

Natural Language Processing

An Introduction

Abdelbacet Mhamdi

2025-01-12

ISET Bizerte

Outline

1. Introduction to Regular Expressions (Regex)
2. Text Tokenization Guide
3. Text Processing, Visualization, and Concepts Guide
4. Text Processing and Visualization Guide
5. Gensim Text Processing Guide
6. Named Entity Recognition (NER) Guide

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

1.2.2 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"

1.2 Core Concepts

Metacharacter	Description	Example
+	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. Data Validation

- Email addresses
- Phone numbers
- Postal codes
- Passwords
- URLs

2. Text Processing

- Finding patterns in text
- Replacing specific text patterns
- Extracting information
- Parsing log files

3. Search Operations

- Advanced find/replace operations


1.3 Common Use Cases

- Pattern matching in large text files
- Content filtering

1.4 Python Implementation

1.4.1 Basic Pattern Matching

```
1  import re
2
3  # Simple pattern matching
4  text = "The quick brown fox jumps over the lazy dog"
5  pattern = r"fox"
6
7  # Search for pattern
8  match = re.search(pattern, text)
9  if match:
10     print(f"Found '{pattern}' at position: {match.start()}-
11           {match.end()}")
12
12 # Find all occurrences
```


 Python

1.4 Python Implementation

```
13 words = re.findall(r"\w+", text)
14 print(f"All words: {words}")
```

1.4.2 Email Validation Example

```
1 def is_valid_email(email):
2     pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
3     return bool(re.match(pattern, email))
4
5 # Test cases
6 emails = [
7     "user@example.com",
8     "invalid.email@com",
9     "user.name+tag@domain.co.uk",
```

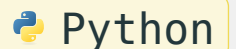
 Python

1.4 Python Implementation

```
10     "@invalid.com"
11 ]
12
13 for email in emails:
14     print(f"{email}: {'Valid' if is_valid_email(email) else 'Invalid'}")
```

1.4.3 Phone Number Formatting

```
1 def format_phone_number(phone):
2     # Remove all non-digit characters
3     digits = re.sub(r'\D', '', phone)
4
5     # Format as (XXX) XXX-XXXX
6     if len(digits) == 10:
```



1.4 Python Implementation

```
7         pattern = r'(\d{3})(\d{3})(\d{4})'
8         formatted = re.sub(pattern, r'(\1) \2-\3', digits)
9         return formatted
10    return "Invalid phone number"
11
12 # Test cases
13 numbers = [
14     "1234567890",
15     "123-456-7890",
16     "(123) 456-7890",
17     "12345"
18 ]
19
```


1.4 Python Implementation

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

1.4.4 Advanced Concepts


1.4.5 Character Classes

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

 Python

1.4.6 Quantifiers and Groups

```
1 # Quantifiers
2 pattern = r'\d{3}' # Exactly 3 digits
```

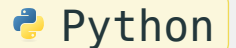
 Python

1.4 Python Implementation

```
3 pattern = r'\d{2,4}'    # Between 2 and 4 digits
4 pattern = r'\d{2,}'     # 2 or more digits
5
6 # Groups
7 pattern = r'(\w+)\s+\1' # Matches repeated words
```

1.4.7 Common Regex Functions in Python

```
1 import re
2
3 text = "The price is $19.99"
4
5 # Different matching functions
6 re.search(r'\$\d+\.\d+', text) # Finds first match
```



1.4 Python Implementation


```
7  re.findall(r'\$\d+\.\d+', text) # Finds all matches
8  re.sub(r'\$(\d+\.\d+)', r'\1', text) # Substitution
9
10 # Splitting text
11 re.split(r'\s+', text) # Split on whitespace
```

1.5 Best Practices

1. Use Raw Strings


- Always prefix regex patterns with `r` to avoid escape character issues

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

 Python

2. Compile Frequently Used Patterns

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

 Python


3. Be Specific

- Make patterns as specific as possible to avoid false matches
- Use start (^) and end (\$) anchors when matching whole strings

4. Test Thoroughly

- Test with both valid and invalid inputs
- Include edge cases in your tests

```
1 def test_pattern(pattern, test_cases):
2     regex = re.compile(pattern)
3     for test, expected in test_cases:
4         result = bool(regex.match(test))
5         print(f"'{test}': {'✓' if result == expected else 'x'}")
```

 Python

1.6 Common Pitfalls

1. Greedy vs. Non-Greedy Matching

```
1 # Greedy (default)
2 re.findall(r'<.*>', '<tag>text</tag>') # ['<tag>text</tag>']
3
4 # Non-greedy
5 re.findall(r'<.*?>', '<tag>text</tag>') # ['<tag>', '</tag>']
```




2. Performance Considerations

- Avoid excessive backtracking
- Be careful with nested quantifiers
- Use more specific patterns when possible

1.7 Exercise Examples


1. Basic Pattern Matching

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

 Python


2. Data Extraction

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

 Python

3. Password Validation

```
1 def is_strong_password(password):
2     # At least 8 chars, 1 upper, 1 lower, 1 digit, 1 special
```

 Python

1.7 Exercise Examples

```
3 pattern = r'^(?=.*[A-Z])(?=.*[a-z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$'
4 return bool(re.match(pattern, password))
```

2. Text Tokenization Guide

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 your NLP task.

