NLP501 - SESSION 03

Vector Space Models

Words as Vectors in Semantic Space

Learning Objectives

1. Understand word vectors in semantic space

2. Build co-occurrence matrices from text

3. Apply TF-IDF weighting schemes

4. Calculate cosine similarity between vectors

5. Apply PCA for dimensionality reduction

6. Implement document search systems

Exercise 1 Released Today! Sentiment Analysis and Vector Spaces (Due: Session 5)

Session Outline

Part 1: Quick Review - Session 02 Naïve Bayes

Part 2: Word Representations

Part 3: Co-occurrence Matrices

Part 4: TF-IDF Weighting

Part 5: Cosine Similarity

Part 6: Dimensionality Reduction (PCA)

Part 7: Applications

Part 8: Summary and Lab Practice

PART 1

Quick Review

Session 02 - Naïve Bayes Classifier

Session 02 Review: Naïve Bayes

Key Formulas

$$P(c|d) = P(d|c) \times P(c) \,/\, P(d)$$

$$P(w_1, w_2, \ldots | c) = \prod P(w_i|c)$$

$$P(w|c) = (\text{count}(w,c) + \alpha) \,/\, (N^c + \alpha|V|)$$

Key Concepts

Bayes Theorem: Update prior beliefs with evidence

Naïve Assumption: Features are conditionally independent

Laplacian Smoothing: Handle unseen words (add α)

Log-space: Prevent underflow with log probabilities

Key Insight: Naïve Bayes is simple, fast, and works well for text classification despite the "naïve" assumption!

From Counts to Vectors: The Journey

Session 01: Logistic Regression

Feature extraction, Sigmoid function, Gradient descent optimization

Session 02: Naïve Bayes

Word frequencies, Conditional probabilities, Bayesian classification

Session 03: Vector Space Models

Words as vectors, Semantic similarity, Document search

Key Question for Today:

"How do we represent the MEANING of words?"

PART 2

Word Representations

From Symbols to Vectors

"You shall know a word by the company it keeps" — J.R. Firth (1957)

One-Hot Encoding

Definition

Vector with dimension = |V|, all zeros except one position = 1

Example: V = [cat, dog, happy, sad]

cat = [1, 0, 0, 0]
dog = [0, 1, 0, 0]
happy = [0, 0, 1, 0]
sad = [0, 0, 0, 1]

Problems

No Similarity: All word pairs are orthogonal

High Dimensional: Vector size = vocabulary size

Sparse: Only one non-zero element

No Semantics: cos(cat, dog) = 0

Key Issue: "cat" and "dog" should be similar (both animals), but one-hot gives cos(v_cat, v_dog) = 0

The Distributional Hypothesis

> *"Words that occur in similar contexts tend to have similar meanings"*
>
> — Zellig Harris (1954), J.R. Firth (1957)

Example: Similar Contexts

"The CAT sat on the mat"
"The DOG sat on the mat"
"A cute CAT is sleeping"
"A cute DOG is sleeping"

✓ "cat" and "dog" share contexts → similar vectors

Example: Different Contexts

"I feel HAPPY today"
"The ALGORITHM converged"
"She is HAPPY with results"
"The ALGORITHM is efficient"

✗ "happy" and "algorithm" → different vectors

💡 Key Insight: We can learn word meaning from context!

Instead of defining meanings manually, we let data tell us which words are similar based on usage patterns.

Two Types of Vector Space Models

📊 Sparse Vectors

Count-based Methods

- Long vectors (10,000+ dims)
- Most values = 0
- Based on co-occurrence counts

Methods: Term-Doc Matrix, TF-IDF

This Session's Focus

🧠 Dense Vectors

Prediction-based Methods

- Short vectors (50-300 dims)
- All values non-zero
- Learned from prediction tasks

Methods: Word2Vec, GloVe, FastText

Session 06: Word Embeddings

Both approaches capture semantic similarity, but dense vectors are more efficient for downstream tasks

PART 3

Co-occurrence Matrices

Building Word Vectors from Context

Term-Document Matrix

Definition

A matrix where:

- **Rows** = Words (terms)
- **Columns** = Documents
- **Cell [i,j]** = Count of word i in doc j

📖 **Example Corpus**

D1: "I love machine learning"
D2: "Machine learning is awesome"
D3: "Deep learning and machine learning"
D4: "I love deep learning"

Key: Each word becomes a vector over documents. Similar words appear in similar documents!

