF for each term in each document: First, let's count terms in each document (after removing common words like "this", "is", "the", etc.):

3.1 Fundamental Concepts and Definitions

For **Document 1** (7 terms):

- $\text{TF}(\text{smartphone}) = \frac{1}{7} \approx 0.143$
- $\text{TF}(\text{great}) = \frac{1}{7} \approx 0.143$
- $\text{TF}(\text{camera}) = \frac{1}{7} \approx 0.143$
- $\text{TF}(\text{long}) = \frac{1}{7} \approx 0.143$
- $\text{TF}(\text{battery}) = \frac{1}{7} \approx 0.143$
- $\text{TF}(\text{life}) = \frac{1}{7} \approx 0.143$

For **Document 2** (7 terms):

- $\text{TF}(\text{camera}) = \frac{1}{7} \approx 0.143$
- $\text{TF}(\text{laptop}) = \frac{1}{7} \approx 0.143$
- $\text{TF}(\text{decent}) = \frac{1}{7} \approx 0.143$
- $\text{TF}(\text{keyboard}) = \frac{1}{7} \approx 0.143$
- $\text{TF}(\text{excellent}) = \frac{1}{7} \approx 0.143$

3.1 Fundamental Concepts and Definitions

For **Document 3** (4 terms):

- $\mathrm{TF}(\text{battery}) = \frac{1}{4} = 0.25$
- $\mathrm{TF}(\text{life}) = \frac{1}{4} = 0.25$
- $\mathrm{TF}(\text{crucial}) = \frac{1}{4} = 0.25$
- $\mathrm{TF}(\text{smartphone}) = \frac{1}{4} = 0.25$

For **Document 4** (5 terms):

- $\mathrm{TF}(\text{laptop}) = \frac{1}{5} = 0.2$
- $\mathrm{TF}(\text{fast}) = \frac{1}{5} = 0.2$
- $\mathrm{TF}(\text{processor}) = \frac{1}{5} = 0.2$
- $\mathrm{TF}(\text{excellent}) = \frac{1}{5} = 0.2$
- $\mathrm{TF}(\text{keyboard}) = \frac{1}{5} = 0.2$

Calculate IDF for each term across all documents:

Total documents (N) = 4

- $\text{IDF}(\text{smartphone}) = \log\left(\frac{4}{2}\right) \approx 0.301$
- $\text{IDF}(\text{great}) = \log\left(\frac{4}{1}\right) \approx 0.602$
- $\text{IDF}(\text{camera}) = \log\left(\frac{4}{2}\right) \approx 0.301$
- $\text{IDF}(\text{long}) = \log\left(\frac{4}{1}\right) \approx 0.602$
- $\text{IDF}(\text{battery}) = \log\left(\frac{4}{2}\right) \approx 0.301$
- $\text{IDF}(\text{life}) = \log\left(\frac{4}{2}\right) \approx 0.301$
- $\text{IDF}(\text{laptop}) = \log\left(\frac{4}{2}\right) \approx 0.301$
- $\text{IDF}(\text{decent}) = \log\left(\frac{4}{1}\right) \approx 0.602$
- $\text{IDF}(\text{keyboard}) = \log\left(\frac{4}{2}\right) \approx 0.301$
- $\text{IDF}(\text{excellent}) = \log\left(\frac{4}{2}\right) \approx 0.301$
- $\text{IDF}(\text{crucial}) = \log\left(\frac{4}{1}\right) \approx 0.602$
- $\text{IDF}(\text{fast}) = \log\left(\frac{4}{1}\right) \approx 0.602$
- $\text{IDF}(\text{processor}) = \log\left(\frac{4}{1}\right) \approx 0.602$

3.1 Fundamental Concepts and Definitions

Calculate TF-IDF for each term in each document:

For **Document 1**:

- $\text{TF-IDF}(\text{smartphone}, d_1) \approx 0.043$
- $\text{TF-IDF}(\text{great}, d_1) \approx 0.086$
- $\text{TF-IDF}(\text{camera}, d_1) \approx 0.043$
- $\text{TF-IDF}(\text{long}, d_1) \approx 0.086$
- $\text{TF-IDF}(\text{battery}, d_1) \approx 0.043$
- $\text{TF-IDF}(\text{life}, d_1) \approx 0.043$

