SUDO CODE - WEEK 2: TEXT PREPROCESSSING

Doan Ngoc Mai

September 4, 2025

Abstract

Text preprocessing is a fundamental step in Natural Language Processing (NLP), aiming to transform raw text into a clean and structured format suitable for analysis and modeling. This survey provides an overview of both traditional preprocessing methods and modern techniques used in applications involving Large Language Models (LLMs) and Retrieval-Augmented Generation (RAG). Traditional methods such as tokenization, normalization, stopword removal, stemming, lemmatization, and vectorization (e.g., Bag-of-Words and TF-IDF) are essential for reducing noise, standardizing textual data, and creating numerical representations. In contrast, LLM-focused preprocessing involves more advanced steps such as subword tokenization, chunking, embedding generation, and pipeline construction for large-scale retrieval and generation tasks. Through examples and code demonstrations, this survey highlights the role and importance of preprocessing, as well as strategies for building effective pipelines that combine multiple techniques.

1 What is Text Preprocessing?

Text preprocessing refers to a set of techniques used to transform raw text into a clean, structured, and machine-readable format. Since most algorithms and models cannot directly handle raw text, preprocessing helps standardize input, reduce noise, and prepare text for analysis.

2 Text Preprocessing Techniques

Text preprocessing is a critical step in Natural Language Processing (NLP), where raw, unstructured text is transformed into a clean, consistent, and machine-readable format. These techniques serve as the foundation for both traditional machine learning pipelines and modern Large Language Model (LLM) applications by reducing noise, standardizing input, and enabling models to capture meaningful patterns. The following subsections highlight the most widely used techniques.

2.1 Tokenization

Tokenization refers to splitting raw text into smaller units, known as *tokens*. Tokens may represent words, subwords, or entire sentences, depending on the application.

- **Traditional approach:** Word-level tokenization, e.g., `"I love NLP"` → `["I", "love", "NLP"]`.
- **Modern approach (LLMs):** Subword tokenization using algorithms such as Byte Pair Encoding (BPE), WordPiece, or SentencePiece. For example: `"playing"` → `["play", "##ing"]`.

2.2 Normalization

Normalization standardizes text into a consistent form, addressing variations in case, punctuation, and character encoding. Common steps include:

- Lowercasing: `"NLP"` → `"nlp"`
- Removing punctuation/special characters: `"Hello!!!"` → `"Hello"`
- Handling diacritics: `"café"` → `"cafe"`
- Expanding contractions: `"don't"` → `"do not"`

2.3 Stopword Removal

Stopwords are high-frequency words that typically carry limited semantic information (e.g., "and", "the", "is"). Removing them reduces dimensionality and noise.

Example: `"I am learning NLP"` → `["learning", "NLP"]`

However, in tasks like sentiment analysis, stopwords may influence meaning and should sometimes be retained.

2.4 Lemmatization and Stemming

Both methods reduce words to their base form, but differ in methodology:

- **Stemming:** Uses heuristic truncation (e.g., `"running"`, `"runs"` → `"run"`).
- **Lemmatization:** Uses linguistic rules and dictionaries (e.g., `"better"` → `"good"`).

Stemming is computationally cheaper, while lemmatization yields more linguistically accurate results.

2.5 Bag-of-Words (BoW)

The Bag-of-Words model is one of the simplest ways to represent text for machine learning. It works by building a vocabulary of all unique words in the dataset, and then representing each document as a vector that counts how many times each word appears. Importantly, BoW ignores grammar and the order of words, focusing only on word frequency.

Example:
Documents: `"I love NLP"`, `"I love AI"`
Vocabulary: [I, love, NLP, AI]
Vector for "I love NLP": [1, 1, 1, 0]
Vector for "I love AI": [1, 1, 0, 1]

2.6 TF-IDF

Term Frequency–Inverse Document Frequency (TF-IDF) is a method that improves on Bag-of-Words by not only counting words, but also considering how important they are.

- **Term Frequency (TF):** Measures how often a word appears in a document.
- **Inverse Document Frequency (IDF):** Reduces the weight of words that appear in many documents, and increases the weight of rare words.

