

CSCI E-89B: Introduction to Natural Language Processing

Lecture 05: TF-IDF and Word Embeddings

Harvard Extension School

Fall 2024

- **Course:** CSCI E-89B: Introduction to Natural Language Processing
- **Week:** Lecture 05
- **Instructor:** Dmitry Kurochkin
- **Objective:** Master TF-IDF representation for text analysis and understand word embedding techniques including Word2Vec and GloVe

Contents

1 Quiz Review: Weight Sharing in Neural Networks

Lecture Overview

This lecture covers TF-IDF (Term Frequency-Inverse Document Frequency) as an improvement over bag-of-words, introduces the concept of word embeddings, and explores Word2Vec and GloVe as advanced embedding techniques that capture semantic relationships.

1.1 Where Does Weight Sharing Occur?

Key Summary

Weight sharing is a key technique to reduce parameters in neural networks. It occurs in:

- **Recurrent Neural Networks (RNNs):** Same weights across all time steps
- **Convolutional Neural Networks (CNNs):** Same filter weights at all spatial positions
- **NOT in Fully Connected Networks:** Each connection has unique weights

1.1.1 Fully Connected Networks

In a fully connected (dense) network:

- Every neuron is connected to every neuron in adjacent layers
- Each connection has its **own unique weight** w_{ij}
- No weight sharing—maximum flexibility, maximum parameters

1.1.2 Recurrent Neural Networks

When unrolled through time, RNNs use the same weights:

- Time step 1: Uses W
- Time step 2: Uses **same** W
- Time step T : Uses **same** W

Why Weight Sharing in RNNs?

We assume that the relationship between signals is **independent of time**. The way we process word 1 should be the same as how we process word 100. This reduces parameters dramatically and enables processing of variable-length sequences.

1.1.3 Convolutional Neural Networks

CNNs share weights across **spatial positions**:

- Filter applied at position (0,0): Uses weights W
- Filter applied at position (1,1): Uses **same** W
- Filter “slides” across the image with identical weights

Example: Analogy: Using the Same Eyes

When you look at the upper-left corner of an image, you use the same eyes as when you look at the lower-right corner. Similarly, CNNs use the same filter (same “eyes”) at every position.

1.2 N-gram Counting

Critical: N-grams are Overlapping

When computing bigrams, tokens **overlap**:

- Text: “port assembly line reduced costs”
- Bigrams: (port, assembly), (assembly, line), (line, reduced), (reduced, costs)
- Count: $n - 1$ bigrams for n words

Bigram Vocabulary Explosion

In practice, the number of bigrams is **much larger** than unigrams:

- “Computer” appears once as a unigram
- But “computer is,” “computer was,” “computer does,” etc. are all different bigrams
- Vocabulary grows dramatically, increasing sparsity and computation

1.3 CNN Padding and Strides

Definition: Preserving Dimensions

To preserve input dimensions in a CNN:

- **Padding:** Must be `same` (add zeros around input)
- **Strides:** Must be 1 (move one position at a time)

If strides > 1, dimensions will be reduced even with `same` padding.

2 TF-IDF: Term Frequency-Inverse Document Frequency

2.1 Motivation: Beyond Bag of Words

Bag of words treats all words equally by just counting occurrences. But consider:

- In Boston local news, “Boston” appears everywhere—it’s not informative
- A rare word like “hurricane” only appears in certain documents—very informative

Key Summary

TF-IDF addresses this by:

1. Counting how often a term appears in a document (TF)
2. Discounting terms that appear in many documents (IDF)
3. Multiplying: $\text{TF} \times \text{IDF}$

2.2 Term Frequency (TF)

Definition: Term Frequency

$$\text{TF}(t, d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

Note: This is a *frequency* (ratio), not just a count. It normalizes by document length.

Example: TF Calculation

Document 1: “cat cat dog” (3 terms)

$$\begin{aligned}\text{TF(cat, doc1)} &= 2/3 = 0.67 \\ \text{TF(dog, doc1)} &= 1/3 = 0.33 \\ \text{TF(mouse, doc1)} &= 0/3 = 0.00\end{aligned}$$

2.3 Inverse Document Frequency (IDF)

Definition: Inverse Document Frequency

$$\text{IDF}(t) = \ln \left(\frac{\text{Total number of documents}}{\text{Number of documents containing term } t} \right)$$

Key insight: IDF is computed from the **training corpus** and remains fixed, even when processing test documents.

