

Natural Language Processing

An Introduction

Abdelbacet Mhamdi

2025-02-01

ISSET Bizerte

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

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 Advanced Concepts

1.4.1 Character Classes

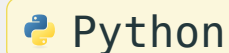
```
1 # Character class examples
2 pattern = r'[aeiou]' # Matches any vowel
3 pattern = r'[0-9]'   # Matches any digit
4 pattern = r'^0-9]'   # Matches any non-digit
```

 Python

1.4 Advanced Concepts

1.4.2 Quantifiers and Groups


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

1.4.3 Common Regex Functions in Python

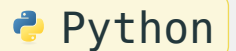
```
1  import re
2
3  text = "The price is $19.99"
4
5  # Different matching functions
6  re.search(r'\$\d+\.\d+', text)  # Finds first match
7  re.findall(r'\$\d+\.\d+', text) # Finds all matches
8  re.sub(r'\$(\d+\.\d+)', r'\1', text) # Substitution
9
10 # Splitting text
11 re.split(r'\s+', text) # Split on whitespace
```

 Python

1.5 Python Implementation

1.5.1 Basic Pattern Matching

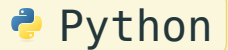
```
1  import re
2  # Simple pattern matching
3  text = "The quick brown fox jumps over the lazy dog"
4  pattern = r"fox"
5  # Search for pattern
6  match = re.search(pattern, text)
7  if match:
8      print(f"Found '{pattern}' at position: {match.start()}-{match.end()}")
9  # Find all occurrences
10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



1.5 Python Implementation

1.5.2 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  # Test cases  
5  emails = [  
6      "user@example.com", # ✓  
7      "invalid.email@com", # ✗  
8      "user.name@bizerte.r-iset.tn", # ✓  
9      "@invalid.com" # ✗  
10 ]  
11 for email in emails:  
12     print(f"{email} → {'Valid' if is_valid_email(email) else 'Invalid'}")
```




Python

1.5 Python Implementation

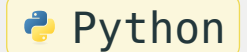
1.5.3 Phone Number Formatting

```
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:
7          pattern = r'(\d{3})(\d{3})(\d{4})'
8          formatted = re.sub(pattern, r'(\1) \2-\3', digits)
9          return formatted
10
11     return "Invalid phone number"
```

 Python

1.5 Python Implementation

```
1  # Test cases
2  numbers = [
3      "1234567890",
4      "123-456-7890",
5      "(123) 456-7890",
6      "12345"
7  ]
8
9  for number in numbers:
10     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