The key idea:

- Common words (e.g., `"the"`, `"and"`) → low weight.
- Rare, domain-specific words (e.g., `"transformer"`, `"photosynthesis"`) → high weight.

  Example:
  Document 1: `"I study machine learning"`
  Document 2: `"I study deep learning"`

  Common words (`"I"`, `"study"`, `"learning"`) have **lower importance**.
  Unique words (`"machine"`, `"deep"`) have **higher importance**.

3 The Role and Importance of Text Preprocessing

Text preprocessing is a fundamental step in Natural Language Processing (NLP) workflows as well as in modern applications involving Large Language Models (LLMs). Raw text data obtained from diverse sources such as websites, social media, or enterprise documents is often unstructured and noisy. Issues like inconsistent capitalization, spelling variations, stopwords, special characters, or irrelevant information can make direct analysis inefficient and even misleading. Preprocessing addresses these challenges by systematically transforming raw text into a cleaner, more standardized, and structured form that is suitable for computational models.

Why is text preprocessing necessary? Text data often contains noise such as punctuation, casing differences, stopwords, or special characters. Without preprocessing, models face:

- Large and redundant vocabulary size.
- Poor feature representation (harder for models to capture patterns).
- High computational costs.

Key benefits of text preprocessing. The role and importance of text preprocessing can be outlined across several dimensions:

- **Data quality improvement:** Text preprocessing enhances the quality of raw data by removing unwanted symbols, HTML tags, or malformed characters. For example, reviews collected from online platforms often include emojis, inconsistent spellings, or abbreviations, all of which need normalization before analysis.

- **Dimensionality reduction:** By removing stopwords, applying stemming or lemmatization, and normalizing variants of the same word, preprocessing reduces the vocabulary size. This simplification makes models less prone to overfitting and decreases storage as well as training time.
- **Better feature representation:** Preprocessing enables the transformation of raw text into machine-readable features such as Bag-of-Words, TF-IDF, or dense embeddings. Clean and well-structured input ensures that these features more accurately capture semantic patterns within the text.
- **Improved model performance:** Models trained on preprocessed data tend to achieve higher accuracy, precision, and recall in downstream tasks. For instance, sentiment analysis systems perform significantly better when text is tokenized, lowercased, and normalized, compared to training on unprocessed text.
- **Efficiency in computation:** Large corpora can contain millions of unique tokens due to minor variations (e.g., "Apple", "apple", "apples"). Preprocessing consolidates these into fewer tokens, reducing memory usage and speeding up training.
- **Support for domain adaptation:** In specialized domains such as medicine or law, preprocessing can include custom tokenization, abbreviation expansion, and domain-specific stopword lists. This ensures that models capture relevant terminology more effectively.

Relevance for modern LLM and RAG systems. While traditional preprocessing was primarily concerned with preparing text for classical machine learning algorithms, it remains equally important in the era of LLMs. In large-scale systems such as Retrieval-Augmented Generation (RAG), preprocessing ensures that documents are chunked consistently, stripped of noise, and tokenized in a manner compatible with the LLM tokenizer. Without this step, retrieval quality suffers, and the LLM may generate incomplete or irrelevant outputs. Furthermore, preprocessing helps align context length, improves indexing efficiency, and reduces hallucinations during text generation.

4 Traditional Text Preprocessing Techniques

Traditional text preprocessing techniques aim to clean and transform raw text into forms that can be directly used by classical machine learning algorithms. These methods are particularly useful in tasks such as text classification, sentiment analysis, and information retrieval. The most common techniques include:

- **Tokenization:** Splitting text into tokens such as words or sentences. For example, the sentence `"I love Natural Language Processing."` can be tokenized into `["I", "love", "Natural", "Language", "Processing"]`.
- **Lowercasing and Normalization:** Converting text to lowercase and removing punctuation to ensure consistency. E.g., `"NLP! is FUN"` → `"nlp is fun"`.
- **Stopword Removal:** Eliminating high-frequency words that do not carry significant meaning (e.g., "the", "is", "and"). This reduces vocabulary size and noise.
- **Stemming and Lemmatization:** Reducing words to their base or root forms. For example, stemming may reduce `"running"` and `"runs"` to `"run"`, while lemmatization would map `"better"` to `"good"`.