2.3.1 IDF Properties

- Term in **every** document: $\text{IDF} = \ln(N/N) = \ln(1) = 0$
- Term in **half** documents: $\text{IDF} = \ln(N/(N/2)) = \ln(2) \approx 0.69$
- Term in **one** document: $\text{IDF} = \ln(N/1) = \ln(N)$ (large!)

Example: IDF Calculation

Corpus with 4 documents:

- Cat appears in 2 documents: $\text{IDF} = \ln(4/2) = 0.69$
- Dog appears in 4 documents: $\text{IDF} = \ln(4/4) = 0$ (useless!)
- Mouse appears in 3 documents: $\text{IDF} = \ln(4/3) = 0.29$

Interpretation: Cat is a good discriminator (occurs sometimes), dog is useless (occurs everywhere).

2.4 TF-IDF Computation

Definition: TF-IDF

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

High TF-IDF means: term appears frequently in this document but rarely across the corpus.

Critical: Train vs Test Data

When computing TF-IDF for test documents:

- **TF:** Computed from the test document itself
- **IDF:** Always uses the training corpus values (pre-computed)

Rationale: IDF measures how discriminative a term is across the corpus. We can't use test data to compute this—that would be data leakage!

2.5 TF-IDF Variations

2.5.1 Smoothed IDF (sklearn default)

$$\text{IDF}_{\text{smooth}}(t) = \ln \left(\frac{N + 1}{df(t) + 1} \right) + 1$$

- +1 in denominator: Prevents division by zero
- +1 added to result: Ensures even common words have **some** weight

2.5.2 Why Add +1 to the Result?

Without the +1, words appearing everywhere get $\text{IDF} = 0$, making their TF-IDF = 0. Adding 1 ensures:

- Common words still contribute (just less than rare words)
- No term is completely ignored

2.6 L2 Normalization

Definition: L2 Normalization

After computing TF-IDF values, normalize the vector to unit length:

$$\tilde{v} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{\sum_i v_i^2}}$$

This places all document vectors on a **unit hypersphere**.

2.6.1 Why Normalize?

1. **Cosine similarity becomes dot product:** For unit vectors, $\cos(\theta) = a \cdot b$
2. **Scale invariance:** Long documents don't dominate short ones
3. **Numerical stability:** All values in similar range

```

1 from sklearn.feature_extraction.text import TfidfVectorizer
2
3 # Default: lowercase=True, L2 normalization
4 vectorizer = TfidfVectorizer()
5 X = vectorizer.fit_transform(documents)
6
7 # Check: each row has L2 norm of 1
8 import numpy as np
9 print(np.linalg.norm(X[0].toarray())) # Should be ~1.0

```

2.7 Document Similarity with Cosine

Definition: Cosine Similarity

$$\text{similarity}(d_1, d_2) = \cos(\theta) = \frac{d_1 \cdot d_2}{\|d_1\| \|d_2\|}$$

For L2-normalized vectors: $\text{similarity} = d_1 \cdot d_2$ (just dot product!)

Interpreting Cosine Similarity

- $\cos(\theta) = 1$ Identical direction (very similar)
- $\cos(\theta) = 0$ Orthogonal (no common terms)
- $\cos(\theta) = -1$ Opposite direction (rare in TF-IDF since values ≥ 0)

3 Word Embeddings: Dense Representations

3.1 Limitations of One-Hot Encoding

One-hot encoding represents each word as a sparse vector:

- “cat” = [0, 0, 1, 0, 0, ..., 0] (10,000 dimensions)
- “dog” = [0, 0, 0, 1, 0, ..., 0] (10,000 dimensions)

Problems:

1. **High dimensionality:** Vector size = vocabulary size
2. **Sparse:** Mostly zeros, computationally wasteful
3. **No semantics:** “cat” and “dog” are as different as “cat” and “table”

3.2 What is an Embedding?

Definition: Word Embedding

A learned mapping from discrete words to dense, continuous vectors in low-dimensional space:

$$\text{embed} : \{1, 2, \dots, V\} \rightarrow \mathbb{R}^d$$

where V = vocabulary size, d = embedding dimension (typically 50–300).