Term-Document Matrix

Term	D1	D2	D3	D4
i	1	0	0	1
love	1	0	0	1
machine	1	1	2	0
learning	1	1	2	1
deep	0	0	1	1
is	0	1	0	0
awesome	0	1	0	0
and	0	0	1	0

v_machine = [1, 1, 2, 0] · **v_learning** = [1, 1, 2, 1]

Word-Word Co-occurrence Matrix

Definition

A square matrix where:

- **Rows & Columns** = Words
- **Cell [i,j]** = Count of word j near word i
- **Symmetric:** M[i,j] = M[j,i]

🔍 Context Window

Defines "nearness" - typically k words on each side

Window k=2: 2 words left + 2 words right

Word-Word Matrix (k=1)

Sentence: I love deep learning

Pair: (I, love), (love, I), (love, deep), (deep, love), (deep, learning), (learning, deep)

	I	love	deep	learning
I	0	1	0	0
love	1	0	1	0
deep	0	1	0	1
learning	0	0	1	0

Benefit: Captures semantic relationships directly between words

Context Window: Example

Sentence: "The quick brown fox jumps over the lazy dog"

The quick brown **fox** jumps over the lazy dog

Target: "fox" · Context window k=2

Context Words for "fox" (k=2)

Left: quick, brown
Right: jumps, over

Context = {quick, brown, jumps, over}

Window Size Effects

- **Small (k=1-2):** Syntactic relationships
- **Medium (k=4-5):** Balanced
- **Large (k=10+):** Topical/semantic

💡 Key Insight

Window size is a hyperparameter. Small windows capture grammar, large windows capture topics.

PART 4

TF-IDF Weighting

Measuring Term Importance

The Problem with Raw Counts

Document About Machine Learning

"THE basics of MACHINE LEARNING are THE foundation for THE field... THE algorithms..."

15
"the" count

3
"ML" count

The Problem

- Common words like "the" have high counts
- Important topic words have low counts
- Raw counts don't reflect importance!

Issue: "the" dominates but tells us nothing about the document's topic

💡 Solution: TF-IDF

Weight terms by how important they are to a document relative to the entire corpus. Downweight common words, upweight distinctive words.

Term Frequency (TF)

Definition

Measures how frequently a term appears in a document.

$$tf(t, d) = f(t,d)$$

Count of term t in document d

Common Variants

Raw Count

tf(t, d) = f(t,d)

Log Normalization

tf(t, d) = 1 + log(ft,d)

Double Normalization

tf(t, d) = 0.5 + 0.5 × f(t,d) / max(fd)

Example Calculation

Document: "machine learning is great. machine learning is powerful. machine learning! "

	machine	learning	is
Raw TF	3	3	2
Boolean TF	1	1	1
Log TF	1 + log(3) ≈ 2.10	1 + log(3) ≈ 2.10	1 + log(2) ≈ 1.69
Augmented TF	0.5 + 0.5×(3/3) = 1.0	0.5 + 0.5×(3/3) = 1.0	0.5 + 0.5×(2/3) ≈ 0.83

Note: Log dampens the effect of high frequency. TF=3 → 2.10, not 3×.

