2.NLP Text Preprocessing and Vectorization: A Complete Guide

Introduction to NLP Pipeline

When solving Natural Language Processing problems like **sentiment analysis**, we follow a systematic approach to transform raw text into numerical representations that machine learning algorithms can process.

Key Terminology

- **Document (d_1, d_2, d_3, ...)**: Individual text samples, also called sentences
- **Corpus**: Collection of all documents combined (like a paragraph)
- **Vocabulary**: Set of unique words across all documents

The Complete NLP Pipeline

```
Problem Statement → Dataset → Text Preprocessing → Vectorization → Model Training → Pre
```

Step 1: Text Pre-processing (Part 1)

1.1 Tokenization

What it does: Breaks down text into smaller pieces (tokens).

Types:

- Paragraph → Sentences
- Sentences → Words

Example:

```
Input: "Natural language processing is fascinating. It helps computers."
Output: ["Natural", "language", "processing", "is", "fascinating", ".", "It", "helps"]
```

1.2 Lowercase Conversion

What it does: Converts all words to lowercase so "Though" and "though" are treated as the same word.

Simple Formula:

$$\text{New Word} = \text{lowercase}(\text{Original Word})$$

Example:

```
Before: ["Though", "though", "THOUGH"]
After:  ["though", "though", "though"] → Counted as 1 unique word
```

Why? Without this, "Cat", "cat", and "CAT" would be counted as 3 different words!

1.3 Regular Expressions (Regex)

What it does: Removes unwanted characters like punctuation, numbers, or special symbols.

Example:

```
Before: "Hello!!! This costs $50.99 #amazing"
After:  "Hello This costs amazing"
```

Step 2: Text Pre-processing (Part 2)

2.1 Stemming

What it does: Chops off word endings to get the root form.

Simple Idea:

$$\text{Stemmed Word} = \text{Root of Word}$$

Example:

```
running → run
runs → run
played → play
studying → studi (not a real word - this is a limitation!)
```

Note: Sometimes creates non-real words (e.g., "studies" → "studi")

2.2 Lemmatization

What it does: Converts words to their dictionary base form (more intelligent than stemming).

Example:

```
running → run
better → good
were → be
am → be
```

Difference from Stemming: Always produces valid dictionary words!

2.3 Stopwords Removal

What it does: Removes common words that don't add much meaning.

Common Stopwords: "the", "is", "at", "which", "on", "a", "an", "and"

Example:

```
Before: "The cat is sitting on the mat"
After:  "cat sitting mat"
```

Result: Vocabulary becomes smaller and more meaningful!

Step 3: Text to Vector Conversion

After preprocessing, we convert cleaned text into numbers that computers can understand.

3.1 One-Hot Encoding

What it does: Represents each word as a list of 0s with a single 1.

Simple Formula:

$$\text{Word Vector} = [0, 0, ..., 1, ..., 0]$$

(Only one position is 1, rest are all 0s)

Example:

Vocabulary: ["cat", "dog", "bird"]

```
cat  → [1, 0, 0]  ← 1st position is 1
dog  → [0, 1, 0]  ← 2nd position is 1
bird → [0, 0, 1]  ← 3rd position is 1
```

Problem:

- If vocabulary has 10,000 words, each vector has 10,000 numbers (mostly zeros)!
- Doesn't capture meaning or relationships

3.2 Bag of Words (BoW)

What it does: Counts how many times each word appears in a document.

Simple Formula:

$$\text{Document Vector} = [\text{count}_1, \text{count}_2, \text{count}_3, ...]$$

Each number = how many times that word appears

Example:

```
Vocabulary: ["good", "food", "bad", "service"]

Document 1: "good food good service"
   → [2, 1, 0, 1]  (good appears 2 times, food 1 time, bad 0 times, service 1 time)

Document 2: "bad food bad service"
   → [0, 1, 2, 1]  (good 0 times, food 1 time, bad 2 times, service 1 time)
```

Advantage: Simple and captures word importance by frequency!

Problem: Ignores word order ("food is good" vs "good is food" look the same)

3.3 TF-IDF (Term Frequency-Inverse Document Frequency)

What it does: Gives higher scores to important/distinctive words.

Part A: Term Frequency (TF)

How often does a word appear in THIS document?

$$\text{TF} = \frac{\text{Number of times word appears in document}}{\text{Total words in document}}$$

Example:

```
Document: "cat cat dog bird"
TF(cat) = 2 ÷ 4 = 0.5  (cat appears 2 times out of 4 total words)
```

Part B: Inverse Document Frequency (IDF)

How rare is this word across ALL documents?