2.2 1. Basic Regex-based Tokenization

2.2.1 Simple Word Tokenization

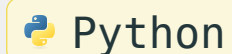
```
1  import re
2
3  def simple_word_tokenize(text):
4      # Split on whitespace and punctuation
5      tokens = re.findall(r'\b\w+\b', text)
6      return tokens
7
8  text = "Hello, world! This is a simple example."
9  tokens = simple_word_tokenize(text)
10 print(tokens) # ['Hello', 'world', 'This', 'is', 'a', 'simple',
    'example']
```

 Python

2.2 1. Basic Regex-based Tokenization

2.2.2 Advanced Regex Tokenization

```
1  def advanced_tokenize(text):  
2      # Pattern matches words, numbers, punctuation, and special characters  
3      pattern = r"""  
4          [\w]+                # Word characters  
5          |(?:[\d])?\d+(?:\.\d+)?%? # Numbers with optional decimal  
6          |[\.,!?\;"]          # Punctuation  
7          |[:']                # Special characters  
8      """  
9      tokens = re.findall(pattern, text, re.VERBOSE)  
10     return tokens  
11  
12 text = "The price is $19.99, and the discount is 15%!"
```



Python

2.2 1. Basic Regex-based Tokenization


```
13 print(advanced_tokenize(text))  
14 # ['The', 'price', 'is', '$19.99', ',', 'and', 'the', 'discount', 'is',  
    '15%', '!']
```

2.3 2. NLTK Tokenization

NLTK provides various tokenizers for different needs.

2.3.1 Installation and Setup

```
1 import nltk
2 nltk.download('punkt') # Required for word and sentence tokenization
```

 Python

2.3.2 Word Tokenization

```
1 from nltk.tokenize import word_tokenize, TreebankWordTokenizer
2
3 # Simple word tokenization
4 text = "Don't hesitate to use NLTK's tokenizer."
5 tokens = word_tokenize(text)
6 print(tokens)
7 # ["Do", "n't", "hesitate", "to", "use", "NLTK", "'", "tokenizer", "."]
```

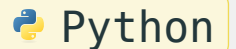
 Python

2.3 2. NLTK Tokenization

```
8
9 # TreebankWordTokenizer for Penn Treebank style tokenization
10 treebank = TreebankWordTokenizer()
11 tokens = treebank.tokenize(text)
12 print(tokens)
```

2.3.3 Sentence Tokenization

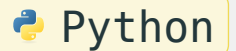
```
1 from nltk.tokenize import sent_tokenize
2
3 text = """Mr. Smith bought a car. He loves driving it!
4         What will he buy next? Only time will tell."""
5 sentences = sent_tokenize(text)
6 print(sentences)
```



2.3 2. NLTK Tokenization

2.3.4 Regular Expression Tokenizer

```
1 from nltk.tokenize import RegexpTokenizer
2
3 # Create custom patterns
4 tokenizer = RegexpTokenizer(r'\w+|[\^\w\s]+' )
5 text = "Hello, World! How's it going?"
6 tokens = tokenizer.tokenize(text)
7 print(tokens)
```



2.4 3. spaCy Tokenization

spaCy provides more advanced tokenization with linguistic features.

2.4.1 Installation and Setup

```
1 import spacy
2 nlp = spacy.load('en_core_web_sm')
```

 Python

2.4.2 Basic Tokenization

```
1 def spacy_tokenize(text):
2     doc = nlp(text)
3     return [token.text for token in doc]
4
5 text = "spaCy's tokenizer is industrial-strength!"
6 tokens = spacy_tokenize(text)
7 print(tokens)
```

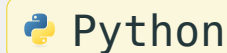
 Python

2.4 3. spaCy Tokenization

```
8 # ["spaCy", "'s", "tokenizer", "is", "industrial", "-", "strength", "!"]
```

2.4.3 Advanced Features

```
1 def analyze_tokens(text):  
2     doc = nlp(text)  
3     for token in doc:  
4         print(f"  
5             Text: {token.text}  
6             Lemma: {token.lemma_}  
7             POS: {token.pos_}  
8             Is stop word: {token.is_stop}  
9             """)  
10
```



2.4 3. spaCy Tokenization

```
11 text = "Running quickly through the forest"  
12 analyze_tokens(text)
```

2.5 4. Polyglot Tokenization

Polyglot is especially useful for multilingual tokenization.

2.5.1 Installation and Setup

```
1 from polyglot.text import Text
```

 Python

2.5.2 Basic Usage

```
1 def polyglot_tokenize(text):  
2     text = Text(text)  
3     return list(text.words)  
4  
5 # Multiple languages  
6 english_text = "Hello, world!"  
7 spanish_text = "¡Hola, mundo!"  
8 chinese_text = "你好，世界！"
```

 Python


2.5 4. Polyglot Tokenization

```
9
10 for sample in [english_text, spanish_text, chinese_text]:
11     tokens = polyglot_tokenize(sample)
12     print(f"Original: {sample}")
13     print(f"Tokens: {tokens}\n")
```

2.6 5. Comparison of Different Approaches

2.6.1 Handling Special Cases

```
1 text = "Don't forget the U.S.A.'s high-quality standards!"
2
3 # Compare different tokenizers
4 print("Regex:", simple_word_tokenize(text))
5 print("NLTK:", word_tokenize(text))
6 print("spaCy:", spacy_tokenize(text))
7 print("Polyglot:", polyglot_tokenize(text))
```

 Python

2.6.2 Strengths and Use Cases

1. **Regex-based Tokenization**

- Best for: Simple, custom tokenization rules
- Pros: Fast, flexible, easy to modify
- Cons: Can't handle complex linguistic cases

2.6 5. Comparison of Different Approaches

2. **NLTK**

- Best for: Academic and research projects
- Pros: Rich features, well-documented
- Cons: Slower than some alternatives

3. **spaCy**

- Best for: Production environments
- Pros: Fast, modern, good defaults
- Cons: Larger memory footprint


4. **Polyglot**

- Best for: Multilingual projects
- Pros: Excellent language coverage
- Cons: Can be slower, fewer features