For **Document 2**:

- $\text{TF-IDF}(\text{camera}, d_2) \approx 0.043$
- $\text{TF-IDF}(\text{laptop}, d_2) \approx 0.043$
- $\text{TF-IDF}(\text{decent}, d_2) \approx 0.086$
- $\text{TF-IDF}(\text{keyboard}, d_2) \approx 0.043$
- $\text{TF-IDF}(\text{excellent}, d_2) \approx 0.043$

For **Document 3**:

- $\text{TF-IDF}(\text{battery}, \text{d}_3) \approx 0.075$
- $\text{TF-IDF}(\text{life}, \text{d}_3) \approx 0.075$
- $\text{TF-IDF}(\text{crucial}, \text{d}_3) \approx 0.151$
- $\text{TF-IDF}(\text{smartphone}, \text{d}_3) \approx 0.075$

For **Document 4**:

- $\text{TF-IDF}(\text{laptop}, \text{d}_4) \approx 0.060$
- $\text{TF-IDF}(\text{fast}, \text{d}_4) \approx 0.120$
- $\text{TF-IDF}(\text{processor}, \text{d}_4) \approx 0.120$
- $\text{TF-IDF}(\text{excellent}, \text{d}_4) \approx 0.060$
- $\text{TF-IDF}(\text{keyboard}, \text{d}_4) \approx 0.060$

The terms with the highest TF-IDF scores in each document:

Document	*Highest TF-IDF Terms*
Document 1	"great" (0.086) and "long" (0.086)
Document 2	"decent" (0.086)
Document 3	"crucial" (0.151)
Document 4	"fast" (0.120) and "processor" (0.120)

3.2 Complete Pipeline Example

Let's analyze a customer review:

Original review:

"I've been using this phone for 3 months now... It's AMAZING!!! The battery life is incredible, and the camera takes beautiful pics. Can't believe how good it is :) Would definitely recommend to my friends & family!!!"

3.2 Complete Pipeline Example

1. **Cleaning**:
   "i have been using this phone for three months now it is amazing the battery life is incredible and the camera takes beautiful pictures cannot believe how good it is would definitely recommend to my friends and family"
2. **Tokenization**:
   ["i", "have", "been", "using", "this", "phone", "for", "three", "months", ...]
3. **Stop Word Removal**:
   ["phone", "three", "months", "amazing", "battery", "life", "incredible", "camera", "takes", "beautiful", "pictures", "good", "definitely", "recommend", "friends", "family"]
4. **Lemmatization**:
   ["phone", "month", "amazing", "battery", "life", "incredible", "camera", "take", "beautiful", "picture", "good", "definitely", "recommend", "friend", "family"]

3.2 Complete Pipeline Example

Text Processing

Marimo/text_processing.py

3.3 Text Visualization Concepts

3.3.1 Frequency Distribution Plots

Charts showing how often different words appear in a text.

Real-world applications:

- Comparing vocabulary usage across different authors;
- Analyzing Twitter hashtag popularity over time;
- Studying language patterns in different genres of literature.

3.3.2 Word Clouds

A visual representation where word size corresponds to its frequency in the text.

- Analyzing customer reviews to identify common themes;
- Visualizing key topics in political speeches;
- Summarizing survey responses.

3.3 Text Visualization Concepts

Text Visualization

Marimo/text_visualization.py

3.4 Common Use Cases and Applications

1. **Sentiment Analysis**
   - Customer review processing
   - Social media monitoring
   - Brand reputation tracking
2. **Content Classification**
   - News article categorization
   - Spam detection
   - Document sorting
3. **Text Summarization**
   - News article summarization
   - Document abstract generation
   - Meeting notes condensation
4. **Keyword Extraction**
   - SEO optimization
   - Content tagging
   - Research paper indexing

3.5 Best Practices and Tips

1. **Choose the Right Tools**
   - Use `NLTK` for research and experimentation
   - Use `spaCy` for production environments
   - Use `scikit-learn` for machine learning integration
2. **Performance Optimization**
3. **Error Handling**
4. **Evaluation Metrics**

4. Gensim Text Processing

4.1 What is Gensim?

Gensim is a robust, efficient library for topic modeling, document indexing, and similarity retrieval with large corpora. The name "Gensim" stands for "Generate Similar" - reflecting its core functionality of finding similar documents.