Inverse Document Frequency (IDF)

Definition

Measures how **rare** or **common** a term is across all documents.

$$idf(t) = \log(N / df(t))$$

N = Total docs **df** = Docs with t

💡 **Intuition**

- Rare terms → High IDF (informative)
- Common terms → Low IDF (less useful)

Example: N = 1,000,000 docs

word	df(t)	Calculation	idf(t)
the	10,000	log(10000/10000)	0.00
computer	1,000	log(10000/1000)	1.00
algorithm	100	log(10000/100)	2.00
transformer	10	log(10000/10)	3.00

Note: IDF = 0 when term appears in ALL documents

TF-IDF: The Complete Formula

$$\text{tf-idf}(t, d) = \text{tf}(t, d) \times \text{idf}(t)$$

tf(t, d) = 1 + log(f_t,d)

idf(t) = log(N / df_t)

What TF-IDF Captures

- High TF + High IDF: Important, distinctive
- High TF + Low IDF: Common (downweighted)
- Low TF + High IDF: Rare but not frequent
- Low TF + Low IDF: Not important

Example: "neural" in ML doc

tf = 1 + log(5) = 2.61

idf = log(1000/50) = 1.30

tf-idf = 2.61 × 1.30 = 3.39

💡 **Key Insight**

TF-IDF balances local importance (TF) with global rarity (IDF). High scores = terms that are frequent in document AND rare across corpus.

TF-IDF: Python Implementation

```
import numpy as np
from collections import Counter
import math

def compute_tf(doc):
    """Tính Term Frequency cho một document"""
    word_counts = Counter(doc)
    max_count = max(word_counts.values())
    tf = {word: count / max_count for word, count in word_counts.items()}
    return tf

def compute_idf(corpus):
    """Tính IDF cho toàn bộ corpus"""
    N = len(corpus)
    # Đếm số documents chứa mỗi từ
    df = Counter()
    for doc in corpus:
        unique_words = set(doc)
        for word in unique_words:
            df[word] += 1
    # Tính IDF
    idf = {word: math.log(N / count) for word, count in df.items()}
    return idf

def compute_tfidf(doc, idf):
    """Tính TF-IDF vector cho một document"""
    tf = compute_tf(doc)
    tfidf = {word: tf[word] * idf.get(word, 0) for word in tf}
    return tfidf
```

Using sklearn (Recommended)

```
from sklearn.feature_extraction.text import TfidfVectorizer
# Khởi tạo TfidfVectorizer
vectorizer = TfidfVectorizer(
    lowercase=True,          # Chuyển về lowercase
    stop_words='english',    # Loại bỏ stop words
    max_df=0.9,              # Bỏ từ xuất hiện >90% docs
    min_df=1,                # Giữ từ xuất hiện ít nhất 1 doc
    ngram_range=(1, 2)       # Unigrams và bigrams
)

# Fit và transform
tfidf_matrix = vectorizer.fit_transform(corpus)
```

sklearn Parameters

- **max_df:** Ignore terms in > X% docs
- **min_df:** Ignore terms in < X docs
- **ngram_range:** (1,2) for unigrams+bigrams

Note: sklearn uses slightly different formulas with L2 normalization by default.

PART 5

Cosine Similarity

Measuring Semantic Distance

Why Not Euclidean Distance?

The Problem with Euclidean

Two documents about same topic:

Doc A (short): "machine learning" → [1, 1, 0, 0]

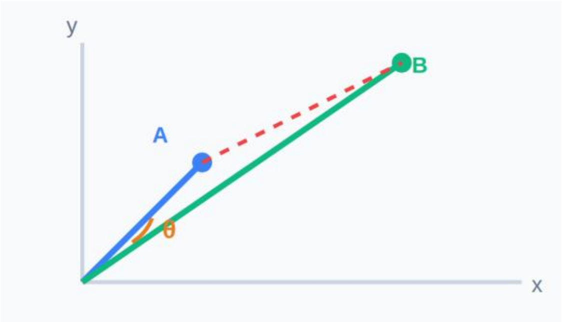
Doc B (long): "machine learning ×3" → [3, 3, 0, 0]

❌ Euclidean Distance

d(A, B) = $\sqrt{[(3-1)^2 + (3-1)^2]} = \sqrt{8} \approx 2.83$

Problem: Same topic but large distance!

Geometric Interpretation



Euclidean	Cosine
Line length	Angle θ

✓ **Same direction = Same topic**, regardless of length!