2.7 6. Best Practices


1. Choose the Right Tokenizer

```
1 def select_tokenizer(text, language='en', needs_speed=False):
2     if needs_speed and language == 'en':
3         return spacy_tokenize(text)
4     elif language != 'en':
5         return polyglot_tokenize(text)
6     else:
7         return word_tokenize(text)
```

 Python

2. Pre-processing

```
1 def preprocess_text(text):
2     # Convert to lowercase
3     text = text.lower()
```


 Python

2.7 6. Best Practices

```
4      # Remove extra whitespace
5      text = re.sub(r'\s+', ' ', text).strip()
6      # Remove special characters (if needed)
7      text = re.sub(r'^\w\s]', '', text)
8      return text
```

3. Handling Special Cases

```
1  def handle_special_cases(tokens):
2      # Handle contractions
3      contractions = {"n't": "not", "'s": "is"}
4      expanded_tokens = []
5      for token in tokens:
6          if token in contractions:
```

 Python

2.7 6. Best Practices

```
7         expanded_tokens.append(contractions[token])
8     else:
9         expanded_tokens.append(token)
10    return expanded_tokens
```

3. Text Processing, Visualization, and Concepts Guide

3.1 Fundamental Concepts and Definitions

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.

Real-world example:

- 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"

3.1.2 Key Preprocessing Steps

3.1.2.1 1. Tokenization

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

3.1 Fundamental Concepts and Definitions

Real-world examples:

```
1 Sentence: "I love natural language processing!"
2 Tokens: ["I", "love", "natural", "language", "processing", "!"]
3
4 Sentence: "New York City is beautiful"
5 Tokens: ["New", "York", "City", "is", "beautiful"]
```

3.1.2.2 2. Stop Word Removal

Definition: Eliminating common words that typically don't carry significant meaning.

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

Real-world example:

```
1 Original: "The cat is on the mat"
```

3.1 Fundamental Concepts and Definitions

2 After stop word removal: "cat mat"

3.1.2.3 3. Lemmatization

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

Real-world examples:

1 am, are, is → be

2 running, ran, runs → run

3 better, best → good

4 wolves → wolf

3.1.2.4 4. Stemming

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

3.1 Fundamental Concepts and Definitions

Real-world examples:

- 1 running → run
- 2 fishing → fish
- 3 completely → complet
- 4 authentication → authent

3.1.3 Bag of Words (BoW)

Definition: 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.

Real-world example:

- 1 Documents:

3.1 Fundamental Concepts and Definitions

```
2 1. "The cat likes milk"
3 2. "The dog hates milk"
4
5 Vocabulary: ["the", "cat", "dog", "likes", "hates", "milk"]
6 BoW representations:
7 Doc 1: [1, 1, 0, 1, 0, 1]
8 Doc 2: [1, 0, 1, 0, 1, 1]
```

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

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

Real-world example:

```
1 Consider these news articles:
```

3.1 Fundamental Concepts and Definitions

2 1. "The new iPhone features advanced AI capabilities"

3 2. "The new Android phone launches today"

4 3. "The weather is nice today"

5

6 The word "the" appears in all documents, so it gets a low IDF score.

7 The word "iPhone" appears in only one document, so it gets a high IDF score.

3.2 Text Visualization Concepts

3.2.1 Word Clouds

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

Real-world applications:

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

3.2.2 Frequency Distribution Plots

Definition: 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 Complete Pipeline Example with Real Text

Let's analyze a customer review:

Original review:

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

Pipeline steps:

1. **Cleaning:**

```
1 "i have been using this phone for three months now it is amazing the  
battery"
```

3.3 Complete Pipeline Example with Real Text

```
2 life is incredible and the camera takes beautiful pictures cannot  
believe how  
3 good it is would definitely recommend to my friends and family"
```

2. Tokenization:

```
1 ["i", "have", "been", "using", "this", "phone", "for", "three",  
"months", ...]
```

3. Stop Word Removal:

```
1 ["phone", "three", "months", "amazing", "battery", "life",  
"incredible",  
2 "camera", "takes", "beautiful", "pictures", "good", "definitely",  
"recommend",  
3 "friends", "family"]
```


3.3 Complete Pipeline Example with Real Text

4. Lemmatization:

```
1 ["phone", "month", "amazing", "battery", "life", "incredible",  
  "camera",  
2  "take", "beautiful", "picture", "good", "definitely", "recommend",  
  "friend",  
3  "family"]
```

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

3.4 Common Use Cases and Applications


- SEO optimization
- Content tagging
- Research paper indexingA

4. Text Processing and Visualization Guide

4.1 1. Word Frequency Visualization

4.1.1 Basic Word Frequency Chart

```
1  import matplotlib.pyplot as plt
2  from collections import Counter
3  import seaborn as sns
4
5  def plot_word_frequency(text, top_n=10):
6      # Tokenize and count words
7      words = text.lower().split()
8      word_freq = Counter(words)
9
10     # Get top N words
11     top_words = dict(word_freq.most_common(top_n))
12
```

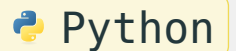
 Python

4.1 1. Word Frequency Visualization

```
13     # Create bar plot
14     plt.figure(figsize=(12, 6))
15     plt.bar(top_words.keys(), top_words.values())
16     plt.xticks(rotation=45, ha='right')
17     plt.title('Top Word Frequencies')
18     plt.xlabel('Words')
19     plt.ylabel('Frequency')
20     plt.tight_layout()
21     plt.show()
```

