Natural Language Processing using Python Programming

Notebook 06.2: Using Scikit-learn for Feature Extraction

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

Learning Objectives:

- Master production-grade text vectorization using scikit-learn's optimized tools
- Implement Bag of Words with CountVectorizer for industrial applications
- Apply TF-IDF vectorization with TfidfVectorizer for weighted feature extraction
- Learn N-grams for capturing word order and contextual relationships
- Understand critical train-test data splitting practices for ML pipelines

---

- In industrial data science, we don't manually calculate BoW or TF-IDF.
- Instead, we use highly optimized libraries.
- **Scikit-learn** is the standard tool for this, providing the `CountVectorizer` (for BoW) and `TfidfVectorizer` (for TF-IDF).
- This notebook demonstrates their practical use on our processed movie review data.

1. Setting up: Loading Data and Tools

- We load the cleaned data saved from Chapter 3.2 and import the Scikit-learn vectorizers.

In [1]:
```
# Import necessary libraries
# We load the cleaned data saved from Chapter 3.2 and import the Scikit-learn vectoriz
# Note: Ensure you have run the data processing notebook (03_2_using_real_world_datase
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

# Load the processed data (assuming it was saved in Chapter 3.2)
FILE_PATH = '../../data/processed/processed_reviews.csv'

try:
    df = pd.read_csv(FILE_PATH)
    # Extract the cleaned text column
    text_data = df['cleaned_review'].fillna('').tolist()
    print("Processed data loaded successfully.")
    print(f"Total documents: {len(text_data)}\n")
except FileNotFoundError:
    print(f"ERROR: Processed data not found at {FILE_PATH}. Please run 03_2_using_real
    text_data = []
```

```
Processed data loaded successfully.
Total documents: 5
```

---

2. Bag of Words (BoW) with `CountVectorizer`

- The `CountVectorizer` performs the entire BoW process: tokenization, vocabulary building, and feature matrix creation, using word counts as features.

In [2]:
```
# 1. Initialize the vectorizer
# min_df ignores terms that appear in less than 1 document (useful for large corpora)
count_vectorizer = CountVectorizer(min_df=1)

# 2. Learn the vocabulary AND transform the data
X_count = count_vectorizer.fit_transform(text_data)

print("CountVectorizer Feature Matrix Shape (Documents x Vocabulary):")
print(X_count.shape)
```

```
CountVectorizer Feature Matrix Shape (Documents x Vocabulary):
(5, 34)
```

In [3]:
```
# 3. Display the vocabulary
feature_names = count_vectorizer.get_feature_names_out()
print(f"\nTotal Unique Features (Words): {len(feature_names)}\n")
```

```
Total Unique Features (Words): 34
```

In [4]:
```
# 4. Convert the matrix to an array and display the first document's vector
print("BoW Matrix (First 2 Documents):")
df_count = pd.DataFrame(X_count.toarray(), columns=feature_names)
df_count.head(2)
```

```
BoW Matrix (First 2 Documents):
```

Out[4]:

	absolutely	acting	bad	believe	bmovie	boring	cinema	experience	fantastic	film	.
0	1	1	0	0	0	0	0	0	1	1	.
1	0	0	1	0	0	1	0	0	0	0	.

2 rows × 34 columns

Using N-grams with `CountVectorizer`

- BoW ignores word order. We can include limited word order by using **N-grams** (sequences of N consecutive words).
  - **Unigrams (1-gram):** Single words (`great` , `film`)
  - **Bigrams (2-gram):** Pairs of words (`great film` , `was terrible`)
- We can set `ngram_range=(1, 2)` to include both unigrams and bigrams.

In [5]:
```
# Initialize vectorizer for Unigrams and Bigrams
ngram_vectorizer = CountVectorizer(ngram_range=(1, 2))
```

```
X_ngram = ngram_vectorizer.fit_transform(text_data)

feature_names_ngram = ngram_vectorizer.get_feature_names_out()

print(f"Total Features (Unigrams + Bigrams): {len(feature_names_ngram)}")
print(f"Example N-grams: {feature_names_ngram[3:8]}")
```

```
Total Features (Unigrams + Bigrams): 66
Example N-grams: ['acting fantastic' 'bad' 'bad movie' 'believe' 'believe spend']
```

---

3. TF-IDF with `TfidfVectorizer`

- The `TfidfVectorizer` is a single class that performs both the counting (TF) and the inverse document frequency weighting (IDF).

