Natural Language Processing using Python Programming

Notebook 06.1: Introduction to Text Vectorization

Python 3.8+ NLTK Latest SpaCy Latest License MIT

Part of the comprehensive learning series: Natural Language Processing using Python Programming

Learning Objectives:

- Master the fundamental concepts of text vectorization for machine learning
- Understand the necessity of converting text to numerical representations
- Implement Bag of Words (BoW) model from scratch for conceptual clarity
- Learn Term Frequency-Inverse Document Frequency (TF-IDF) theory and mathematics
- Build foundation for advanced vectorization techniques with scikit-learn

---

- Machine Learning algorithms only understand numbers.
- **Text Vectorization** is the process of converting raw text data (words, sentences, documents) into numerical feature vectors that a model can process.
- This step is critical for any statistical or machine learning-based NLP task.

1. The Necessity of Vectorization

- Consider a classification task: determining if a review is positive or negative.
- We need to measure the importance of words like 'great' vs. 'terrible'.
- A computer needs a quantitative measure for each word.

In [1]:
```
# sample documents to demonstrate text vectorization
documents = [
    "The film was great, very great and entertaining.",
    "The plot was terrible, but the acting was fine.",
    "Great film, terrible plot, but the acting was okay."
]

# enumerate() : adds a counter to an iterable and returns it as an enumerate obje
print("Sample Documents:")
for i, doc in enumerate(documents):
    print(f"D{i+1}: {doc}")
```

```
Sample Documents:
D1: The film was great, very great and entertaining.
D2: The plot was terrible, but the acting was fine.
D3: Great film, terrible plot, but the acting was okay.
```

---

2. Bag of Words (BoW)

- The **Bag of Words (BoW)** model is the simplest vectorization technique.
- It represents a text document as an unordered collection (a "bag") of words, disregarding grammar and word order, but keeping track of **word frequency**.

How BoW Works:

1. **Vocabulary Creation:** Create a unique list of all words across all documents.
2. **Vector Generation:** For each document, create a vector where each dimension corresponds to a word in the vocabulary, and the value is the **count** of that word in the document.

In [2]:

```python
# Import necessary libraries
# pandas is used for data manipulation and analysis
# Counter is used to count hashable objects from collections module
import pandas as pd
from collections import Counter

# Step 1: Manual Preprocessing (for this conceptual example)
tokenized_docs = [
    ['film', 'great', 'great', 'entertaining'],
    ['plot', 'terrible', 'acting', 'fine'],
    ['great', 'film', 'terrible', 'plot', 'acting', 'okay']
]

# Step 2: Build Vocabulary
vocabulary_set = set(word for doc in tokenized_docs for word in doc)
vocabulary = sorted(list(vocabulary_set))

print(f"Vocabulary (Total {len(vocabulary)} words): {vocabulary}\n")

# Step 3: Create BoW Vectors
bow_vectors = []
for doc in tokenized_docs:
    word_counts = Counter(doc)
    vector = [word_counts[word] for word in vocabulary]
    bow_vectors.append(vector)

# Display results as a DataFrame for clarity
df_bow = pd.DataFrame(bow_vectors, columns=vocabulary, index=[f'D{i+1}' for i in r

print("Bag of Words (BoW) Matrix:")
print(df_bow)
```

```
Vocabulary (Total 8 words): ['acting', 'entertaining', 'film', 'fine', 'great', 'ok
ay', 'plot', 'terrible']

Bag of Words (BoW) Matrix:
    acting  entertaining  film  fine  great  okay  plot  terrible
D1       0             1     1     0      2     0     0         0
D2       1             0     0     1      0     0     1         1
D3       1             0     1     0      1     1     1         1
```

Limitations of BoW:

1. **High Dimensionality:** The vector size equals the vocabulary size. For large corpora, this vector can have hundreds of thousands of dimensions.
2. **Sparsity:** Most entries in the matrix are zero (most words don't appear in most documents), leading to storage and computational inefficiencies.
3. **No Semantic Weighting:** It treats the frequent, important word ('great') the same as a frequent, unimportant word ('the') if they appear the same number of times (though in this example, we pre-cleaned the stopwords).

---

3. TF-IDF (Term Frequency-Inverse Document Frequency)

- **TF-IDF** addresses the major limitation of BoW by weighting words based on their **importance**.
- It assigns a higher score to words that are **frequent in a specific document** but **rare across the entire corpus**.
- The final TF-IDF score for a term t in a document d in corpus D is calculated as:

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

3.1 Term Frequency (TF)

- Measures how often a term t appears in a document d.
- This is simply the count, or normalized count, of the word.

In [3]:
```
print("Term Frequency (TF) - Simple Count:")
print(df_bow) # Same as the BoW counts
```

```
Term Frequency (TF) - Simple Count:
    acting  entertaining  film  fine  great  okay  plot  terrible
D1       0             1     1     0      2     0     0         0
D2       1             0     0     1      0     0     1         1
D3       1             0     1     0      1     1     1         1
```

3.2 Inverse Document Frequency (IDF)

- Measures how unique or rare a term is across the entire corpus D.
- The formula (with smoothing to prevent division by zero) is:

$$\text{IDF}(t, D) = \log\left(\frac{N}{DF(t)}\right) + 1$$

- Where:
  - N is the total number of documents (3 in our case).
  - $DF(t)$ is the number of documents containing term t.

Term	DF(t)	$\text{IDF}(t, D) = \log(3/DF(t)) + 1$ **(Conceptual)**	Rarity Weight
great	2	$\log(3/2) + 1 \approx 1.40$	Low
terrible	2	$\log(3/2) + 1 \approx 1.40$	Low
entertaining	1	$\log(3/1) + 1 \approx 2.09$	High

> **Intuition:** A word like 'entertaining' which appears in only 1 out of 3 documents gets a higher IDF weight than a word like 'great' which appears in 2 out of 3 documents.

3.3 The Final TF-IDF Score

- By multiplying **TF** (count) and **IDF** (rarity), we get a final score that is high only for words that are frequent **locally** (in the document) AND rare **globally** (in the corpus).
- This gives us powerful, feature-rich vectors.

4. Summary and Next Steps

- We have established that vectorization is mandatory for ML, and that **TF-IDF** is generally superior to raw **BoW** as it incorporates a measure of word importance across the corpus.
- In the next notebook (**6.2**), we will practically implement these concepts using Scikit-learn's optimized vectorizers, which is the standard approach in the data science industry.

Key Takeaways

- **Vectorization Necessity:** We mastered the fundamental requirement of converting text to numerical representations for machine learning algorithms to process language data.

- **Bag of Words Implementation:** We built BoW vectors from scratch, understanding vocabulary creation, frequency counting, and the resulting high-dimensional sparse matrices.

- **TF-IDF Mathematical Foundation:** We learned the theory behind Term Frequency-Inverse Document Frequency, including the mathematical formulation and importance weighting concepts.

- **Limitations Understanding:** We identified key challenges with basic vectorization approaches including dimensionality, sparsity, and semantic weighting issues.

Next Notebook Preview

- With vectorization theory mastered, we're ready to implement **production-grade text vectorization**.

- The next notebook will dive into **practical vectorization with scikit-learn**, using optimized implementations of BoW, TF-IDF, and advanced techniques.

About This Project

This notebook is part of the **Natural Language Processing using Python Programming for Beginners** repository - a comprehensive, beginner-friendly guide for mastering NLP using Python, NLTK, and SpaCy.

Repository: `NLP`

Author

Prakash Ukhalkar
GitHub prakash-ukhalkar

Built with ❤️ for the Python community