Natural Language Processing

An Introduction

Abdelbacet Mhamdi

2025-05-08

ISET Bizerte

Outline

1. Introduction to Regular Expressions (Regex)

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.

1.2.1 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"
+	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. **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;
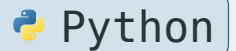   - URLs.

1.4.1 Character Classes

```Python
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
```

1.4.2 Quantifiers and Groups

```python
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
```

1.4.3 Common Regex Functions in 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
```
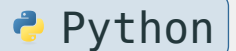
1.5 Python Implementation

1.5.1 Basic Pattern Matching

```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}")
```
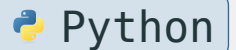
1.5.2 Email Validation Example

```python
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'}")
```

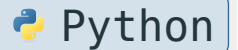🐍 Python

1.5.3 Phone Number Formatting

```python
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  # 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

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

2. **Compile Frequently Used Patterns**

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

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

```python
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 '✗'}")
```

> "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. **Basic Pattern Matching**

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

2. **Data Extraction**

```python
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)
```

3. **Password Validation**

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

🐍 Python

1.8 Applications
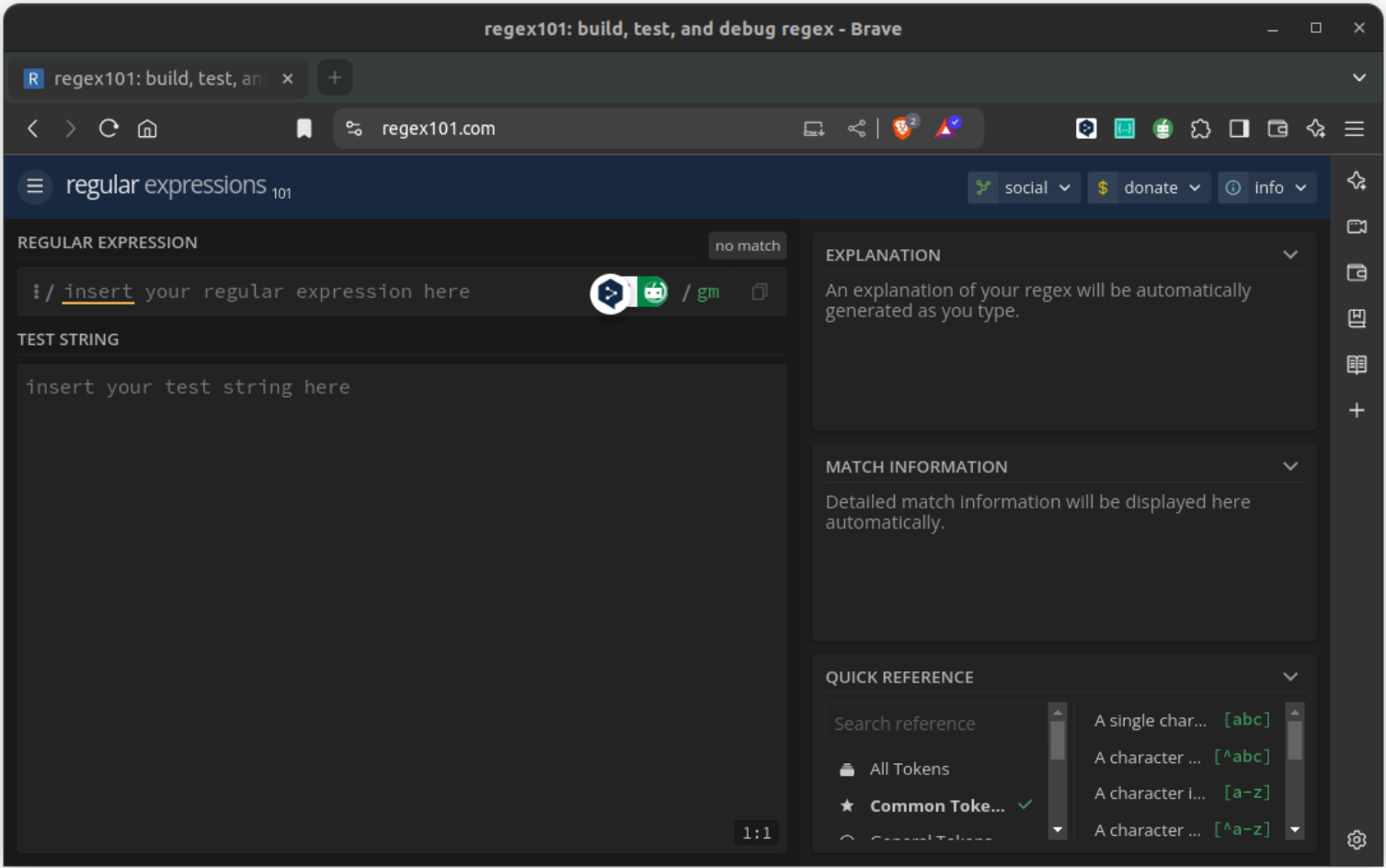

Figure 1: Build, test and debug regex patterns.

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)

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,}$")

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})")

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,})$")

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

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
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']

2.2.2 Advanced Regex Tokenization

```python
1  def advanced_tokenize(text):
2      pattern = r"""
3          [a-zA-Z]+ | (?<=\$)\d+(?:\.\d+)? | \d+(?:\.\d+)?(?=%)
4      """
5      tokens = re.findall(pattern, text, re.VERBOSE)
6      return tokens
7
8  text = "The price is $19.99, and the discount is 15%!"
9  print(advanced_tokenize(text))
```
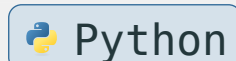
['The', 'price', 'is', '19.99', 'and', 'the', 'discount', 'is', '15']

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

2.3.2 Word Tokenization

```python
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  # Output: ['Do', "n't", 'hesitate', 'to', 'use', 'NLTK', "'s",
   'tokenizer', '.']
6  treebank = TreebankWordTokenizer()
7  tokens = treebank.tokenize(text)
8  print(tokens)
```

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

2.3.3 Sentence Tokenization

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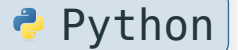
['Mr. Smith bought a car.', 'He loves driving it!', 'What will he buy next?', 'Only time will tell.']

2.3.4 Regular Expression Tokenizer

```python
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)
```
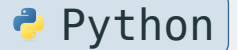
['Hello', ',', 'World', '!', 'How', '''', 's', 'it', 'going', '?']

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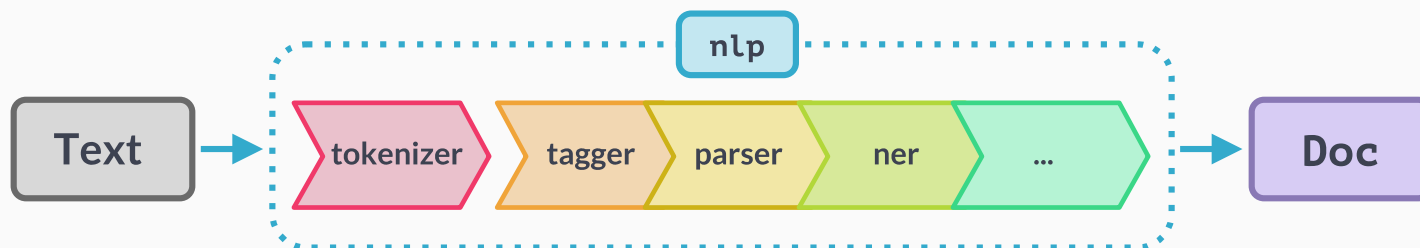
2.4.1 Installation and Usage

```bash
1 python -m spacy download en_core_web_sm
```
Bash