- **Vectorization:** Converting tokens into numerical representations. Common approaches include:
  - Bag-of-Words (BoW): Representing documents as word count vectors.
  - TF-IDF: Assigning weights based on frequency and inverse document frequency to highlight important terms.

Example using NLTK and Scikit-learn:

```
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer

docs = ["I love NLP", "NLP is fun and exciting"]

# Tokenization
tokens = [word_tokenize(doc.lower()) for doc in docs]

# Stopword removal
stop_words = set(stopwords.words('english'))
filtered = [[w for w in doc if w not in stop_words] for doc in tokens]

# TF-IDF representation
vectorizer = TfidfVectorizer()
tfidf = vectorizer.fit_transform([" ".join(doc) for doc in filtered])

print(vectorizer.get_feature_names_out())
print(tfidf.toarray())
```

This code snippet produces a vocabulary (e.g., `["exciting", "fun", "love", "nlp"]`) and corresponding TF-IDF vectors, demonstrating how preprocessing prepares text for machine learning.

5 Preprocessing for LLM and RAG Applications

Modern systems like Large Language Models (LLMs) and Retrieval-Augmented Generation (RAG) need more advanced preprocessing:

- **Subword Tokenization:** Using BPE, WordPiece, or SentencePiece for rare words.
- **Chunking:** Splitting long documents into smaller windows for LLM context limits.
- **Metadata enrichment:** Adding titles, timestamps, or categories to improve retrieval.
- **Embedding generation:** Transforming text into dense vectors for semantic search.

Example with Hugging Face Tokenizer:

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-multilingual-cased")
text = "Retrieval-Augmented Generation improves LLMs on long documents."

# Subword tokenization
tokens = tokenizer.tokenize(text)
print(tokens)

# Simple chunking
def chunk_text(text, max_len=10):
    words = text.split()
    return [" ".join(words[i:i+max_len]) for i in range(0, len(words), max_len)]

chunks = chunk_text(text, max_len=5)
print(chunks)
```

6 Building a Text Preprocessing Pipeline

In real-world NLP applications, preprocessing is rarely a single operation. Instead, it is organized into a *pipeline*, where multiple steps are executed in a fixed order. A pipeline ensures that every text sample undergoes the same sequence of transformations, thereby improving reproducibility, scalability, and consistency across experiments or deployments.

Core stages of a preprocessing pipeline. A typical text preprocessing pipeline may include the following stages:

1. **Text Cleaning:** Remove noise such as HTML tags, numbers, or special characters that do not contribute to meaning.
2. **Normalization:** Standardize text by lowercasing, expanding contractions (e.g., "don't" $\rightarrow$ "do not"), or removing diacritics.
3. **Tokenization:** Segment text into words, subwords, or sentences, depending on the downstream model.
4. **Stopword Removal and Lemmatization:** Reduce vocabulary size by filtering out common but uninformative words and mapping words to their root forms while preserving semantics.
5. **Feature Representation:** Convert tokens into numerical form using Bag-of-Words, TF-IDF, or dense embeddings for machine learning models.
6. **LLM/RAG-specific Processing:** For modern systems, apply model-compatible subword tokenization, split long documents into chunks to fit context windows, and generate embeddings for retrieval or indexing in vector databases.

Illustrative example with Scikit-learn. The following example shows how preprocessing can be encapsulated into a reusable pipeline that integrates feature extraction with model training:

```
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression

pipeline = Pipeline([
    ('tfidf', TfidfVectorizer(stop_words='english')),
    ('clf', LogisticRegression())
])

docs = ["I love NLP", "Deep learning is powerful"]
labels = [1, 0]

pipeline.fit(docs, labels)
print(pipeline.predict(["I love deep learning"]))
```

This pipeline combines TF-IDF vectorization with stopword removal and a classifier into a single workflow. Such designs simplify experimentation, reduce errors from manual preprocessing, and make the system more maintainable.