In [6]:
```
# 1. Initialize the vectorizer
tfidf_vectorizer = TfidfVectorizer()

# 2. Learn the vocabulary AND transform the data
X_tfidf = tfidf_vectorizer.fit_transform(text_data)

feature_names_tfidf = tfidf_vectorizer.get_feature_names_out()

print(f"TF-IDF Matrix Shape: {X_tfidf.shape}")
```

```
TF-IDF Matrix Shape: (5, 34)
```

In [7]:
```
# 3. Display the scores for the first document
df_tfidf = pd.DataFrame(X_tfidf.toarray(), columns=feature_names_tfidf)
print("\nTF-IDF Matrix (First 2 Documents):")
df_tfidf.head(2).round(3)
```

```
TF-IDF Matrix (First 2 Documents):
```

Out[7]:

	absolutely	acting	bad	believe	bmovie	boring	cinema	experience	fantastic	film
0	0.34	0.34	0.000	0.0	0.0	0.000	0.0	0.0	0.34	0.34
1	0.00	0.00	0.322	0.0	0.0	0.322	0.0	0.0	0.00	0.00

2 rows × 34 columns

Interpreting TF-IDF Scores

- Words with high TF-IDF scores are the best features for distinguishing one document from the rest:

    1. **High TF-IDF scores** indicate words that are frequent in the specific document but rare across the entire corpus

    2. **Low TF-IDF scores** (including 0.0) indicate either common words that appear in many documents or words that appear infrequently in the current document

    3. **Zero scores** mean the word doesn't appear in that particular document

- **Key Insight**: TF-IDF successfully identifies unique, characteristic words that best represent each document's content, making them valuable features for machine learning models.
- **Note**: Run the cell above to see the actual TF-IDF scores for your specific dataset - the exact values will depend on your processed movie review data.

---

4. Crucial Concept: `fit_transform` vs. `transform`

- This is the most common mistake for beginners.
- When preparing data for a machine learning model, you must split your data into a **Training Set** and a **Test Set**.
  - The vocabulary (feature list) must **ONLY** be learned from the **Training Data**.
  - The **Test Data** must use the exact same vocabulary/weights learned from the training data.

Data Set	Vectorizer Method	Action
Training Data	`.fit_transform()`	Learns the vocabulary (FIT) AND creates the vectors (TRANSFORM)
Test Data	`.transform()`	Uses the vocabulary/weights learned in the FIT step (TRANSFORM ONLY)

In [8]:
```
# Import train_test_split from sklearn for data splitting
from sklearn.model_selection import train_test_split

# Split data into training and testing sets
X_train_text, X_test_text, y_train, y_test = train_test_split(text_data, df['sentiment

print(f"Training Documents: {len(X_train_text)}")
print(f"Testing Documents: {len(X_test_text)}\n")

# --- The Correct Workflow ---
tfidf = TfidfVectorizer()

# 1. Train Data: FIT and TRANSFORM
X_train_vectors = tfidf.fit_transform(X_train_text)
print(f"Train Vector Shape: {X_train_vectors.shape}")

# 2. Test Data: TRANSFORM ONLY (must use the vocabulary learned above)
X_test_vectors = tfidf.transform(X_test_text)
print(f"Test Vector Shape:  {X_test_vectors.shape}")

# Observation: Both the train and test vectors have the SAME number of columns (featur
# corresponding to the vocabulary learned ONLY from the training data.
```

```
Training Documents: 3
Testing Documents: 2

Train Vector Shape: (3, 20)
Test Vector Shape:  (2, 20)
```

5. Summary and Next Steps

- We successfully implemented **BoW** (CountVectorizer) and **TF-IDF** (TfidfVectorizer) using Scikit-learn, learned how to incorporate **N-grams**, and established the crucial best practice of separating `fit_transform` and `transform` for training and testing data.
- We now have feature vectors ready for machine learning!
- In **Chapter 7**, we will apply these vectors to solve a focused, high-value problem: **Sentiment Analysis**.

Key Takeaways

- **Production Vectorization:** We mastered industry-standard text vectorization using scikit-learn's optimized CountVectorizer and TfidfVectorizer implementations.
- **Advanced Feature Engineering:** We implemented N-grams to capture word order and contextual relationships, moving beyond simple bag-of-words approaches.
- **ML Pipeline Best Practices:** We learned the critical distinction between fit_transform() for training data and transform() for test data, preventing data leakage.
- **Real-world Application:** We applied vectorization techniques to actual movie review data, preparing feature matrices ready for machine learning algorithms.

Next Notebook Preview

- With feature vectors prepared, we're ready to build **machine learning models** for NLP tasks.
- The next notebook will dive into **Sentiment Analysis**, applying our vectorized features to classify movie reviews as positive or negative.

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