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

---

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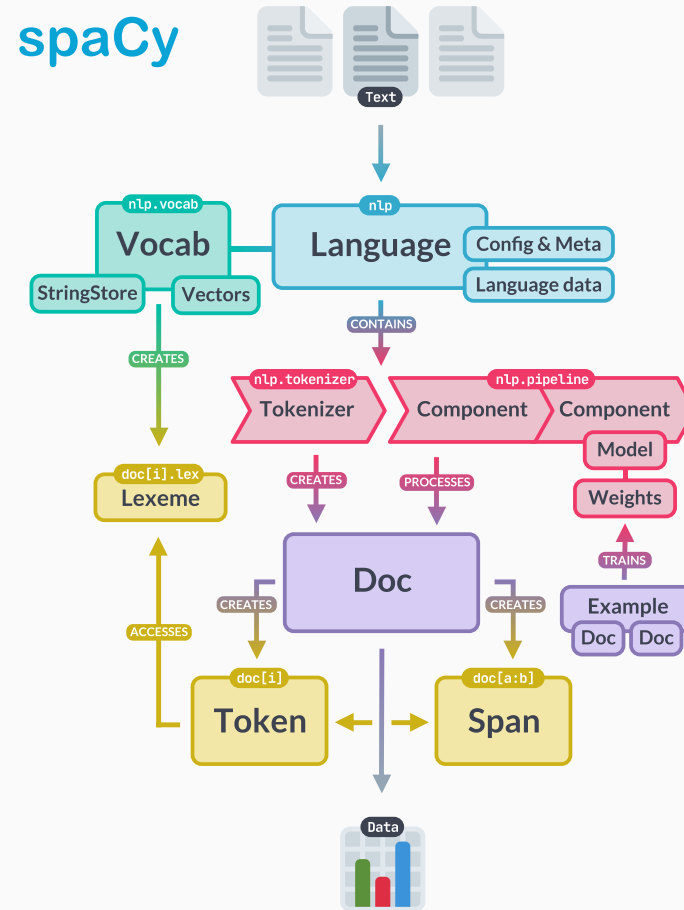
```python
1 def spacy_tokenize(text):
2     doc = nlp(text)
3     return [token.text for token in doc]
4
5 text = "spaCy's tokenizer is industrial-strength!"
6 tokens = spacy_tokenize(text)
7 print(tokens)
```

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

2.4.2 Tokenization with Linguistic Features

```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}"
    )
```

```
13  """
14  Token: Apple       POS: PROPN    Lemma: Apple     Is punctuation? False
15  Token: 's          POS: PART     Lemma: 's        Is punctuation? False
16  Token: stock       POS: NOUN     Lemma: stock     Is punctuation? False
17  Token: price       POS: NOUN     Lemma: price     Is punctuation? False
18  Token: rose        POS: VERB     Lemma: rise      Is punctuation? False
19  Token: 5.2         POS: NUM      Lemma: 5.2       Is punctuation? False
20  Token: %           POS: NOUN     Lemma: %         Is punctuation? True
21  Token: to          POS: ADP      Lemma: to        Is punctuation? False
22  Token: $           POS: SYM      Lemma: $         Is punctuation? False
23  Token: 200         POS: NUM      Lemma: 200       Is punctuation? False
24  ...
25  """
```

```python
1   def analyze_tokens(text):
2       """
3       Analyze the tokens in a text using spaCy.
4       """
5       doc = nlp(text)
6       for token in doc:
7           print(f"""
8           Text: {token.text}
9           Lemma: {token.lemma_}
10          POS: {token.pos_}
11          Is stop word: {token.is_stop}
12          """)
13
```

```
14  text = "Running quickly through the forest"
15  analyze_tokens(text)
16  """
17  Text: Running
18  Lemma: run
19  POS: VERB
20  Is stop word: False
21
22  Text: quickly
23  Lemma: quickly
24  POS: ADV
25  Is stop word: False
26  """
```

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

```bash
1 pip install polyglot
```

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

2.5.2 Basic Usage

```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
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', '!']
```

2.6.2 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

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

Test case 1: Basic English, no speed requirement

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

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

Test case 2: Non-English text

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

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

Test case 3: English with speed requirement

```python
1  print("Test 3: English with speed optimization")
2  result3 = select_tokenizer(test_texts["english_complex"],
   needs_speed=True)
3  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
1  print("Test 4: Technical text")
2  result4 = select_tokenizer(test_texts["technical"])
3  print(f"Tokens: {result4}")
```

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

2. **Pre-processing**

```python
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
```

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

'hello world'   '_'   'hello world python'

3. **Handling Special Cases**

```python
1  def handle_special_cases(tokens):
2      """Expand contractions in a token list."""
3      contractions = {"n't": "not", "'s": "is", "'re": "are"}  # Added
       common cases
4      expanded_tokens = []
5      for token in tokens:
6          expanded_tokens.append(contractions.get(token, token)) #
           dict.get(key, default)
7      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"*

3.1.2 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"]*

3.1.2.2 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"*

3.1.2.3 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
```