4.1.2 Word Cloud Visualization

```
1  from wordcloud import WordCloud
2
```




4.1 1. Word Frequency Visualization

```
3  def generate_wordcloud(text):
4      wordcloud = WordCloud(
5          width=800, height=400,
6          background_color='white',
7          max_words=100
8      ).generate(text)
9
10     plt.figure(figsize=(10, 5))
11     plt.imshow(wordcloud, interpolation='bilinear')
12     plt.axis('off')
13     plt.show()
```

4.1 1. Word Frequency Visualization

4.1.3 Advanced Visualization with Seaborn

```
1  def plot_word_distribution(text, top_n=10):
2      # Create word frequency distribution
3      words = text.lower().split()
4      word_freq = Counter(words)
5
6      # Convert to DataFrame for Seaborn
7      import pandas as pd
8      df = pd.DataFrame(word_freq.most_common(top_n),
9                          columns=['Word', 'Frequency'])
10
11     # Create plot
12     plt.figure(figsize=(12, 6))
```

 Python

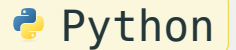
4.1 1. Word Frequency Visualization

```
13     sns.barplot(data=df, x='Word', y='Frequency')
14     plt.xticks(rotation=45, ha='right')
15     plt.title('Word Frequency Distribution')
16     plt.tight_layout()
17     plt.show()
```

4.2 2. Bag of Words Implementation

4.2.1 Simple Bag of Words

```
1  from sklearn.feature_extraction.text import CountVectorizer
2
3  def create_bow(documents):
4      # Initialize vectorizer
5      vectorizer = CountVectorizer()
6
7      # Create BOW representation
8      X = vectorizer.fit_transform(documents)
9
10     # Get feature names (vocabulary)
11     feature_names = vectorizer.get_feature_names_out()
12
```



4.2 2. Bag of Words Implementation

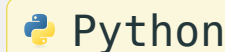
```
13     # Convert to array
14     bow_array = X.toarray()
15
16     return bow_array, feature_names, vectorizer
17
18 # Example usage
19 documents = [
20     "The cat sat on the mat",
21     "The dog ran in the park",
22     "The cat and dog played"
23 ]
24
25 bow_array, features, vectorizer = create_bow(documents)
```

4.2 2. Bag of Words Implementation

```
26 print("Features:", features)
27 print("BOW Matrix:\n", bow_array)
```

4.2.2 TF-IDF Implementation

```
1 from sklearn.feature_extraction.text import TfidfVectorizer
2
3 def create_tfidf(documents):
4     # Initialize TF-IDF vectorizer
5     tfidf = TfidfVectorizer()
6
7     # Create TF-IDF matrix
8     X = tfidf.fit_transform(documents)
9
```




4.2 2. Bag of Words Implementation

```
10     # Get feature names
11     feature_names = tfidf.get_feature_names_out()
12
13     return X.toarray(), feature_names, tfidf
```

4.3 3. Complete Text Preprocessing Pipeline

4.3.1 Comprehensive Pipeline Class

```
1  import nltk
2  import re
3  from nltk.tokenize import word_tokenize
4  from nltk.corpus import stopwords
5  from nltk.stem import WordNetLemmatizer
6  from nltk.stem.porter import PorterStemmer
7
8  class TextPreprocessor:
9      def __init__(self, language='english'):
10         # Download required NLTK data
11         nltk.download('punkt')
12         nltk.download('stopwords')
```

 Python

4.3 3. Complete Text Preprocessing Pipeline

```
13         nltk.download('wordnet')
14
15         self.language = language
16         self.stop_words = set(stopwords.words(language))
17         self.lemmatizer = WordNetLemmatizer()
18         self.stemmer = PorterStemmer()
19
20     def clean_text(self, text):
21         """Basic text cleaning"""
22         # Convert to lowercase
23         text = text.lower()
24
25         # Remove special characters and digits
```

4.3 3. Complete Text Preprocessing Pipeline

```
26         text = re.sub(r'^a-zA-Z\s', '', text)
27
28         # Remove extra whitespace
29         text = re.sub(r'\s+', ' ', text).strip()
30
31         return text
32
33     def tokenize(self, text):
34         """Tokenize text"""
35         return word_tokenize(text)
36
37     def remove_stopwords(self, tokens):
38         """Remove stop words"""
```


4.3 3. Complete Text Preprocessing Pipeline

```
39         return [token for token in tokens if token not in
40                 self.stop_words]
41     def lemmatize(self, tokens):
42         """Lemmatize tokens"""
43         return [self.lemmatizer.lemmatize(token) for token in tokens]
44
45     def stem(self, tokens):
46         """Stem tokens"""
47         return [self.stemmer.stem(token) for token in tokens]
48
49     def process(self, text, use_stemming=False):
50         """Complete preprocessing pipeline"""
51         # Clean text
```

4.3 3. Complete Text Preprocessing Pipeline


```
52         cleaned_text = self.clean_text(text)
53
54         # Tokenize
55         tokens = self.tokenize(cleaned_text)
56
57         # Remove stopwords
58         tokens = self.remove_stopwords(tokens)
59
60         # Lemmatize or stem
61         if use_stemming:
62             tokens = self.stem(tokens)
63         else:
64             tokens = self.lemmatize(tokens)
```

4.3 3. Complete Text Preprocessing Pipeline

```
65
66         return tokens
67
68 # Example usage
69 preprocessor = TextPreprocessor()
70 text = "The cats are running quickly through the forest!"
71 processed_tokens = preprocessor.process(text)
72 print("Processed tokens:", processed_tokens)
```

4.3.2 Advanced Pipeline with Custom Features

```
1 class AdvancedTextPreprocessor(TextPreprocessor):
2     def __init__(self, language='english', custom_stopwords=None):
3         super().__init__(language)
```