Cosine Similarity: The Formula

$$\cos(A, B) = (A \cdot B) / (\|A\| \times \|B\|)$$

$A \cdot B$ $= \Sigma\, a_i b_i$ (Dot product)

$\|A\|$ $= \sqrt{\Sigma\, a_i^2}$ (L2 norm)

Value Range

- **cos = 1:** Identical direction
- **cos = 0:** Orthogonal (unrelated)
- **cos = -1:** Opposite direction

Key Properties

- **Length-invariant:** Only direction matters
- **Symmetric:** cos(A,B) = cos(B,A)
- **Efficient:** O(d) computation
- **Non-negative for TF-IDF:** Range [0, 1]

Cosine Distance

d(A, B) = 1 - cos(A, B)

Converts similarity to distance. Range [0, 2]. Used in clustering algorithms.

Cosine Similarity: Example

Document A

"I love machine learning"

A = [1, 1, 1, 1, 0, 0]

Document B

"I love deep learning"

B = [1, 1, 0, 1, 1, 0]

Document C

"NLP is fun"

C = [0, 0, 0, 0, 0, 1]

Vocabulary: (I, love, machine, learning, deep, NLP)

Step-by-Step Calculation

cos(A, B):

$A \cdot B = 1\times1+1\times1+1\times0+1\times1+0\times1+0\times0 = \mathbf{3}$

$\|A\| = \sqrt{4} = 2$, $\|B\| = \sqrt{4} = 2$

cos(A,B) = 3/(2×2) = 0.75 ✓

cos(A, C):

$A \cdot C = 1\times0+1\times0+1\times0+1\times0+0\times0+0\times1 = \mathbf{0}$

$\|A\| = 2$, $\|C\| = 1$

cos(A,C) = 0/(2×1) = 0.00 ✗

cos(B, C):

$B \cdot C = 1\times0+1\times0+0\times0+1\times0+1\times0+0\times1 = \mathbf{0}$

$\|B\| = 2$, $\|C\| = 1$

cos(B,C) = 0/(2×1) = 0.00 ✗

Interpretation: A and B share 75% similarity (both about learning). C is orthogonal (no shared terms).

Cosine Similarity: Python Implementation

From Scratch (NumPy)

```
import numpy as np

def cosine_similarity(a, b):
    """Tính cosine similarity giữa hai vectors"""
    dot_product = np.dot(a, b)
    norm_a = np.linalg.norm(a)
    norm_b = np.linalg.norm(b)

    if norm_a == 0 or norm_b == 0:
        return 0.0

    return dot_product / (norm_a * norm_b)

# Ví dụ
D_A = np.array([0.5, 0.3, 0.0, 0.2])
D_B = np.array([0.4, 0.4, 0.0, 0.1])

sim = cosine_similarity(D_A, D_B)
print(f"cos(D_A, D_B) = {sim:.4f}")  # Output: 0.9615
```

Using sklearn

```
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# Tạo ma trận documents (mỗi hàng là 1 document)
documents = np.array([
    [0.5, 0.3, 0.0, 0.2],  # D_A
    [0.4, 0.4, 0.0, 0.1],  # D_B
    [0.0, 0.0, 0.6, 0.3]   # D_C
])

# Tính similarity matrix
sim_matrix = cosine_similarity(documents)

print("Similarity Matrix:")
print(sim_matrix)
```

PART 6

Dimensionality Reduction

"Principal Component Analysis (PCA)"

From High-Dimensional Sparse → Low-Dimensional Dense

10,000D → 300D

The Curse of Dimensionality

⚠️ Problem: High-Dimensional Vectors

With |V| = 50,000 words, each vector has **50,000 dimensions**! Most zeros → Sparse and Inefficient

[0, 0, 0.2, 0, 0, 0.1, ...]