3.1.2.4 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
```

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!!!"

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"]

3.1.3 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]

- **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:

Doc 1 · Doc 2 = (1x1) + (1x1) + (1x0) + (1x1) + (0x1) + (0x0) + (0x0) + (0x0) + (0x0) = 3

Doc 1 · Doc 3 = (1x0) + (1x0) + (1x1) + (1x2) + (0x1) + (0x1) + (0x1) + (0x0) + (0x1) = 3

Doc 2 · Doc 3 = (1x0) + (1x0) + (0x1) + (1x2) + (1x1) + (0x1) + (0x1) + (0x1) + (0x1) = 3

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.

3.1.4.1 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:

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

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:

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

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", "has", "a","is", "the", etc.):

For **Document 1** (6 terms):

- $\text{TF}(\text{smartphone}) = \frac{1}{6} \approx 0.167$
- $\text{TF}(\text{great}) = \frac{1}{6} \approx 0.167$
- $\text{TF}(\text{camera}) = \frac{1}{6} \approx 0.167$
- $\text{TF}(\text{long}) = \frac{1}{6} \approx 0.167$
- $\text{TF}(\text{battery}) = \frac{1}{6} \approx 0.167$
- $\text{TF}(\text{life}) = \frac{1}{6} \approx 0.167$

For **Document 2** (5 terms):

- $\text{TF}(\text{camera}) = \frac{1}{5} = 0.2$
- $\text{TF}(\text{laptop}) = \frac{1}{5} = 0.2$
- $\text{TF}(\text{decent}) = \frac{1}{5} = 0.2$
- $\text{TF}(\text{keyboard}) = \frac{1}{5} = 0.2$
- $\text{TF}(\text{excellent}) = \frac{1}{5} = 0.2$

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

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

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

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

Calculate IDF for each term across all documents:

Total documents (N) = 4

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

- $\text{IDF}(\text{decent}) = \log\left(\frac{4}{1}\right) \approx 0.602$
- $\text{IDF}(\text{keyboard}) = \log\left(\frac{4}{2}\right) \approx 0.301$
- $\text{IDF}(\text{excellent}) = \log\left(\frac{4}{2}\right) \approx 0.301$
- $\text{IDF}(\text{crucial}) = \log\left(\frac{4}{1}\right) \approx 0.602$
- $\text{IDF}(\text{fast}) = \log\left(\frac{4}{1}\right) \approx 0.602$
- $\text{IDF}(\text{processor}) = \log\left(\frac{4}{1}\right) \approx 0.602$

Calculate TF-IDF for each term in each document:

For **Document 1**:
- TF-IDF$(\text{smartphone}, d_1) \approx 0.05$
- TF-IDF$(\text{great}, d_1) \approx 0.1$
- TF-IDF$(\text{camera}, d_1) \approx 0.05$
- TF-IDF$(\text{long}, d_1) \approx 0.1$
- TF-IDF$(\text{battery}, d_1) \approx 0.05$
- TF-IDF$(\text{life}, d_1) \approx 0.05$

For **Document 2**:
- TF-IDF$(\text{camera}, d_2) \approx 0.06$
- TF-IDF$(\text{laptop}, d_2) \approx 0.06$
- TF-IDF$(\text{decent}, d_2) \approx 0.12$
- TF-IDF$(\text{keyboard}, d_2) \approx 0.06$
- TF-IDF$(\text{excellent}, d_2) \approx 0.06$

For **Document 3**:

- $\text{TF-IDF}(\text{battery}, d_3) \approx 0.1$
- $\text{TF-IDF}(\text{life}, d_3) \approx 0.1$
- $\text{TF-IDF}(\text{crucial}, d_3) \approx 0.2$
- $\text{TF-IDF}(\text{smartphone}, d_3) \approx 0.1$

For **Document 4**:

- $\text{TF-IDF}(\text{laptop}, d_4) \approx 0.06$
- $\text{TF-IDF}(\text{fast}, d_4) \approx 0.12$
- $\text{TF-IDF}(\text{processor}, d_4) \approx 0.12$
- $\text{TF-IDF}(\text{excellent}, d_4) \approx 0.06$
- $\text{TF-IDF}(\text{keyboard}, d_4) \approx 0.06$

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

Document	*Highest TF-IDF Terms*
Document 1	"great" (0.1) and "long" (0.1)
Document 2	"decent" (0.12)
Document 3	"crucial" (0.2)
Document 4	"fast" (0.12) and "processor" (0.12)

An n-gram is a contiguous sequence of n items extracted from a text or speech sample. These items can be characters, words, syllables, or other linguistic units depending on the application context.

Unigrams (1-grams) Individual items *(e.g., single words like "computer")*

Bigrams (2-grams) Pairs of consecutive items *(e.g., "natural language")*

Trigrams (3-grams) Three consecutive items *(e.g., "machine learning algorithm")*

4-grams, 5-grams etc. Longer sequences following the same pattern

N-gram models operate on the **Markov** assumption that the probability of a word depends primarily on the previous $n - 1$ words, creating a probabilistic language model. While larger n values capture more context, they also increase computational complexity and data sparsity challenges.

N-grams serve as fundamental building blocks in various NLP applications:
- Text prediction and auto-completion systems
- Machine translation
- Speech recognition
- Information retrieval and search engines
- Text classification and sentiment analysis
- Spelling correction
- Computational biology (for analyzing DNA sequences)

"The cat sat on the mat."
1. Identify all unigrams and their frequencies
2. List all possible bigrams
3. List all possible trigrams
4. Calculate the probability of "sat" following "cat"
5. If we add the sentence "The cat ate the fish," calculate the probability of "the" being followed by "mat"

1. Unigrams and frequencies

- "The": 1
- "cat": 1
- "sat": 1
- "on": 1
- "the": 1
- "mat": 1

2. All possible bigrams:

- "The cat"
- "cat sat"
- "sat on"
- "on the"
- "the mat"

3. All possible trigrams:

- "The cat sat"
- "cat sat on"
- "sat on the"
- "on the mat"

If treating "The" and "the" as case-insensitive, then "the" would have a frequency of 2.

4. Probability of "sat" following "cat":

$$P(\text{sat}|\text{cat}) = \frac{\text{Count}(\text{cat sat})}{\text{Count}(\text{cat})} = \frac{1}{1} = 1.0$$

5. With added sentence "The cat ate the fish":

Combined text: "The cat sat on the mat. The cat ate the fish."

Updated frequencies for "the": 2 (if case-insensitive, counting "The" and "the") Count of "the mat": 1

$$P(\text{mat}|\text{the}) = \frac{\text{Count}(\text{the mat})}{\text{Count}(\text{the})} = \frac{1}{2} = 0.5$$

Task 7:

Text sample: "The dog ran fast. The dog jumped high. A cat ran past the dog. The cat was quick."

1. Count all unigrams, bigrams, and trigrams in the text (treat the text as case-insensitive)
2. Calculate the following probabilities:
   - P(ran|dog): Probability that "ran" follows "dog"
   - P(cat|The): Probability that "cat" follows "The"
   - P(high|jumped): Probability that "high" follows "jumped"
   - Using the bigram model, identify the most likely word to follow "The dog"
   - Using the bigram model, calculate the probability of the sequence "The cat ran"

1. N-gram Counts (treating text as case-insensitive)

Unigrams (1-grams):	Bigrams (2-grams):	Trigrams (3-grams):
• "the": 4	• "the dog": 3	• "the dog ran": 1
• "dog": 3	• "dog ran": 1	• "dog ran fast": 1
• "ran": 2	• "ran fast": 1	• "ran fast the": 1
• "fast": 1	• "fast the": 1	• "fast the dog": 1
• "jumped": 1	• "dog jumped": 1	• "the dog jumped": 1
• "high": 1	• "jumped high": 1	• "dog jumped high": 1
• "a": 1	• "high a": 1	• "jumped high a": 1
• "cat": 2	• "a cat": 1	• "high a cat": 1
• "past": 1	• "cat ran": 1	• "a cat ran": 1
• "was": 1	• "ran past": 1	• "cat ran past": 1
• "quick": 1	• "past the": 1	• "ran past the": 1
	• "the cat": 1	• "past the dog": 1

- "cat was": 1
- "was quick": 1

- "the dog the": 1
- "the cat was": 1
- "cat was quick": 1

2. Probability Calculations

$$P(\text{ran}|\text{dog}) = \frac{\text{Count(dog ran)}}{\text{Count(dog)}} = \frac{1}{3} = 0.33$$

$$P(\text{cat}|\text{The}) = \frac{\text{Count(the cat)}}{\text{Count(the)}} = \frac{1}{4} = 0.25$$

$$P(\text{high}|\text{jumped}) = \frac{\text{Count(jumped high)}}{\text{Count(jumped)}} = \frac{1}{1} = 1.0$$

To determine most likely word to follow "The dog" (using bigram model), we check the counts of "the dog X" for all X:

- Count("the dog ran") = 1
- Count("the dog jumped") = 1
- Count("the dog the") = 1

So there's a tie between with equal probability of 0.333 each $(\frac{1}{3})$.

Probability of the sequence "The cat ran" Using the bigram model and chain rule:

$$P(\text{The cat ran}) = P(\text{the}) \times P(\text{cat|the}) \times P(\text{ran|cat}) = \left(\frac{4}{18}\right) \times \left(\frac{1}{4}\right) \times \left(\frac{1}{2}\right) = 0.028$$

Where:
- $P(\text{the}) = \frac{4}{18}$ (frequency of "the" out of all words)
- $P(\text{cat|the}) = \frac{1}{4}$ (frequency of "the cat" out of all bigrams starting with "the")
- $P(\text{ran|cat}) = \frac{1}{2}$ (frequency of "cat ran" out of all bigrams starting with "cat")

3. Handling Unseen N-grams (Bonus)

To handle unseen n-grams, we would use smoothing techniques such as:

Add-one (Laplace) smoothing Add 1 to all counts to ensure no zero probabilities

$$P(w_2|w_1) = \frac{\text{Count}(w_1, w_2) + 1}{\text{Count}(w_1) + V}$$

Where V is the vocabulary size (number of unique words)

Add-k smoothing Add a small k (0 < k < 1) to all counts

Good-Turing smoothing Estimate the probability of unseen events based on the frequency of rare events

Backoff models If a higher-order n-gram isn't observed, fall back to a lower-order n-gram

Interpolation Combine probabilities from different order n-grams with weights

Text Processing

Marimo/text_processing.py

3.4.1 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.

3.4.2 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.

Text Visualization

Marimo/text_visualization.py

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

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. Word Embedding

Word embedding is a technique that represents words as dense vectors of real numbers in a continuous vector space. Words with similar meanings are placed close to each other in this space. It captures semantic relationships between words. For example, the embedding can learn that:

```
1  king - man + woman ≈ queen
```

Dimension Reduction Unlike sparse one-hot encodings, embeddings use lower-dimensional vectors *(e.g., 100-300 dimensions)*

Context Awareness Embeddings are learned from large text corpora, so they reflect word usage and context.

Word Embedding Relationships (King-Queen-Man-Woman)

Key:
— Gender relationship
--- Royalty relationship
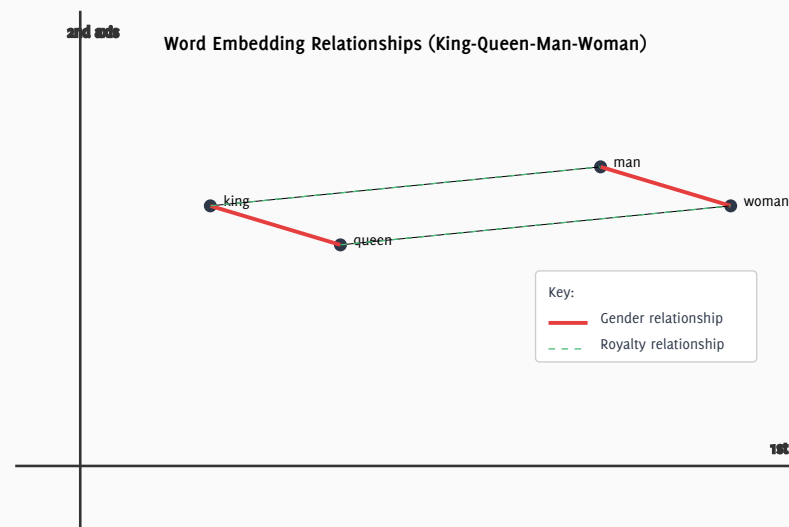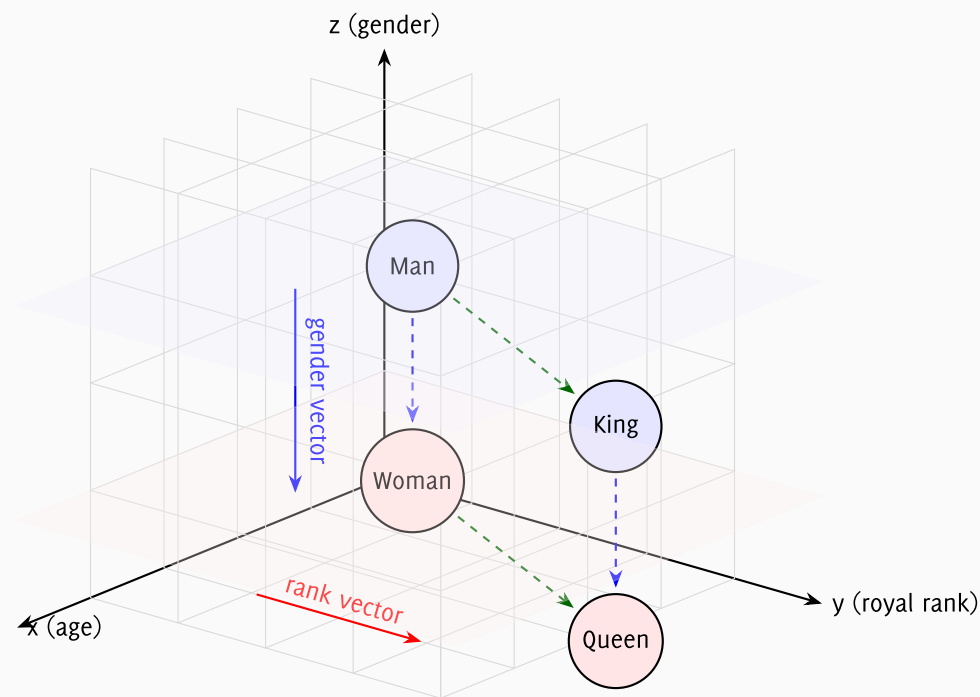
Given these words:

```
1  ["king", "queen", "man", "woman"]
```

They might be embedded as:

```
1  king → [0.7, 0.1, ...]
2  queen → [0.68, 0.11, ...]
3  man → [0.6, -0.2, ...]
4  woman → [0.58, -0.19, ...]
```

These vectors preserve semantic relationships and can be used to perform reasoning, classification, translation, and even analogy tasks in a meaningful way. For example, if we want to find the word that is to "Italy" as "Paris" is to "France", we can use the embedding vectors:

```
1  ["Paris", "France", "Rome", "Italy"]
```

In a well-trained word embedding space, the model learns not just word meanings but also analogical relationships. It can capture that:

```
1  Paris - France + Italy ≈ Rome
```
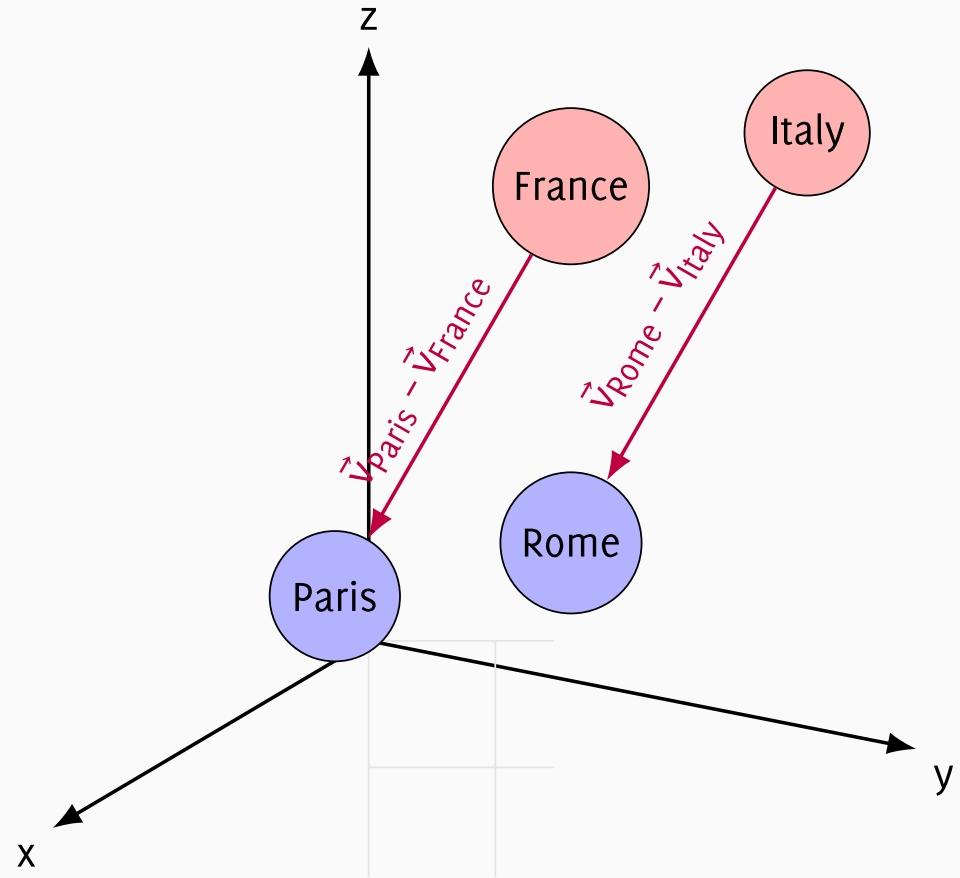
This means that the difference between "Paris" and "France" is similar to the difference between "Rome" and "Italy" — they are both capital-country pairs.