 Python

4.3 3. Complete Text Preprocessing Pipeline

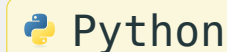
```
4
5     # Add custom stopwords if provided
6     if custom_stopwords:
7         self.stop_words.update(custom_stopwords)
8
9     def remove_short_words(self, tokens, min_length=3):
10         """Remove words shorter than min_length"""
11         return [token for token in tokens if len(token) >= min_length]
12
13     def normalize_elongated_words(self, text):
14         """Normalize elongated words (e.g., 'hellooo' -> 'hello')"""
15         pattern = re.compile(r'(\.)\1{2,}')
16         return pattern.sub(r'\1\1', text)
```

4.3 3. Complete Text Preprocessing Pipeline

```
17
18     def process(self, text, use_stemming=False, min_word_length=3):
19         # Normalize elongated words
20         text = self.normalize_elongated_words(text)
21
22         # Get tokens from parent class
23         tokens = super().process(text, use_stemming)
24
25         # Remove short words
26         tokens = self.remove_short_words(tokens, min_word_length)
27
28         return tokens
```

4.4 4. Complete Example Pipeline

```
1  def process_and_visualize_text(text):
2      # Initialize preprocessor
3      preprocessor = AdvancedTextPreprocessor()
4
5      # Process text
6      tokens = preprocessor.process(text)
7
8      # Create BOW representation
9      bow_array, features, _ = create_bow([' '.join(tokens)])
10
11     # Create visualizations
12     print("Processed tokens:", tokens)
13     print("\nBag of Words representation:")
```



4.4 4. Complete Example Pipeline

```
14     print("Features:", features)
15     print("BOW array:", bow_array)
16
17     # Generate word frequency plot
18     plot_word_frequency(' '.join(tokens))
19
20     # Generate word cloud
21     generate_wordcloud(' '.join(tokens))
22
23     return tokens, bow_array, features
24
25 # Example usage
26 sample_text = ""
```

4.4 4. Complete Example Pipeline

```
27 Natural language processing (NLP) is a subfield of linguistics, computer
    science,
28 and artificial intelligence concerned with the interactions between
    computers and
29 human language, in particular how to program computers to process and
    analyze large
30 amounts of natural language data.
31 """
32
33 tokens, bow, features = process_and_visualize_text(sample_text)
```


4.5 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


```
1 # Cache processed results for large datasets
2 from functools import lru_cache
3
4 @lru_cache(maxsize=1000)
5 def cached_preprocess(text):
6     preprocessor = TextPreprocessor()
7     return preprocessor.process(text)
```



4.5 5. Best Practices and Tips


3. Error Handling

```
1 def safe_preprocess(text):
2     try:
3         return preprocessor.process(text)
4     except Exception as e:
5         print(f"Error processing text: {e}")
6         return []
```

 Python

4. Evaluation Metrics

```
1 def evaluate_preprocessing(original_text, processed_tokens):
2     # Calculate reduction ratio
3     original_tokens = word_tokenize(original_text)
4     reduction_ratio = 1 - (len(processed_tokens) / len(original_tokens))
```

 Python

4.5 5. Best Practices and Tips

```
5
6     print(f"Original token count: {len(original_tokens)}")
7     print(f"Processed token count: {len(processed_tokens)}")
8     print(f"Reduction ratio: {reduction_ratio:.2%}")
```

5. Gensim Text Processing Guide

5.1 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)

5.2 1. Basic Gensim Usage


5.2.1 Installation and Setup

```
1 pip install gensim
2 import gensim
3 from gensim import corpora, models
```

 Python

5.2.2 Creating a Document Corpus

```
1 # Sample documents
2 documents = [
3     "The quick brown fox jumps over the lazy dog",
4     "Python is a great programming language",
5     "Text processing with Gensim is efficient",
6     "The lazy dog sleeps all day",
7 ]
```

 Python

5.2 1. Basic Gensim Usage

```
8
9 # Tokenize documents
10 def preprocess(text):
11     # Convert to lowercase and split into words
12     return text.lower().split()
13
14 # Process all documents
15 processed_docs = [preprocess(doc) for doc in documents]
16
17 # Create dictionary (maps words to IDs)
18 dictionary = corpora.Dictionary(processed_docs)
19
20 # Convert documents to bag-of-words format
```

5.2 1. Basic Gensim Usage

```
21 corpus = [dictionary.doc2bow(doc) for doc in processed_docs]
22
23 print("Dictionary:", dictionary.token2id)
24 print("\nFirst document BoW:", corpus[0])
```


5.3 2. TF-IDF with Gensim

5.3.1 Understanding TF-IDF in Gensim


TF-IDF (Term Frequency-Inverse Document Frequency) in Gensim helps identify important words by considering both their frequency in individual documents and their rarity across all documents.

Real-world example:

- 1 Consider a collection of news articles:
- 2 - Common words like "the" or "and" appear frequently but in most documents
- 3 - Topic-specific words like "cryptocurrency" might appear less frequently but in specific documents
- 4 - TF-IDF will give higher weights to topic-specific words

5.3.2 Basic TF-IDF Implementation

```
1 from gensim import models
```

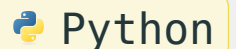
 Python

5.3 2. TF-IDF with Gensim

```
2
3 # Create TF-IDF model
4 tfidf = models.TfidfModel(corpus)
5
6 # Transform corpus to TF-IDF space
7 corpus_tfidf = tfidf[corpus]
8
9 # Print TF-IDF vectors for each document
10 for doc in corpus_tfidf:
11     print("\nTF-IDF scores:", doc)
```