Key features:

- Memory efficient processing of large text collections
- Built-in implementations of popular algorithms like Word2Vec, Doc2Vec, FastText
- Streamlined document similarity calculations
- Topic modeling capabilities (LSA, LDA)

4.2 Basic Gensim Usage

4.2.1 Installation and Usage

```shell
pip install gensim
```

```python
import gensim
from gensim import corpora, models
```

4.2 Basic Gensim Usage

Building a Complete Text Analysis Pipeline

Marimo/gensim_text_processing.py

4.3 Best Practices and Tips

1. **Preprocessing**
   - Convert to lowercase
   - Remove stop words
   - Apply lemmatization
   - Handle special characters

2. **Memory Efficiency**
   - Use streaming corpus for large datasets
   - Implement memory-efficient iterators

```python
class MyCorpus:
    def __iter__(self):
        for line in open('mycorpus.txt'):
            yield dictionary.doc2bow(line.lower().split())
```

4.3 Best Practices and Tips

3. **Model Persistence**

   Save models

```python
dictionary.save('dictionary.gensim')
tfidf_model.save('tfidf.gensim')
```

   Load models

```python
dictionary = corpora.Dictionary.load('dictionary.gensim')
tfidf_model = models.TfidfModel.load('tfidf.gensim')
```

5. Named Entity Recognition (NER)

5.1 What is Named Entity Recognition?

Named Entity Recognition (NER) is a natural language processing technique that identifies and classifies named entities (key elements) in text into predefined categories.

The categories include but are not limited to:

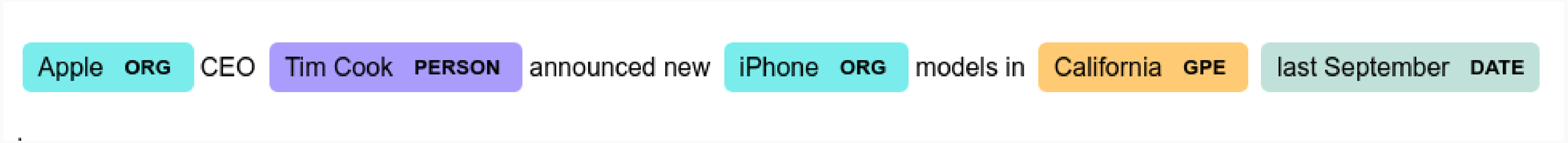
- Person names (e.g., “Barack Obama”, “Shakespeare”)
- Organizations (e.g., “Microsoft”, “United Nations”)
- Locations (e.g., “Paris”, “Mount Everest”)
- Date/Time expressions (e.g., “June 2024”, “last Monday”)
- Monetary values (e.g., “$1000”, “€50”)
- Percentages (e.g., “25%”, “three-quarters”)

5.2 Usage Example

Input text: *"Apple CEO Tim Cook announced new iPhone models in California last September."*

Identified entities:

- Apple (ORG)
- Tim Cook (PERSON)
- iPhone (ORG)
- California (GPE)
- September (DATE)



Apple ORG CEO Tim Cook PERSON announced new iPhone ORG models in California GPE last September DATE

5.2 Usage Example

Python

```python
import spacy
from spacy import displacy

nlp = spacy.load('en_core_web_sm')

doc = nlp("Apple CEO Tim Cook announced new iPhone models in California last September.")

displacy.serve(doc, style='ent')
```

5.3 NLTK vs spaCy

Extracting Named Entities

Marimo/extracting_named_entity.py

5.4 Common Use Cases

1. **Information Extraction**
   - Extracting company names from news articles
   - Identifying people mentioned in social media posts
   - Finding locations in travel blogs
2. **Document Classification**
   - Categorizing documents based on mentioned organizations
   - Sorting news articles by location
   - Grouping documents by date mentions

5.4 Common Use Cases

3. **Relationship Extraction**
   - Identifying business relationships between companies
   - Finding connections between people
   - Mapping event locations and dates

4. **Content Enrichment**
   - Adding metadata to documents
   - Linking entities to knowledge bases
   - Creating document summaries

Thank you for your attention!