```
1  Paris  → [0.5, 0.1, 0.6, ...]
2  France → [0.4, 0.2, 0.5, ...]
3  Rome   → [0.52, 0.12, 0.62, ...]
4  Italy  → [0.42, 0.22, 0.52, ...]
```

If we compute:

```
1  Paris - France + Italy
2      ≈ [0.5-0.4+0.42,
3         0.1-0.2+0.22,
4         0.6-0.5+0.52]
5      = [0.52, 0.12, 0.62]
6      ≈ Rome
```

Models like `Word2Vec`, `GloVe`, and `FastText` are commonly used to generate these embeddings.

Word2Vec Introduced by Google; Predicts surrounding words given a target word (CBOW) or vice versa (Skip-gram).

GloVe (Global Vectors) Developed by Stanford; leverages global word co-occurrence statistics.

FastText Developed by Facebook; includes subword information to handle rare and morphologically complex words.

4.1.1 Word2Vec

Two variants

1. **Skip-Gram**: Predicts context words from a target word.
2. **CBOW (Continuous Bag-of-Words)**: Predicts target word from context.

- Sliding window captures local context relationships.
- Captures semantic/grammatical analogies *(e.g., $king - man + woman \approx queen$).*
- Lightweight and scalable for large datasets.
- Excels at contextual similarity tasks *(e.g., word clustering).*
- Pre-trained embeddings *(e.g., Google News, 300D)* for plug-and-play NLP.
- Foundational for advanced models *(RNNs, Transformers).*

4.1.2 GloVe (Global Vectors)

- Hybrid approach: Combines global corpus co-occurrence statistics with local context patterns.
- Weighted loss function reduces impact of rare/frequent word pairs.
- Precomputed co-occurrence matrix speeds up training.
- Captures linear relationships (e.g., `king - man + woman ≈ queen`).
- Pre-trained embeddings (e.g., Wikipedia, Common Crawl, 50D-300D).
- Outperforms `Word2Vec` on semantic similarity tasks.
- Scalable, interpretable, and efficient for large corpora.

4.1.3 FastText

- Extends `Word2Vec` by incorporating *subword information* (character n-grams).
- Represents words as a bag of character n-grams (e.g., "apple" → "ap", "app", "pple").
- Uses **skip-gram** or **CBOW** with subword embeddings.
- Handles rare/out-of-vocabulary (**OOV**) words via subword components.
- Robust for morphologically rich languages *(e.g., Turkish, Finnish)*.
- Pre-trained vectors for 157+ languages *(e.g., Wikipedia, Common Crawl)*.
- Better at handling typos, slang, and rare words.
- Strong performance in text classification and low-resource languages.
- Beats `Word2Vec` in semantic tasks
- Scalable, interpretable, integrates into NLP pipelines

4.2.1 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")

4.2.2 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`

.

```python
1  import spacy
2  from spacy import displacy
3
4  nlp = spacy.load('en_core_web_sm')
5
6  doc = nlp("Apple CEO Tim Cook announced new iPhone models in California
   last September.")
7
8  displacy.serve(doc, style='ent')
```

Extracting Named Entities

Marimo/extracting_named_entity.py

4.2.3 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 be-tween 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

4.3.1 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
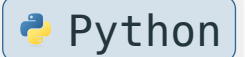- Topic modeling capabilities (LSA, LDA)

4.3.2 Basic Gensim Usage

4.3.2.1 Installation and Usage

```Bash
1  pip install gensim
```

```Python
1  import gensim
2  from gensim import corpora, models
```

Building a Complete Text Analysis Pipeline

Marimo/gensim_text_processing.py

4.3.3 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())
```