5.3.3 Advanced TF-IDF Processing

```
1 def create_tfidf_model(documents):
```



5.3 2. TF-IDF with Gensim

```
2      """
3      Create a TF-IDF model from a list of documents
4      """
5      # Preprocess documents
6      processed_docs = [preprocess(doc) for doc in documents]
7
8      # Create dictionary
9      dictionary = corpora.Dictionary(processed_docs)
10
11     # Create BOW corpus
12     bow_corpus = [dictionary.doc2bow(doc) for doc in processed_docs]
13
14     # Create TF-IDF model
```

5.3 2. TF-IDF with Gensim

```
15     tfidf_model = models.TfidfModel(bow_corpus)
16
17     # Transform corpus to TF-IDF space
18     corpus_tfidf = tfidf_model[bow_corpus]
19
20     return dictionary, tfidf_model, corpus_tfidf
21
22 # Example usage
23 docs = [
24     "Machine learning is fascinating",
25     "Deep learning is a subset of machine learning",
26     "Neural networks are used in deep learning",
27     "Python is great for machine learning"
```


5.3 2. TF-IDF with Gensim

```
28 ]
29
30 dictionary, tfidf_model, corpus_tfidf = create_tfidf_model(docs)
31
32 # Print TF-IDF scores for each document
33 for i, doc in enumerate(corpus_tfidf):
34     print(f"\nDocument {i+1} TF-IDF scores:")
35     for id, score in doc:
36         print(f"Word: {dictionary[id]}, Score: {score:.4f}")
```

5.4 3. Document Similarity with TF-IDF

5.4.1 Computing Similarity Between Documents

```
1  from gensim import similarities
2
3  def compute_document_similarity(documents, query):
4      """
5      Compute similarity between a query and all documents
6      """
7      # Create TF-IDF model
8      dictionary, tfidf_model, corpus_tfidf = create_tfidf_model(documents)
9
10     # Convert query to TF-IDF space
11     query_bow = dictionary.doc2bow(preprocess(query))
12     query_tfidf = tfidf_model[query_bow]
```

 Python

5.4 3. Document Similarity with TF-IDF

```
13
14     # Initialize similarity matrix
15     index = similarities.MatrixSimilarity(corpus_tfidf)
16
17     # Compute similarities
18     sims = index[query_tfidf]
19
20     # Return sorted similarities
21     return list(enumerate(sims))
22
23 # Example usage
24 documents = [
25     "The cat sits on the mat",
```

5.4 3. Document Similarity with TF-IDF

```
26     "The dog runs in the park",
27     "Cats and dogs are pets",
28     "The mat is comfortable"
29 ]
30
31 query = "Where is the cat sitting?"
32 similarities = compute_document_similarity(documents, query)
33
34 # Print sorted similarities
35 print("\nDocument similarities to query:")
36 for doc_id, score in sorted(similarities, key=lambda x: x[1],
37                             reverse=True):
37     print(f"Document {doc_id+1}: {score:.4f} - {documents[doc_id]}")
```


5.5 4. Advanced Gensim Features

5.5.1 Building a Complete Text Analysis Pipeline

```
1  class GensimTextAnalyzer:
2      def __init__(self):
3          self.dictionary = None
4          self.tfidf_model = None
5          self.similarity_index = None
6
7      def fit(self, documents):
8          """Train the analyzer on a corpus of documents"""
9          # Preprocess documents
10         processed_docs = [preprocess(doc) for doc in documents]
11
12         # Create dictionary
```

 Python

5.5 4. Advanced Gensim Features

```
13         self.dictionary = corpora.Dictionary(processed_docs)
14
15         # Create BOW corpus
16         bow_corpus = [self.dictionary.doc2bow(doc) for doc in
17                        processed_docs]
18
19         # Create TF-IDF model
20         self.tfidf_model = models.TfidfModel(bow_corpus)
21
22         # Transform corpus to TF-IDF space
23         corpus_tfidf = self.tfidf_model[bow_corpus]
24
25         # Create similarity index
```

5.5 4. Advanced Gensim Features

```
25         self.similarity_index =  
            similarities.MatrixSimilarity(corpus_tfidf)  
26  
27     def get_similar_documents(self, query, top_n=5):  
28         """Find most similar documents to query"""  
29         # Process query  
30         query_bow = self.dictionary.doc2bow(preprocess(query))  
31         query_tfidf = self.tfidf_model[query_bow]  
32  
33         # Compute similarities  
34         sims = self.similarity_index[query_tfidf]  
35  
36         # Return top N similar documents
```

5.5 4. Advanced Gensim Features

```
37         return sorted(enumerate(sims), key=lambda x: x[1], reverse=True)
38         [:top_n]
39
40 # Example usage
41 analyzer = GensimTextAnalyzer()
42
43 documents = [
44     "Artificial intelligence is transforming industries",
45     "Machine learning models need good data",
46     "Deep learning requires powerful GPUs",
47     "Data science combines statistics and programming",
48     "Neural networks are inspired by biology"
49 ]
```

5.5 4. Advanced Gensim Features


```
50 # Train analyzer
51 analyzer.fit(documents)
52
53 # Find similar documents
54 query = "How is AI changing the world?"
55 similar_docs = analyzer.get_similar_documents(query)
56
57 print("\nMost similar documents to query:")
58 for doc_id, score in similar_docs:
59     print(f"Score: {score:.4f} - {documents[doc_id]}")
```

5.6 5. Best Practices and Tips

1. Memory Efficiency

- Use streaming corpus for large datasets
- Implement memory-efficient iterators


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

 Python

2. Preprocessing

- Remove stop words
- Apply lemmatization
- Handle special characters

```
1 def advanced_preprocess(text):
```