3.3 Embedding as a Neural Network Layer

Example: Embedding Layer Mechanics

Consider vocabulary size 3, embedding dimension 2:

Embedding matrix W (learnable parameters):

$$W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

One-hot input for word 2: $x = [0, 1, 0]$

Embedding output: $x \cdot W = [w_{21}, w_{22}]$ (just row 2!)

Efficient implementation: Instead of matrix multiplication, just look up row by index.

Embedding Layer

The embedding layer is mathematically a linear layer without bias. But because input is one-hot:

- Matrix multiplication with one-hot = selecting one row
- Implementation: Direct lookup by index (much faster)
- Parameters: $V \times d$ (vocabulary size \times embedding dimension)

3.4 Training Embeddings

Two approaches:

1. **Task-specific:** Train embedding layer as part of your model (classification, etc.)
2. **Pre-trained:** Use Word2Vec, GloVe, etc. trained on large corpora

Task-Specific vs Pre-trained Tradeoffs

Task-specific embeddings:

- Optimized for your specific task
- May not capture general semantics
- Synonyms might not be similar if task doesn't require it

Pre-trained embeddings (Word2Vec, GloVe):

- Capture general semantic relationships
- Transfer learning: works even with limited data
- May include biases from training corpus

4 Word2Vec: Learning from Context

4.1 The Key Idea

Key Summary

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

Word2Vec learns embeddings by predicting words from their context (or vice versa).

4.2 Two Architectures

4.2.1 Skip-gram: Predict Context from Word

- **Input:** Current word
- **Output:** Surrounding context words
- **Example:** Given “cat,” predict [“the”, “sat”, “on”, “mat”]

4.2.2 CBOW (Continuous Bag of Words): Predict Word from Context

- **Input:** Surrounding context words
- **Output:** Current word
- **Example:** Given [“the”, “sat”, “on”, “mat”], predict “cat”

Example: Window Size

With window size 5:

- Consider 5 words before and 5 words after
- Total context: up to 10 words (plus target)
- Larger window = broader context, slower training

4.3 Why Word2Vec Captures Semantics

Because embeddings are trained to predict context:

- Words appearing in similar contexts get similar embeddings
- “king” and “queen” appear in similar contexts \Rightarrow similar vectors
- Synonyms naturally cluster together

4.4 Word Analogies

Definition: Vector Arithmetic

Word2Vec embeddings enable semantic arithmetic:

$$\vec{\text{king}} - \vec{\text{man}} + \vec{\text{woman}} \approx \vec{\text{queen}}$$

The vector $\vec{\text{man}} - \vec{\text{woman}}$ encodes the concept of “gender.”

```

1 from gensim.models import Word2Vec
2
3 # Train model
4 model = Word2Vec(sentences, vector_size=100, window=5)
5
6 # Word analogy
7 result = model.wv.most_similar(
8     positive=['king', 'woman'],
9     negative=['man'],
10    topn=1
11 )
12 print(result) # [('queen', 0.85)]
```

4.5 Finding Similar Words

```

1 # Find words most similar to "destruction"
2 similar = model.wv.most_similar('destruction', topn=5)
3 # Might return: [('flood', 0.91), ('damage', 0.87), ...]
4
5 # Similarity is cosine of angle between vectors
6 similarity = model.wv.similarity('cat', 'dog')
7 # Returns value between -1 and 1
```

5 GloVe: Global Vectors for Word Representation

5.1 Motivation

Word2Vec only looks at **local** context windows. GloVe uses **global** co-occurrence statistics from the entire corpus.

Definition: Co-occurrence Matrix

Build a matrix X where $X_{ij} = \text{number of times word } i \text{ appears near word } j \text{ across the entire corpus.}$

5.2 The GloVe Objective

GloVe learns embeddings such that:

$$\vec{w}_i \cdot \vec{w}_j + b_i + b_j = \log(X_{ij})$$

Interpretation: The dot product of word vectors should approximate the log of their co-occurrence count.

Why Log Co-occurrence?