```
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

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

 Python


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

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

 Python

1.8 Applications

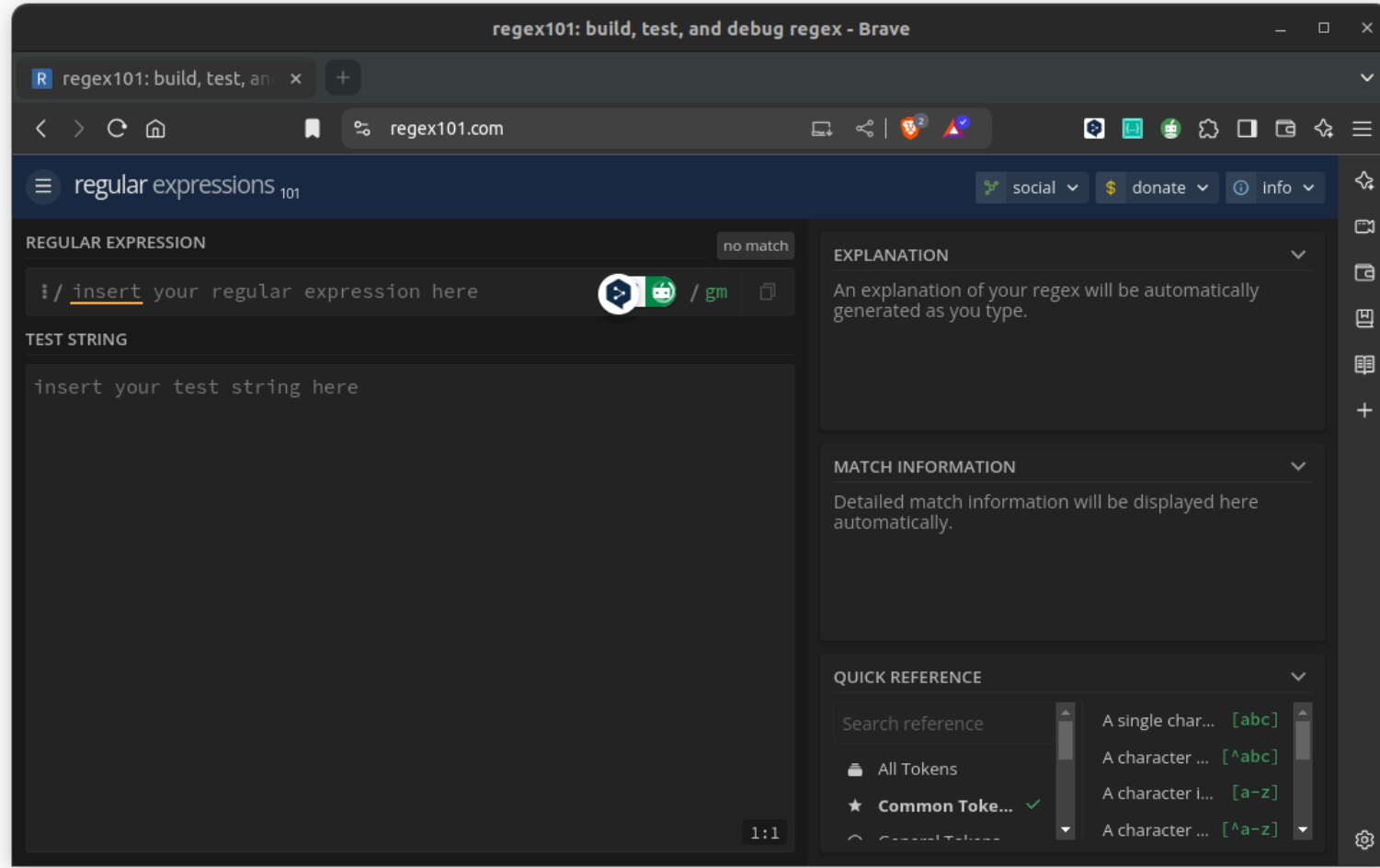


Figure 1: Build, test and debug regex patterns.

2. Text Tokenization

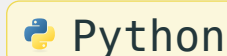
2.1 Introduction to Tokenization

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 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)
```



2.2 1. Basic Regex-based Tokenization

2.2.2 Advanced Regex Tokenization

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

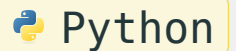
 Python

2.3 2. NLTK Tokenization

NLTK provides various tokenizers for different needs.

2.3.1 Installation and Setup

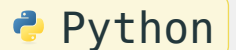
```
1 import nltk
2 nltk.download('punkt_tab') # Required for word and sentence tokenization
```



2.3 2. NLTK Tokenization

2.3.2 Word Tokenization

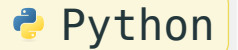
```
1 from nltk.tokenize import word_tokenize, TreebankWordTokenizer
2 text = "Don't hesitate to use NLTK's tokenizer."
3 tokens = word_tokenize(text)
4 print(tokens)
5 treebank = TreebankWordTokenizer()
6 tokens = treebank.tokenize(text)
7 print(tokens)
```



2.3 2. NLTK Tokenization

2.3.3 Sentence Tokenization

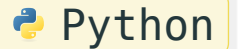
```
1 from nltk.tokenize import sent_tokenize
2 text = """Mr. Smith bought a car. He loves driving it!
3       What will he buy next? Only time will tell."""
4 sentences = sent_tokenize(text)
5 print(sentences)
```



2.3 2. NLTK Tokenization

2.3.4 Regular Expression Tokenizer

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

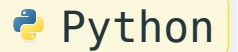


2.4 3. spaCy Tokenization

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')
```



2.4 3. spaCy Tokenization

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

 Python

2.4 3. spaCy Tokenization

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

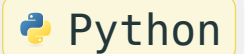
 Python

2.5 4. Polyglot Tokenization

Polyglot is especially useful for multilingual tokenization.

2.5.1 Installation and Setup

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



2.5 4. Polyglot Tokenization

2.5.2 Basic Usage


```
1  def polyglot_tokenize(text):
2      text = Text(text)
3      return list(text.words)
4
5  english_text = "Hello, world!"
6  spanish_text = "¡Hola, mundo!"
7  arabic_text = "مرحبا أيها العالم!"
8
9  for sample in [english_text, spanish_text, arabic_text]:
10     tokens = polyglot_tokenize(sample)
11     print(f"Original: {sample}")
12     print(f"Tokens: {tokens}\n")
```

 Python

2.6 5. Comparison of Different Approaches

2.6.1 Handling Special Cases




```
1 text = "Don't forget those in need!"
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 5. Comparison of Different Approaches

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 5. Comparison of Different Approaches

3. spaCy

-  Production environments
-  Fast, modern, good defaults
-  Larger memory footprint


4. Polyglot

-  Multilingual projects
-  Excellent language coverage
-  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

2.7 6. Best Practices


```
1  # Sample texts for testing different tokenization paths
2  test_texts = {
3      "english_basic": """This sentences contains simple punctuation,
4      numbers (123), and some special characters !@#$. """,
5      "english_complex": """Mr. Smith's car broke down at 3:30 P.M. "This
6      is terrible," he thought.""",
7      "multilingual": """Hello in French is Bonjour. In Spanish, hello is
8      ¡Hola!""",
9      "technical": """Python3 supports utf-8 encoding. Variables use
10     snake_case by convention."""
11 }
```



2.7 6. Best Practices

Test case 1: Basic English, no speed requirement

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


 Python

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

2.7 6. Best Practices

Test case 2: English with speed requirement