 Python

5.6 5. Best Practices and Tips

```
2      # Remove special characters
3      text = re.sub(r'^\w\s', '', text)
4      # Convert to lowercase
5      text = text.lower()
6      # Remove stop words
7      stop_words = set(['the', 'is', 'at', 'which'])
8      return [word for word in text.split() if word not in stop_words]
```

3. Model Persistence

```
1  # Save models
2  dictionary.save('dictionary.gensim')
3  tfidf_model.save('tfidf.gensim')
4
```

 Python

5.6 5. Best Practices and Tips

```
5 # Load models
6 dictionary = corpora.Dictionary.load('dictionary.gensim')
7 tfidf_model = models.TfidfModel.load('tfidf.gensim')
```


6. Named Entity Recognition (NER) Guide

6.1 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 such as:

- 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”)

Real-world example:

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

```
2
```

```
3 Identified entities:
```

6.1 What is Named Entity Recognition?

4 - Apple (ORGANIZATION)

5 - Tim Cook (PERSON)

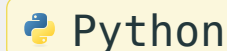
6 - California (LOCATION)

7 - September (DATE)

6.2 1. NLTK Implementation

6.2.1 Basic NER with NLTK

```
1  import nltk
2  from nltk import ne_chunk
3  from nltk import word_tokenize, pos_tag
4
5  # Download required NLTK data
6  nltk.download('averaged_perceptron_tagger')
7  nltk.download('maxent_ne_chunker')
8  nltk.download('words')
9
10 def nltk_ner(text):
11     # Tokenize and tag parts of speech
12     tokens = word_tokenize(text)
```




6.2 1. NLTK Implementation

```
13     pos_tags = pos_tag(tokens)
14
15     # Perform NER
16     named_entities = ne_chunk(pos_tags)
17
18     return named_entities
19
20 # Example usage
21 text = "John works at Google in New York."
22 entities = nltk_ner(text)
23 print(entities)
```

6.2 1. NLTK Implementation

6.2.2 Extracting Named Entities

```
1  def extract_entities(text):
2      """
3      Extract and categorize named entities from text
4      """
5      entities = {
6          'PERSON': [],
7          'ORGANIZATION': [],
8          'GPE': [], # Geo-Political Entity
9          'LOCATION': [],
10         'DATE': [],
11         'TIME': [],
12         'MONEY': [],
```

 Python

6.2 1. NLTK Implementation

```
13         'PERCENT': []
14     }
15
16     # Get named entities
17     named_entities = nltk_ner(text)
18
19     # Extract entities
20     for chunk in named_entities:
21         if hasattr(chunk, 'label'):
22             entity_name = ' '.join(c[0] for c in chunk)
23             entity_type = chunk.label()
24             if entity_type in entities:
25                 entities[entity_type].append(entity_name)
```


6.2 1. NLTK Implementation

```
26
27     return entities
28
29 # Example usage
30 text = """
31 Tim Cook, CEO of Apple Inc., announced yesterday that the company's
32 revenue
33 grew by 15% to reach $365 billion in New York City.
34 """
35 entities = extract_entities(text)
36 for entity_type, entity_list in entities.items():
37     if entity_list:
38         print(f"{entity_type}: {entity_list}")
```


6.3 2. spaCy Implementation

6.3.1 Basic NER with spaCy

```
1  import spacy
2
3  # Load English language model
4  nlp = spacy.load("en_core_web_sm")
5
6  def spacy_ner(text):
7      """
8      Perform NER using spaCy
9      """
10     # Process text
11     doc = nlp(text)
12
```

 Python

6.3 2. spaCy Implementation


```
13     # Extract entities
14     entities = [
15         {
16             'text': ent.text,
17             'label': ent.label_,
18             'start': ent.start_char,
19             'end': ent.end_char
20         }
21         for ent in doc.ents
22     ]
23
24     return entities
25
```

6.3 2. spaCy Implementation

```
26 # Example usage
27 text = "Microsoft's CEO Satya Nadella visited London last week."
28 entities = spacy_ner(text)
29
30 for entity in entities:
31     print(f"Entity: {entity['text']}")
32     print(f"Type: {entity['label']}")
33     print(f"Position: {entity['start']}-{entity['end']}\n")
```

6.3.2 Advanced spaCy NER

```
1 class NamedEntityExtractor:
2     def __init__(self, model="en_core_web_sm"):
3         self.nlp = spacy.load(model)
```

 Python

6.3 2. spaCy Implementation

```
4
5     def analyze_text(self, text):
6         """
7         Comprehensive NER analysis
8         """
9         doc = self.nlp(text)
10
11        # Extract entities with context
12        analysis = []
13        for ent in doc.ents:
14            # Get entity context (surrounding words)
15            start = max(0, ent.start - 2)
16            end = min(len(doc), ent.end + 2)
```

6.3 2. spaCy Implementation

```
17         context = doc[start:end].text
18
19         analysis.append({
20             'entity': ent.text,
21             'type': ent.label_,
22             'context': context,
23             'explanation': spacy.explain(ent.label_)
24         })
25
26     return analysis
27
28     def get_entity_statistics(self, text):
29         """
```

6.3 2. spaCy Implementation

```
30      Generate statistics about entities in text
31      """
32      doc = self.nlp(text)
33
34      stats = {
35          'total_entities': len(doc.ents),
36          'entity_types': {},
37          'entity_density': len(doc.ents) / len(doc) if len(doc) > 0
38                          else 0
39      }
40      # Count entity types
41      for ent in doc.ents:
42          stats['entity_types'][ent.label_] = \
```