💾 Storage

10K docs × 50K dims = ~2GB

⏱️ Computation

All pairs: $O(n^2d)$ → Very slow

📉 Sparsity

99% zeros, noise amplification

✓ Solution: Dimensionality Reduction

Goals:

- 50,000D → 100-300D
- Preserve semantics

Methods:

- PCA, SVD, LSA

Benefits:

- Faster, less memory

PCA: Principal Component Analysis

Definition

PCA finds **orthogonal directions** (principal components) that capture **maximum variance**. Project high-dimensional data onto first k components to reduce dimensions while preserving information.

📐 Example: 2D → 1D

PC1 captures most variance, PC2 captures remaining

🎯 Key Concepts

- **Principal Components:** Orthogonal directions of max variance
- **Eigenvalues:** Amount of variance captured by each PC
- **Eigenvectors:** Direction of each PC

📊 Variance Explained

Choose k components that explain ~90-95% of total variance. Scree plot helps visualize: plot eigenvalues and find "elbow".

PCA: Mathematical Formulation

Step 1: Center the Data

$$\tilde{X} = X - \mu$$

$\mu_j = (1/n)\ \Sigma_i\ x_{ij}$ (mean of each column)

Step 2: Covariance Matrix

$$C = (1/(n-1))\ \tilde{X}^T\tilde{X}$$

C is d×d matrix (d = original dimensions)

Step 3: Eigendecomposition

$$Cv = \lambda v$$

v: eigenvector (direction), λ: eigenvalue (variance)

Step 4: Sort Eigenvectors

Sort eigenvectors by eigenvalues (descending). First eigenvector = direction of maximum variance.

Step 5: Project to k Dimensions

$$X\_reduced = \tilde{X} \cdot V\_k$$

V_k = matrix of first k eigenvectors

Variance Explained Ratio

$$VE\_k = \Sigma_{i=1}{}^{k}\ \lambda_i\ /\ \Sigma_{j=1}{}^{d}\ \lambda_j$$

Choose k where VE_k ≥ 0.90-0.95

PCA: Python Implementation

📦 Using sklearn

```
import numpy as np
from sklearn.decomposition import PCA

# Dữ liệu gốc (4 documents, 1000 features)
X = np.random.rand(4, 1000)

# Khởi tạo PCA với 2 components
pca = PCA(n_components=2)

# Fit và transform
X_2d = pca.fit_transform(X)

print("Shape gốc:", X.shape)        # (4, 1000)
print("Shape sau PCA:", X_2d.shape)  # (4, 2)

# Xem variance explained
print("Variance ratio:", pca.explained_variance_ratio_)
print("Total variance:", sum(pca.explained_variance_ratio_))
```

⚠️ PCA Limitations for Sparse Data

Problem: PCA centers data (subtracts mean), destroying sparsity. For TF-IDF matrices, use **TruncatedSVD** instead - it works directly on sparse matrices without centering.

SVD: Alternative to Eigendecomposition

$$X = U \Sigma V^T$$

U (n×n)
Left singular vectors (docs)

Σ (n×d)
Singular values (diagonal)

V^T (d×d)
Right singular vectors (words)

✂️ Truncated SVD (LSA)

$$X \approx U\_k \Sigma\_k V\_k^T$$

Keep only k largest singular values. Best rank-k approximation (Eckart-Young theorem).

LSA: Apply truncated SVD to TF-IDF matrix → latent semantic space

📦 sklearn Implementation

```python
from sklearn.decomposition import TruncatedSVD
from sklearn.feature_extraction.text import TfidfVectorizer

# Corpus
corpus = [
    "machine learning algorithms",
    "deep learning neural networks",
    "natural language processing",
    "machine learning for NLP"
]

# TF-IDF vectorization
vectorizer = TfidfVectorizer()
X_tfidf = vectorizer.fit_transform(corpus)
print("TF-IDF shape:", X_tfidf.shape)  # (4, vocab_size)

# TruncatedSVD
svd = TruncatedSVD(n_components=2, random_state=42)
X_lsa = svd.fit_transform(X_tfidf)

print("LSA shape:", X_lsa.shape)  # (4, 2)
print("Variance explained:", svd.explained_variance_ratio_)
print("Total:", sum(svd.explained_variance_ratio_))
```