Basic Formula (most common in theory):

$$\text{IDF} = \log\left(\frac{\text{Total number of documents}}{\text{Number of documents containing the word}}\right)$$

Smoothed Formula (used in practice, like scikit-learn):

$$\text{IDF} = \log\left(\frac{\text{Total documents} + 1}{\text{Documents containing word} + 1}\right) + 1$$

Why smoothing?

- Adds 1 to numerator and denominator to prevent division by zero
- Adds 1 to final result so IDF is never negative
- Handles unseen words during prediction

Example (Basic):

```
We have 100 documents total
Word "cat" appears in 10 documents

IDF(cat) = log(100 ÷ 10) = log(10) ≈ 2.3
```

Example (Smoothed):

```
IDF(cat) = log((100 + 1) ÷ (10 + 1)) + 1
         = log(101 ÷ 11) + 1
         = log(9.18) + 1
         ≈ 2.2 + 1 = 3.2
```

Interpretation:

- Rare words get **higher** IDF scores
- Common words (appearing in many documents) get **lower** IDF scores
- Words appearing in ALL documents get IDF ≈ 1 (with smoothing)

Final Score: TF-IDF

$$\text{TF-IDF Score} = \text{TF} \times \text{IDF}$$

What this means:

- **High TF**: Word appears often in this document
- **High IDF**: Word is rare across all documents
- **High TF-IDF**: Word is important and distinctive for this document!

Complete Example:

```
We have 2 documents:
Document 1: "cat cat dog"
Document 2: "dog bird bird"

For word "cat" in Document 1:

Step 1 - Calculate TF:
TF = 2 ÷ 3 ≈ 0.67  (cat appears 2 times out of 3 words)

Step 2 - Calculate IDF (basic formula):
IDF = log(2 ÷ 1) = log(2) ≈ 0.69  (cat appears in 1 out of 2 documents)

Step 3 - Calculate TF-IDF:
TF-IDF = 0.67 × 0.69 ≈ 0.46
```

Why better than BoW?: Words like "the" and "is" appear everywhere, so they get low IDF scores (less importance)!

3.4 Word2Vec

What it does: Converts words into dense number vectors that capture meaning and relationships.

Simple Idea:

$$\text{Word} \rightarrow \text{Vector of Numbers}$$

Example: "king" → [0.2, 0.5, -0.3, 0.8, ...] (typically 100-300 numbers)

The Magic of Word2Vec

Similarity Between Words (Cosine Similarity):

$$\text{Similarity} = \frac{\text{Vector 1} \cdot \text{Vector 2}}{||\text{Vector 1}|| \times ||\text{Vector 2}||}$$

Where:

- **Vector 1 · Vector 2** = Dot product (multiply matching positions and add them up)
- **||Vector||** = Length/Magnitude of vector = $\sqrt{v_1^2 + v_2^2 + v_3^2 + \ldots}$

This gives a number between -1 and 1:

- **1** = Very similar words (vectors point in same direction)

- **0** = Unrelated words (vectors are perpendicular)
- **-1** = Opposite words (vectors point in opposite directions)

Famous Mathematical Relationship:

$$\text{king} - \text{man} + \text{woman} = \text{queen}$$

This actually works with word vectors! 🤯

Example Similarity:

```
Similarity(cat, dog) = 0.8  (both are animals - similar!)
Similarity(cat, car) = 0.1  (not related - low score)
```

Advantages:

- Captures meaning and relationships
- Much smaller vectors (300 numbers instead of 10,000!)
- Can use pre-trained models (don't need to train from scratch)

Tools: Gensim library in Python

3.5 Average Word2Vec

What it does: Represents an entire document by averaging all word vectors.

Simple Formula:

$$\text{Document Vector} = \frac{1}{n} \sum_{i=1}^{n} \text{Word Vector}_i$$

Where:

- **n** = Number of words in the document
- ∑ (sigma) = Sum of all
- **Word Vector_i** = The vector for word i

In Plain English: Add all word vectors together, then divide by the number of words.