```
1 print("Test 2: English with speed optimization")
2 result2 = select_tokenizer(test_texts["english_basic"], needs_speed=True)
3 print(f"Tokens: {result2}\n")
```

 Python

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


2.7 6. Best Practices

Test case 3: Non-English text

```
1 print("Test 3: Multilingual text")
2 result3 = select_tokenizer(test_texts["multilingual"], language="fr")
3 print(f"Tokens: {result3}\n")
```


 Python

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

2.7 6. Best Practices

Test case 4: Technical text with special characters

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


 Python

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

2.7 6. Best Practices

2. Pre-processing

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

 Python

'Hello WORLD '

'!@#\$\$%^&*()_+'

'HellontWorldn Python'

'hello world'


'_'

'hello world python'

2.7 6. Best Practices

3. Handling Special Cases

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

 Python

['They', "can't", 'believe', "it's", 'already', 'Friday']

['They', 'can', 'not', 'believe', 'it', 'is', 'already', 'Friday']

3. Text Processing and Visualization

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.

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 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 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 3. Lemmatization

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

1 am, are, is → be

2 running, ran, runs → run

3 better, best → good

4 wolves → wolf

3.1 Fundamental Concepts and Definitions

3.1.2.4 4. Stemming

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

1 running → run

2 fishing → fish

3 completely → complet

4 authentication → authent

3.1 Fundamental Concepts and Definitions

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



Create BoW

[text_processing/create_bow.py](#)

3.1 Fundamental Concepts and Definitions

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



Create TF-IDF

[text_processing/create_tfidf.py](#)

3.2 Text Visualization Concepts

3.2.1 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.

3.2 Text Visualization Concepts



```
Generate WordCloud
```

```
text_processing/generate_wordcloud.py
```


3.2 Text Visualization Concepts

3.2.2 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 Complete Pipeline Example with Real Text

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!!!"

3.3 Complete Pipeline Example with Real Text

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"]

3.3 Complete Pipeline Example with Real Text



Plot Word Frequency

[text_processing/plot_word_frequency.py](#)

3.3 Complete Pipeline Example with Real Text



Visualization with Seaborn

[text_processing/visualization_with_seaborn.py](#)

3.3 Complete Pipeline Example with Real Text



Advanced Pipeline with Custom Features

[text_processing/advanced_pipeline_with_custom_features.py](#)

3.3 Complete Pipeline Example with Real Text



Complete Example Pipeline

[text_processing/complete_example_pipeline.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

3.5 Best Practices and Tips

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)
```

 Python

3.5 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

3.5 Best Practices and Tips

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))  
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%}")
```

 Python

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 1. Basic Gensim Usage

4.2.1 Installation and Setup

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

 Python

4.2 1. Basic Gensim Usage



Creating a Document Corpus

[gensim_text_processing/creating_a_document_corpus.py](#)

4.3 2. TF-IDF with Gensim

4.3.1 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.


Consider a collection of news articles:

- Common words like “the” or “and” appear frequently but in most documents
- Topic-specific words like “cryptocurrency” might appear less frequently but in specific documents
- TF-IDF will give higher weights to topic-specific words

4.3 2. TF-IDF with Gensim

4.3.2 Basic TF-IDF Implementation

```
1  from gensim import models
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)
```

 Python

4.3.3 Advanced TF-IDF Usage

4.3 2. TF-IDF with Gensim



TF-IDF With Gensim

[gensim_text_processing/tfidf_with_gensim.py](#)

4.3 2. TF-IDF with Gensim



Computing Similarity Between Documents

[gensim_text_processing/computing_similarity_between_documents.py](#)

4.3 2. TF-IDF with Gensim



Building a Complete Text Analysis Pipeline


[gensim_text_processing/building_a_complete_text_analysis_pipeline.py](#)

4.4 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

4.4 Best Practices and Tips

2. Preprocessing

- Remove stop words
- Apply lemmatization
- Handle special characters

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


 Python

4.4 Best Practices and Tips

3. Model Persistence


Save models

```
1 dictionary.save('dictionary.gensim')  
2 tfidf_model.save('tfidf.gensim')
```

 Python

Load models

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

 Python

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”)

5.1 What is Named Entity Recognition?

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

Identified entities:

- Apple (ORGANIZATION)
- Tim Cook (PERSON)
- California (LOCATION)
- September (DATE)

5.2 NLTK Implementation



Basic NER with NLTK

[ner/ner_with_nltk.py](#)

5.2 NLTK Implementation



Extracting Named Entities

[ner/extracting_named_entities.py](#)

5.3 spaCy Implementation



Basic NER with spaCy

[ner/ner_with_spacy.py](#)

5.3 spaCy Implementation



Advanced spaCy NER

[ner/advanced_spacy_ner.py](#)

5.3 spaCy Implementation



Custom NER Training

[ner/custom_ner_training.py](#)

5.4 Best Practices and Tips



Text Preprocessing for NER

[ner/text_preprocessing_for_ner.py](#)

5.4 Best Practices and Tips



Entity Validation

[ner/entity_validation.py](#)

5.4 Best Practices and Tips



Performance Evaluation

[ner/performance_evaluation.py](#)

5.5 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

5.5 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!