PART 7

Applications

"Putting Vector Space Models to Work"

Document Search

Word Similarity

Document Clustering

Document Search System

📊 Document Search Pipeline

1. Documents → 2. TF-IDF → 3. Query → 4. Similarity → 5. Rank

💻 Python Implementation (handout)

Word Similarity with Vector Spaces

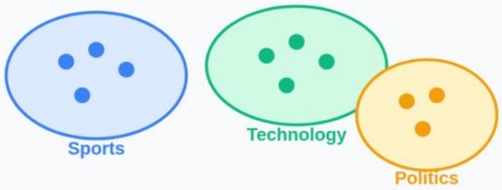
🎯 Finding Similar Words

Words in similar contexts have similar vectors. Use cosine similarity to find related words.

Document Clustering

🎯 Grouping Similar Documents

Use TF-IDF vectors with clustering algorithms to group documents by topic without labels.



✓ Pros

Unsupervised, scalable, interpretable clusters

✗ Cons

Need to choose k, sensitive to initialization

Use Cases

News categorization • Customer feedback grouping • Email organization • Research paper clustering

PART 8

Summary & Lab Practice

Key Concepts Formulas Lab Practice Exercise 1

Key Concepts Summary

1. Word Representations

- One-hot: sparse, no similarity
- Distributional: context-based
- Count vs prediction methods

2. Co-occurrence Matrices

- Term-document: document vectors
- Word-word: word vectors
- Window size impacts semantics

3. TF-IDF Weighting

- TF: local term importance
- IDF: global term rarity
- Balances frequency vs specificity

4. Cosine Similarity

- Measures angle, not magnitude
- Range: [-1, 1] or [0, 1] for TF-IDF
- Cosine distance = 1 - cosine

5. Dimensionality Reduction

- PCA: eigendecomposition
- SVD: works on sparse matrices
- LSA: SVD on TF-IDF

6. Applications

- Document search: query→rank
- Word similarity: nearest neighbors
- Clustering: unsupervised grouping

Key Formulas Reference

📊 TF-IDF

$$\text{tf-idf}(t,d) = \text{tf}(t,d) \times \log(N/\text{df\_t})$$

tf(t,d) = count of term t in doc d; N = total docs; df_t = docs containing t

📐 Cosine Similarity

$$\cos(A,B) = (A \cdot B) / (\|A\| \times \|B\|)$$

$A \cdot B = \Sigma a_i b_i$ (dot product); $\|A\| = \sqrt{(\Sigma a_i^2)}$ (L2 norm)

🔄 PCA (Eigendecomposition)

$$Cv = \lambda v$$

C = covariance matrix; v = eigenvector; λ = eigenvalue (variance)

✂️ SVD

$$X = U\Sigma V^T$$

U = left singular vectors; Σ = singular values; V = right singular vectors

💡 TF-IDF captures term importance • Cosine measures angle • PCA/SVD reduce dimensions

Lab Practice Overview

Task 1: TF-IDF Vectors

- Build TF-IDF vectors from scratch
- Compare with sklearn TfidfVectorizer
- Visualize top terms per document

Task 2: Cosine Similarity

- Implement cosine similarity function
- Build document similarity matrix
- Create similarity heatmap

Task 3: Document Search

- Build search engine with TF-IDF
- Implement query → TF-IDF → rank
- Evaluate with sample queries

🌟 Challenge

Apply TruncatedSVD to reduce TF-IDF dimensions, then use t-SNE for 2D visualization. Color points by document category to see if similar documents cluster together.

📋 Exercise 1 Release

Sentiment Analysis & Vector Spaces

Weight: 10%

📅 Release	⏰ Deadline	📆 Duration
End of Session 3	**Beginning of Session 5**	**2 Weeks**

Tasks

Part A

Sentiment classifier with Logistic Regression or Naïve Bayes

Part B

Build word vectors, compute cosine similarity

Part C

Document search with TF-IDF and cosine similarity

📦 Deliverables

Jupyter notebook with code + explanations • Short report (2-3 pages) with results and analysis