Step-by-step Example:

```
Document: "good food"

Step 1: Get Word2Vec for each word
  good → [0.2, 0.5, 0.3]
  food → [0.4, 0.3, 0.5]

Step 2: Add them together
  [0.2, 0.5, 0.3] + [0.4, 0.3, 0.5] = [0.6, 0.8, 0.8]

Step 3: Divide by number of words (n = 2)
  [0.6, 0.8, 0.8] ÷ 2 = [0.3, 0.4, 0.4]

Final Document Vector = [0.3, 0.4, 0.4]
```

Why useful?: Every document gets the same size vector, regardless of length!

Mathematical Note: The 1/n in front is equivalent to dividing the sum by n at the end - both are correct!

Step 4: Model Training

Once we have vectors (numbers), we can train machine learning models!

The Complete Flow:

$$\text{Raw Text} \xrightarrow{\text{Clean}} \text{Clean Text} \xrightarrow{\text{Vectorize}} \text{Numbers} \xrightarrow{\text{Train}} \text{Predictions}$$

Common ML Algorithms:

- Logistic Regression
- Naive Bayes
- Support Vector Machines
- Random Forest
- Neural Networks (Deep Learning)

Complete Example: Sentiment Analysis

Let's put it all together!

Input Dataset:

```
d₁: "The food is good!" → Label: Positive
d₂: "Bad service, terrible experience." → Label: Negative
d₃: "Good food, good service." → Label: Positive
```

Step 1 - Tokenization & Lowercase:

```
d₁: ["the", "food", "is", "good"]
d₂: ["bad", "service", "terrible", "experience"]
d₃: ["good", "food", "good", "service"]
```

Step 2 - Remove Stopwords ("the", "is"):

```
d₁: ["food", "good"]
d₂: ["bad", "service", "terrible", "experience"]
d₃: ["good", "food", "good", "service"]
```

Step 3 - Create Vocabulary:

```
Vocabulary: ["food", "good", "bad", "service", "terrible", "experience"]
Position:      0       1       2        3           4            5
```

Step 4 - Bag of Words Vectors:

```
d₁ → [1, 1, 0, 0, 0, 0]
     (1 food, 1 good, 0 bad, 0 service, 0 terrible, 0 experience)

d₂ → [0, 0, 1, 1, 1, 1]
     (0 food, 0 good, 1 bad, 1 service, 1 terrible, 1 experience)

d₃ → [1, 2, 0, 1, 0, 0]
     (1 food, 2 good, 0 bad, 1 service, 0 terrible, 0 experience)
```

Step 5 - Train Model:

```
Input (X):
  [1, 1, 0, 0, 0, 0]
  [0, 0, 1, 1, 1, 1]
  [1, 2, 0, 1, 0, 0]

Output (y):
  Positive
  Negative
  Positive

Model learns: High "good" count → Positive, High "bad" count → Negative
```

Step 6 - Predict New Reviews:

```
New Review: "terrible food"
Vectorize: [1, 0, 0, 0, 1, 0]
Prediction: Negative ✓
```

Quick Comparison Table

Technique	How It Works	Vector Size	Captures Meaning?
One-Hot	1 in one position, rest 0s	= Vocabulary size (large!)	❌ No
Bag of Words	Count each word	= Vocabulary size	❌ No
TF-IDF	Weighted word counts	= Vocabulary size	⚠️ Partly
Word2Vec	Dense number vectors	Small (100-300)	✅ Yes!
Avg Word2Vec	Average of word vectors	Small (100-300)	✅ Yes!

Key Takeaways

When to Use Each Technique:

1. **Starting out?** → Use **Bag of Words** (simple and effective)
2. **Want better results?** → Use **TF-IDF** (handles common words better)
3. **Need semantic meaning?** → Use **Word2Vec** (understands relationships)
4. **Large vocabulary?** → Avoid One-Hot (too many zeros!)
5. **Production systems?** → Word2Vec or Advanced (BERT, Transformers)

Remember:

✅ **Always preprocess first**: Clean text = Better results
✅ **Start simple**: Begin with BoW, then move to advanced techniques
✅ **Test different methods**: What works depends on your specific problem
✅ **Vectorization is key**: Computers need numbers, not words!

What's Next?

1. **Practice**: Implement each technique in Python
2. **Compare**: See which works best for your problem
3. **Learn Advanced**: Explore BERT, GPT, and Transformer models
4. **Build Projects**: Apply to real sentiment analysis, classification, chatbots!

Remember: These techniques are the foundation of ALL NLP! Even the most advanced AI models (like ChatGPT) build on these core concepts. Master these, and you're ready for anything! 🚀