6.3 2. spaCy Implementation

```
43         stats['entity_types'].get(ent.label_, 0) + 1
44
45     return stats
46
47 # Example usage
48 extractor = NamedEntityExtractor()
49
50 text = """
51 In 2024, Google and Microsoft announced a partnership worth $5 billion.
52 The deal was signed in Seattle by Sundar Pichai and Satya Nadella.
53 """
54
55 # Analyze text
```


6.3 2. spaCy Implementation

```
56 analysis = extractor.analyze_text(text)
57 print("Named Entity Analysis:")
58 for item in analysis:
59     print(f"\nEntity: {item['entity']}")
60     print(f"Type: {item['type']} ({item['explanation']})")
61     print(f"Context: \"{item['context']}\"")
62
63 # Get statistics
64 stats = extractor.get_entity_statistics(text)
65 print("\nEntity Statistics:")
66 print(f"Total entities found: {stats['total_entities']}")
67 print("Entity types distribution:", stats['entity_types'])
68 print(f"Entity density: {stats['entity_density']:.2%}")
```


6.4 3. Custom NER Training

6.4.1 Training a Custom Model with spaCy

```
1  from spacy.tokens import DocBin
2  from spacy.util import minibatch, compounding
3
4  def train_custom_ner(training_data, model=None, output_dir=None,
5                        n_iter=100):
6      """
7      Train a custom NER model
8
9      training_data format:
10     [
11         ("Text goes here", {"entities": [(0, 4, "LABEL")]}),
12         ...
13     ]
```

 Python

6.4 3. Custom NER Training

```
13     ""
14     if model is not None:
15         nlp = spacy.load(model)
16     else:
17         nlp = spacy.blank("en")
18
19     # Create or get NER component
20     if "ner" not in nlp.pipe_names:
21         ner = nlp.create_pipe("ner")
22         nlp.add_pipe("ner")
23     else:
24         ner = nlp.get_pipe("ner")
25
```

6.4 3. Custom NER Training

```
26     # Add labels
27     for _, annotations in training_data:
28         for ent in annotations.get("entities"):
29             ner.add_label(ent[2])
30
31     # Train
32     optimizer = nlp.begin_training()
33     for itn in range(n_iter):
34         losses = {}
35         batches = minibatch(training_data, size=compounding(4., 32.,
36             1.001))
37         for batch in batches:
38             texts, annotations = zip(*batch)
39             nlp.update(texts, annotations, drop=0.5, losses=losses)
```

6.4 3. Custom NER Training

```
39         print(f"Loss: {losses}")
40
41     # Save model
42     if output_dir is not None:
43         nlp.to_disk(output_dir)
44
45     return nlp
46
47 # Example training data
48 training_data = [
49     ("Apple Inc. is looking to buy U.K. startup for $1 billion",
50      {"entities": [(0, 9, "ORG"), (27, 31, "GPE"), (43, 54, "MONEY")] }),
51     ("Microsoft hired new CEO",
```


6.4 3. Custom NER Training

```
52     {"entities": [(0, 9, "ORG")]},  
53 ]  
54  
55 # Train model  
56 custom_model = train_custom_ner(training_data,  
    output_dir="custom_ner_model")
```

6.5 4. Best Practices and Tips

6.5.1 1. Text Preprocessing for NER

```
1  def preprocess_for_ner(text):
2      """
3      Preprocess text for better NER results
4      """
5      # Convert to proper case (helps with name recognition)
6      text = text.title()
7
8      # Handle special characters
9      text = re.sub(r'[^w\s.,!?-]', ' ', text)
10
11     # Normalize whitespace
12     text = ' '.join(text.split())
```


 Python

6.5 4. Best Practices and Tips

```
13
14     return text
```

6.5.2 2. Entity Validation

```
1  def validate_entities(entities, gazetteer):
2      """
3      Validate extracted entities against known lists
4      """
5      validated_entities = []
6
7      for entity in entities:
8          # Check against known entities
9          if entity['text'] in gazetteer.get(entity['label'], []):
```


 Python

6.5 4. Best Practices and Tips

```
10         entity['validated'] = True
11     else:
12         entity['validated'] = False
13     validated_entities.append(entity)
14
15     return validated_entities
```

6.5.3 3. Performance Evaluation

```
1  def evaluate_ner_model(model, test_data):
2      """
3      Evaluate NER model performance
4      """
5      true_positives = 0
```

 Python

6.5 4. Best Practices and Tips

```
6         false_positives = 0
7         false_negatives = 0
8
9         for text, annotations in test_data:
10             # Get predicted entities
11             doc = model(text)
12             predicted_entities = set([
13                 (ent.text, ent.label_) for ent in doc.ents
14             ])
15
16             # Get true entities
17             true_entities = set([
18                 (text[start:end], label)
```

6.5 4. Best Practices and Tips

```
19         for start, end, label in annotations['entities']
20     ]))
21
22     # Calculate metrics
23     true_positives += len(predicted_entities & true_entities)
24     false_positives += len(predicted_entities - true_entities)
25     false_negatives += len(true_entities - predicted_entities)
26
27     # Calculate precision, recall, F1
28     precision = true_positives / (true_positives + false_positives)
29     recall = true_positives / (true_positives + false_negatives)
30     f1 = 2 * (precision * recall) / (precision + recall)
31
```

6.5 4. Best Practices and Tips

```
32     return {  
33         'precision': precision,  
34         'recall': recall,  
35         'f1': f1  
36     }
```

6.6 5. 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

3. Relationship Extraction

- Identifying business relationships between companies
- Finding connections between people
- Mapping event locations and dates

4. Content Enrichment

6.6 5. Common Use Cases

- Adding metadata to documents
- Linking entities to knowledge bases
- Creating document summaries

Thank you for your attention!