3. **Model Persistence**

Save models

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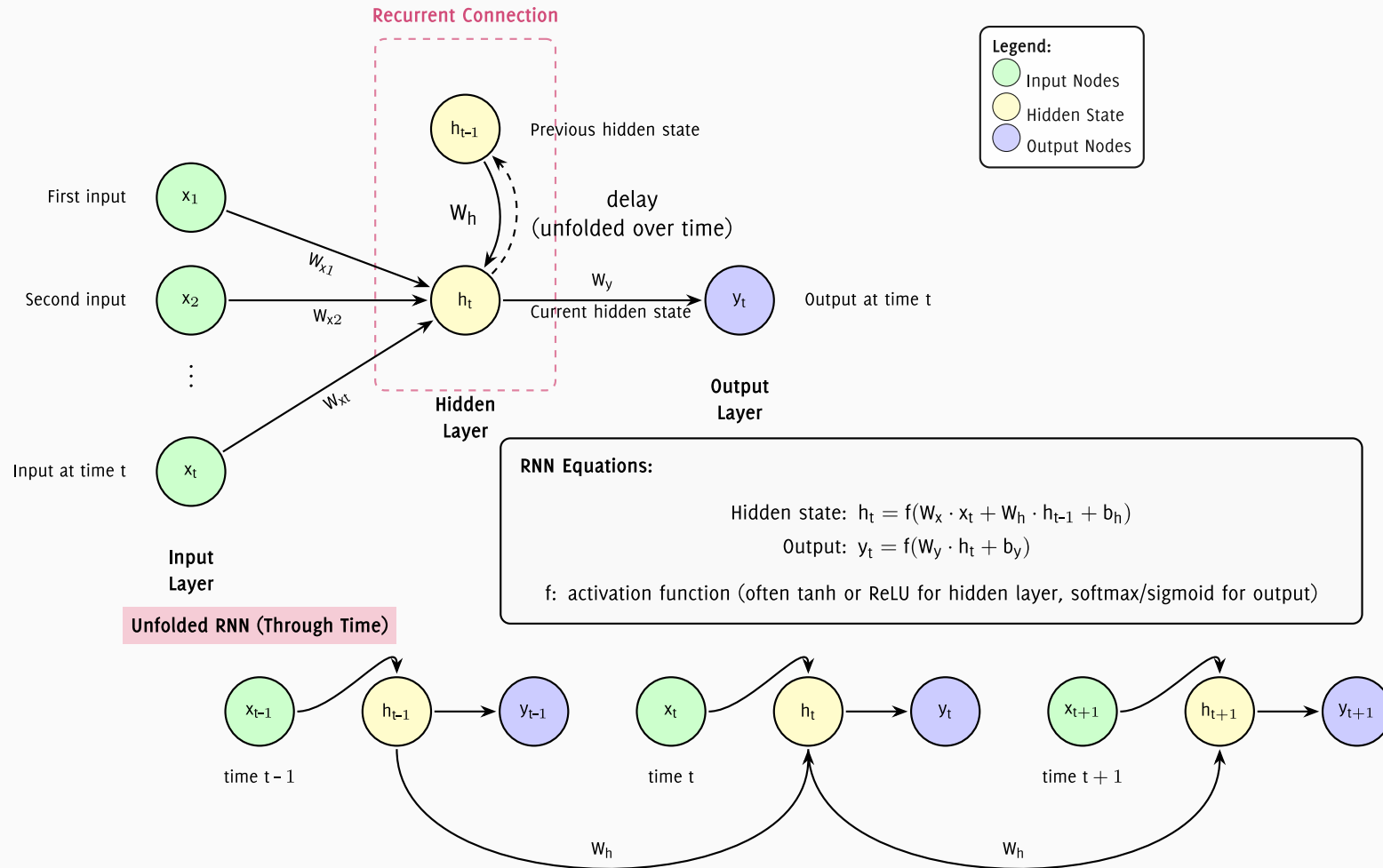
Load models

```python
1  dictionary = corpora.Dictionary.load('dictionary.gensim')
2  tfidf_model = models.TfidfModel.load('tfidf.gensim')
```
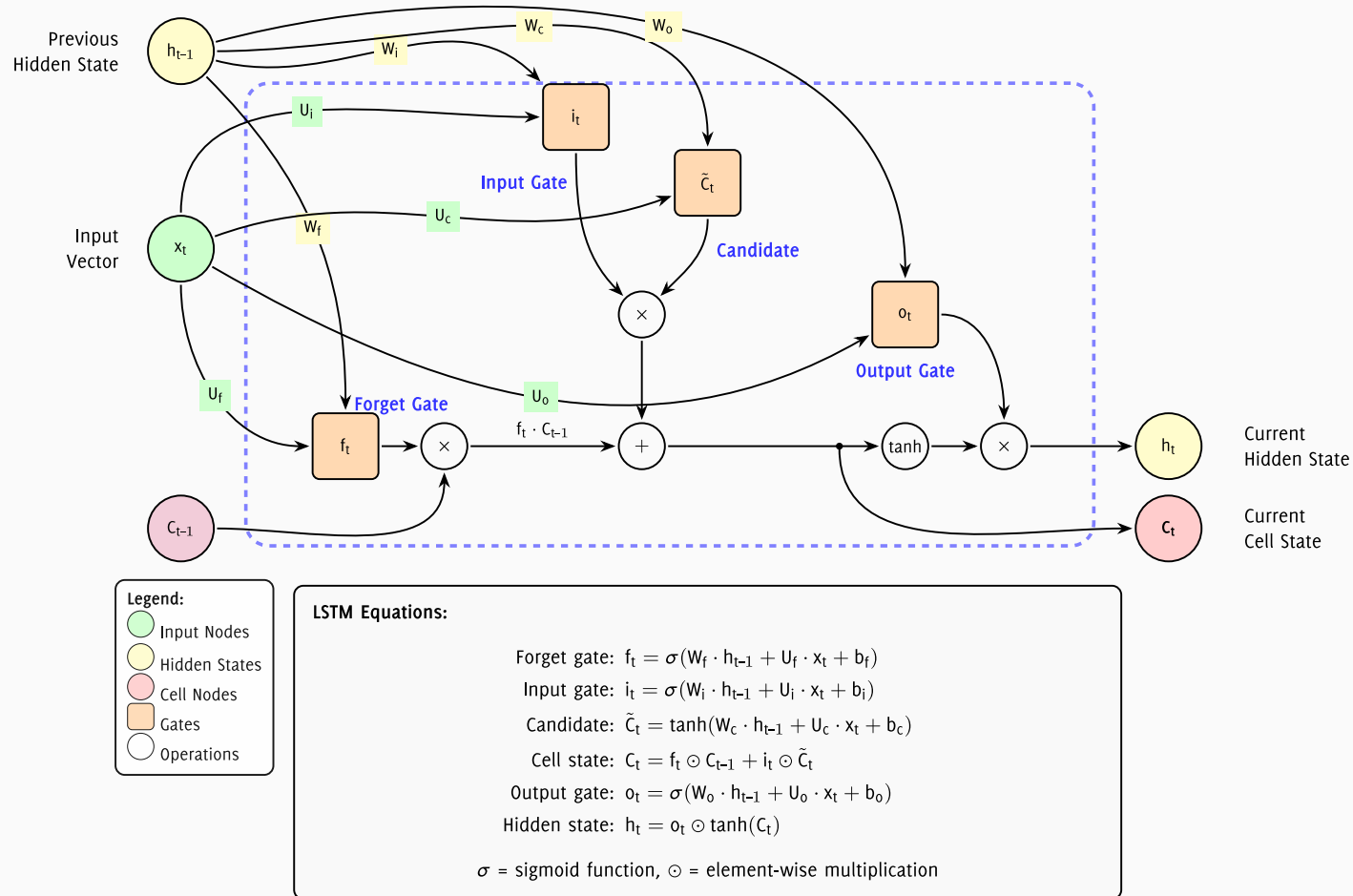
5. Deep Learning for NLP

- Processes sequential data
- Shares parameters across time
- Memory of previous inputs
- Useful for text, speech, time series
- Struggles with long-term dependencies
- Prone to vanishing/exploding gradients
- Suitable for tasks like sentiment analysis, text classification, and sequence labeling.

- Extends **RNN** by adding a cell state.
- Uses gates *(input, forget, output)* to control information flow.
- Better at capturing long-term dependencies.
- Uses a cell state to carry information across time steps.
- Can be slower to train due to the complexity of the architecture.
- Can be computationally expensive for long sequences.

5.3 Gated Recurrent Unit (GRU)

Legend:
- Input Nodes
- Hidden States
- Gates
- Operations
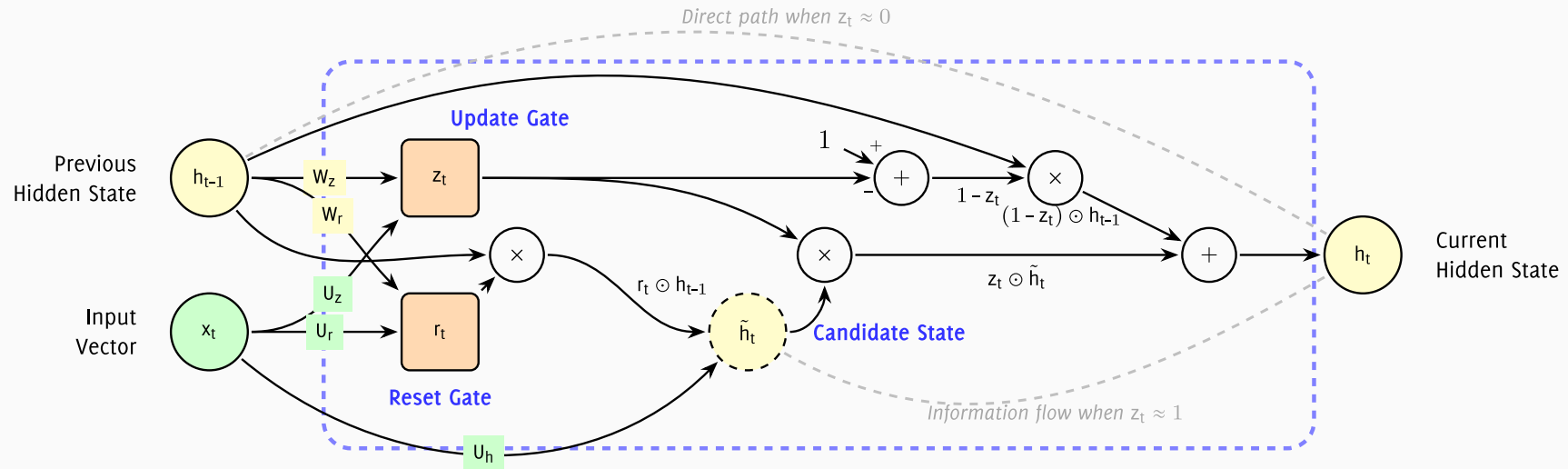
GRU Equations:

Update Gate: $z_t = \sigma(W_z \cdot h_{t-1} + U_z \cdot x_t + b_z)$

Reset Gate: $r_t = \sigma(W_r \cdot h_{t-1} + U_r \cdot x_t + b_r)$

Candidate State: $\tilde{h}_t = \tanh(W_h \cdot (r_t \odot h_{t-1}) + U_h \cdot x_t + b_h)$

Hidden State: $h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$

σ = sigmoid function, $\odot$ = element-wise multiplication

- A simplified version of **LSTM**.
- Uses update gate (z_t) to control information flow
- Uses reset gate (r_t) to control the hidden state
- Combines input and forget gates into a single update gate.
- Less complex than **LSTM**, making it easier to implement and understand.
- Can be faster to train due to fewer parameters.

RNN vs LSTM vs GRU

- **RNNs** are suitable for simpler tasks with shorter sequences, but struggle with long-term dependencies.
- **LSTMs** are more complex but better at capturing long-term dependencies.
- **GRUs** are a middle ground, offering a simpler architecture with comparable performance to **LSTMs**.
- **LSTMs** and **GRUs** are often used in applications where long-term dependencies are crucial, such as language modeling and text generation.

The **Transformer** is a neural network architecture that has become the foundation of most modern NLP models. Introduced in 2017 by Vaswani et al. in the paper **"Attention Is All You Need"**[1], it replaced earlier sequence models like RNNs and LSTMs by introducing a fully attention-based mechanism.

Transformers have significantly advanced the field of NLP by enabling models to learn deeper context, handle longer dependencies, and train faster due to parallel computation. Unlike RNNs, which process input sequentially, **Transformers** can process all tokens at once. This allows faster training and better scalability on large datasets.
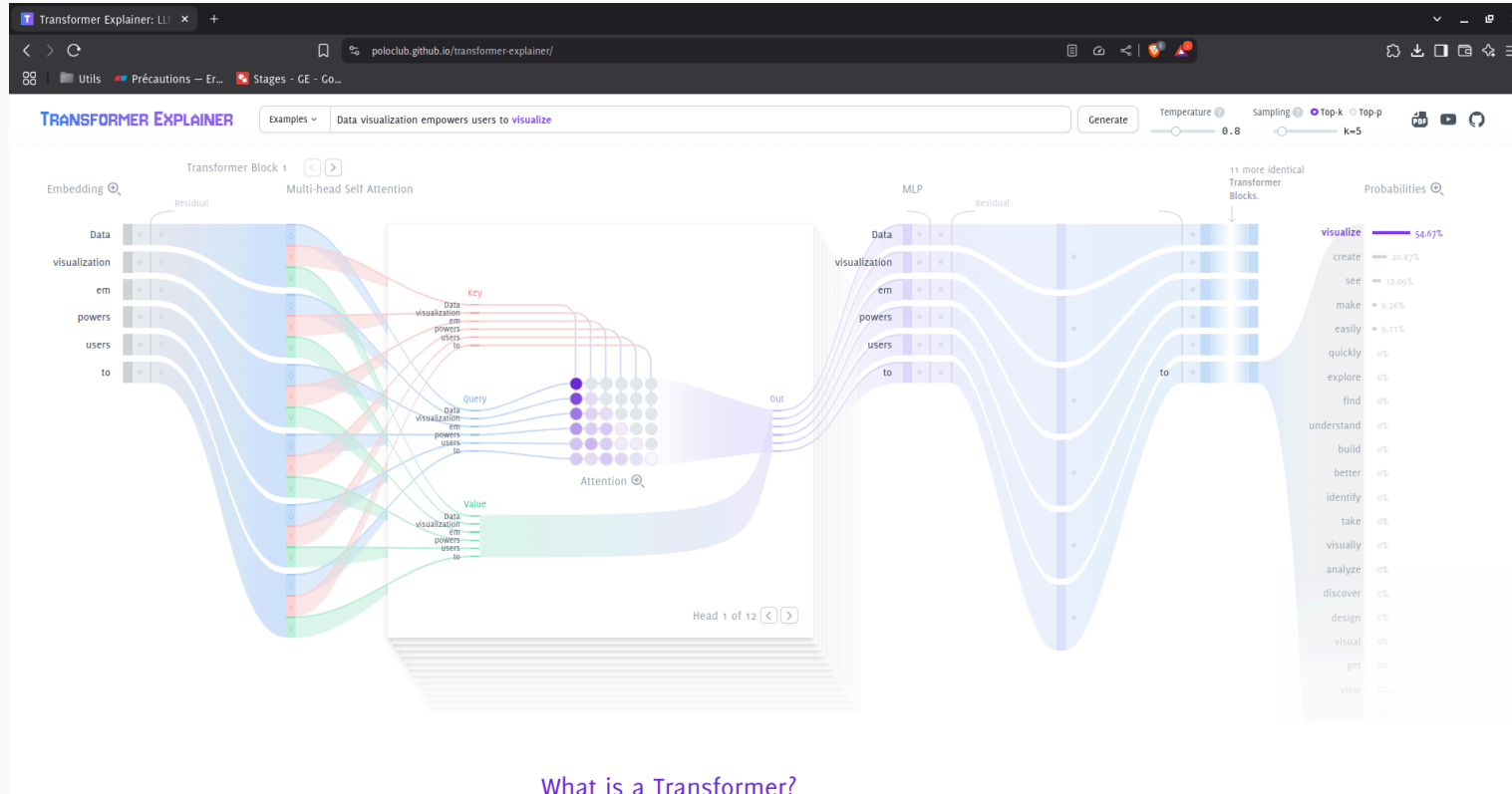
---

[1] https://arxiv.org/abs/1706.03762

Figure 5: Transformer explained visually[1].

---

[1] https://poloclub.github.io/transformer-explainer/
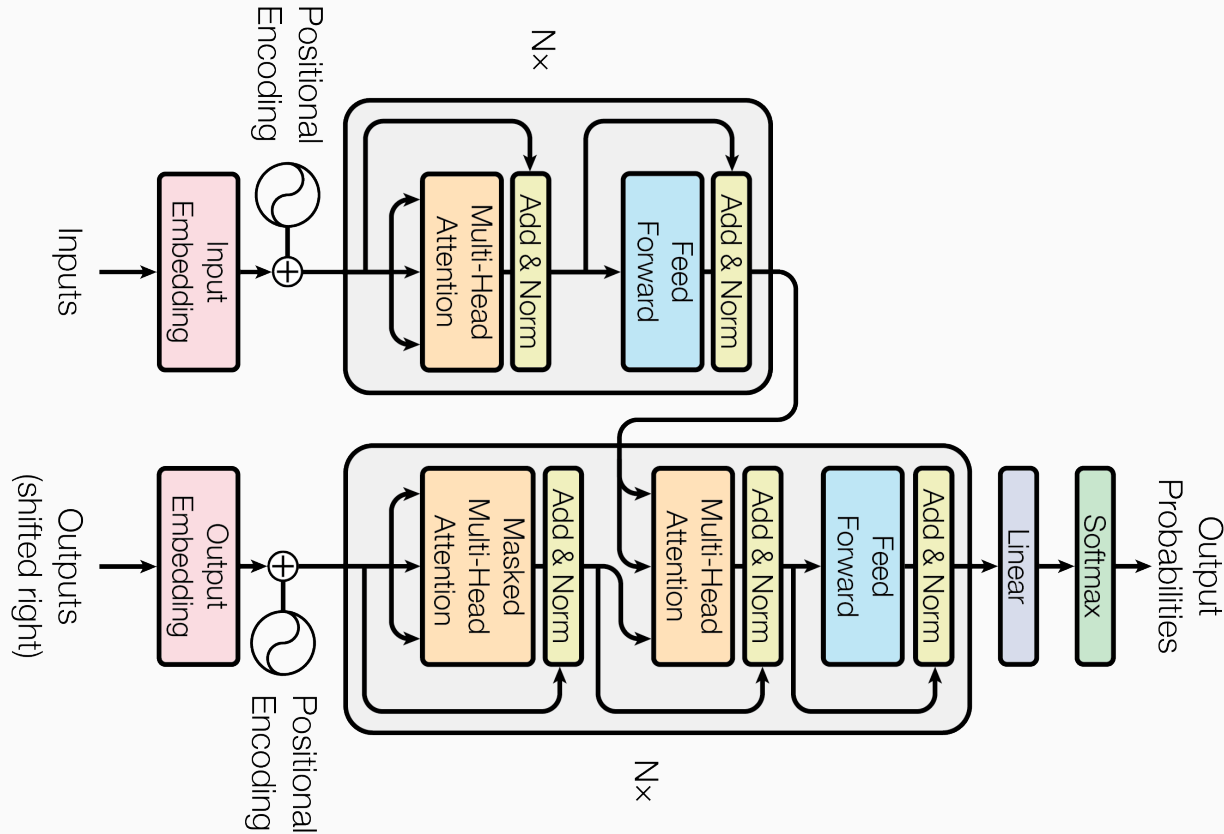
Figure 6: Core Components[1].

---

[1] https://arxiv.org/abs/1706.03762

5.4.1 Positional Encoding

- Transformers don't inherently understand word order.
- Positional encodings are added to word embeddings to inject information about the position of words in a sentence.

5.4.2 Encoder-Decoder Structure

- **Encoder**: Takes the input sequence and encodes it into a context-aware representation.
- **Decoder**: Uses the encoded representation to generate output *(e.g., in translation or summarization)*.

5.4.3 Self-Attention

- Each word in a sentence pays attention to every other word to understand context.
- This allows the model to dynamically weigh the importance of different words.
- Example: In the sentence "She poured water into the glass until it was full.", the model can learn that "it" refers to "glass".

The key/value/query concept is analogous to retrieval systems. For example, when you search for videos on Youtube, the search engine will map your query (text in the search bar) against a set of keys (video title, description, etc.) associated with candidate videos in their database, then present you the best matched videos (values)[1].

---

[1] https://stats.stackexchange.com/questions/421935/what-exactly-are-keys-queries-and-values-in-attention-mechanisms

Architecture

- Multi-head self-attention layers
- Feed-forward neural networks
- Layer normalization
- Residual connections

Use Cases

- Machine Translation
- Text Generation
- Question Answering
- Text Classification
- Summarization

Popular Models
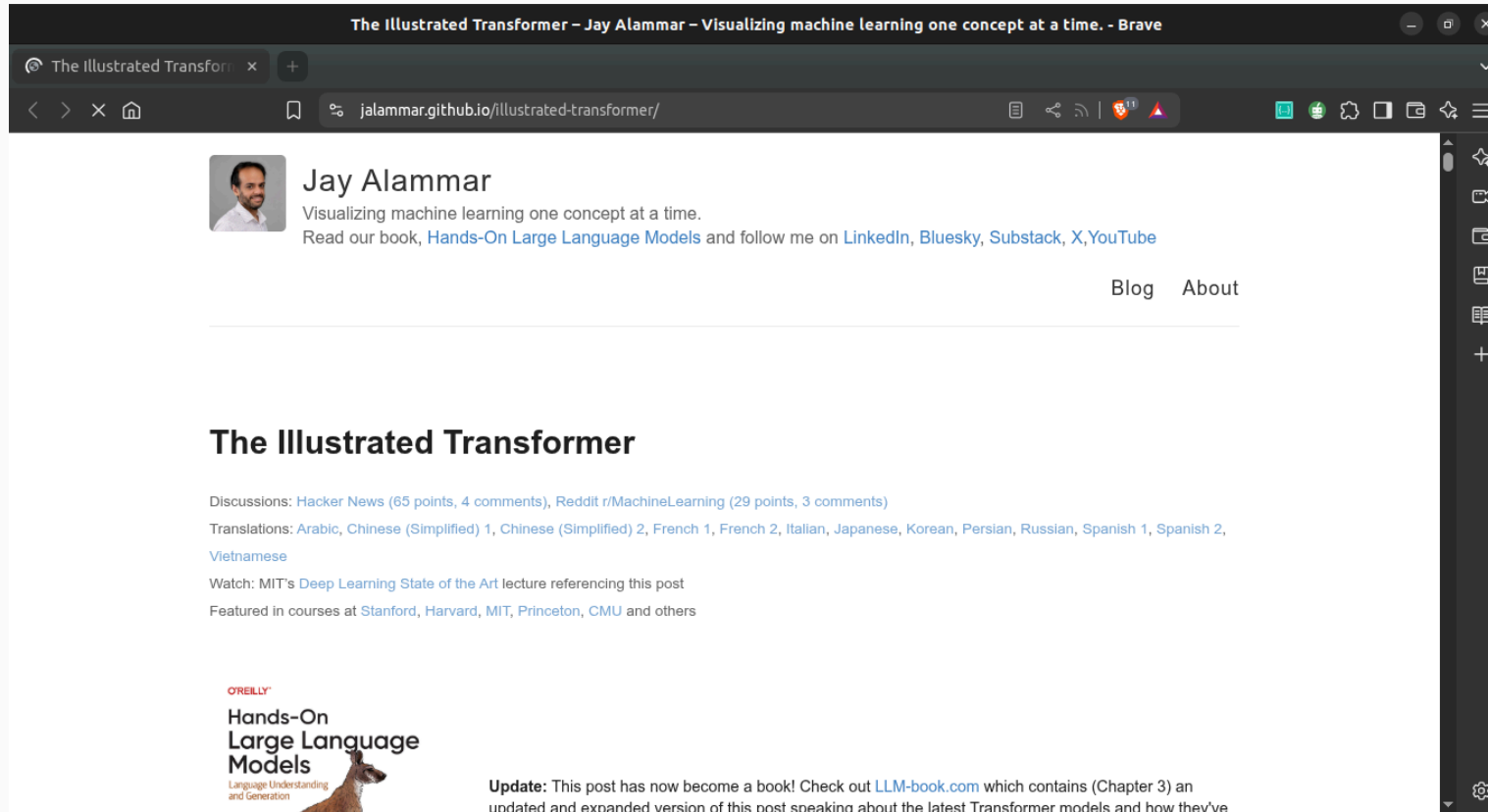
- BERT
- GPT
- T5
- RoBERTa, XLNet, DistilBERT

Figure 7: Transformer Illustrated[1].

---

[1] https://jalammar.github.io/illustrated-transformer/

Thank you for your attention!

References

[1] A. Vaswani *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[2] M. Honnibal, I. Montani, S. Van Landeghem, and A. Boyd, "spaCy: Industrial-strength Natural Language Processing in Python," 2020, [Online]. Available: https://spacy.io/

[3] R. Řehůřek and P. Sojka, "Software framework for topic modelling with large corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, 2010, pp. 45–50.

[4] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[5] Z. S. Harris, "Distributional structure," *Word*, vol. 10, no. 2–3, pp. 146–162, 1954.

[6] K. Sparck Jones, "A statistical interpretation of term specificity and its application in retrieval," MCB UP Ltd, 1972, pp. 11–21.

[7] P. F. Brown, P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-based n-gram models of natural language," *Computational linguistics*, vol. 18, no. 4, pp. 467–479, 1992.

[8] K. Cho *et al.*, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1724–1734.