- Raw counts vary wildly (some pairs co-occur millions of times)
- Log compresses the range
- Dot product naturally represents similarity

5.3 Word2Vec vs GloVe

Aspect	Word2Vec	GloVe
Training	Local context windows	Global co-occurrence matrix
Approach	Predictive (neural network)	Count-based (matrix factorization)
Memory	Streams through corpus	Needs entire co-occurrence matrix
Speed	Slower per epoch	Faster convergence
Results	Very similar quality	Very similar quality

5.4 Using Pre-trained Embeddings

```

1 # Load pre-trained GloVe
2 import gensim.downloader as api
3 glove = api.load('glove-wiki-gigaword-100')
4
5 # Use like Word2Vec
6 similar = glove.most_similar('computer', topn=5)

```

```
7 vector = glove['computer'] # 100-dimensional vector
```

6 Practical Considerations

6.1 Embedding Dimension

Choosing Embedding Dimension

- **Too small (e.g., 10)**: Can't capture rich semantics
- **Too large (e.g., 1000)**: Overfitting, slow training
- **Common choices**: 50, 100, 200, 300
- **Rule of thumb**: Try 100–300 for most applications

6.2 Out-of-Vocabulary (OOV) Words

OOV Problem

Pre-trained embeddings only cover words in their training vocabulary:

- New words (“ChatGPT”) won’t have embeddings
- Misspellings won’t be recognized
- Rare technical terms may be missing

Solutions:

- Use subword embeddings (FastText, BPE)
- Map unknown words to special “UNK” token
- Train domain-specific embeddings

6.3 Bias in Embeddings

Societal Biases

Embeddings learn from text, including biases:

- “doctor” may be closer to “man” than “woman”
- Historical texts contain outdated stereotypes
- These biases affect downstream applications

Mitigation: Debiasing techniques, careful data curation, bias auditing.

6.4 Aggregating Word Embeddings

For classification without sequence models:

```

1 def document_embedding(doc, model):
2     """Average word embeddings for document."""
3     vectors = [model[word] for word in doc if word in model]
4     if vectors:
5         return np.mean(vectors, axis=0)
6     return np.zeros(model.vector_size)

```

Averaging Embeddings

Simple averaging:

- Loses word order (like bag of words)
- Works surprisingly well for classification
- Fast and simple baseline

Better approaches: TF-IDF weighted averaging, use RNN/Transformer instead.

7 TF-IDF Implementation Guide

7.1 Basic TF-IDF with sklearn

```

1 from sklearn.feature_extraction.text import TfidfVectorizer
2
3 corpus = [
4     "The cat sat on the mat",
5     "The dog sat on the log",
6     "Cats and dogs are friends"
7 ]
8
9 # Create and fit vectorizer
10 vectorizer = TfidfVectorizer()
11 X = vectorizer.fit_transform(corpus)
12
13 # View vocabulary
14 print(vectorizer.get_feature_names_out())
15 # ['and', 'are', 'cat', 'cats', 'dog', 'dogs', 'friends', ...]
16
17 # View TF-IDF matrix
18 print(X.toarray())

```

7.2 Customizing TfidfVectorizer

```

1 vectorizer = TfidfVectorizer(
2     lowercase=True,                      # Convert to lowercase
3     stop_words='english',                # Remove English stop words
4     ngram_range=(1, 2),                  # Unigrams and bigrams
5     max_features=10000,                 # Top 10k features only
6     min_df=2,                          # Ignore terms in < 2 docs
7     max_df=0.95,                      # Ignore terms in > 95% docs
8     norm='l2'                           # L2 normalization (default)
9 )

```

7.3 Using with Preprocessing

```

1 from nltk.stem import PorterStemmer
2 from nltk.tokenize import word_tokenize
3
4 stemmer = PorterStemmer()
5
6 def preprocess(text):
7     tokens = word_tokenize(text.lower())
8     stems = [stemmer.stem(t) for t in tokens]
9     return ' '.join(stems)
10
11 # Preprocess documents

```

```
12 processed = [preprocess(doc) for doc in corpus]
13
14 # Then apply TF-IDF
15 vectorizer = TfidfVectorizer()
16 X = vectorizer.fit_transform(processed)
```

8 One-Page Summary

TF-IDF

Term Frequency: $\text{TF}(t, d) = \frac{\text{count}(t, d)}{\text{len}(d)}$

Inverse Document Frequency: $\text{IDF}(t) = \ln \left(\frac{N}{df(t)} \right)$

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

Key insight: High TF-IDF = frequent in document, rare in corpus

Word Embeddings

One-hot: Sparse, high-dimensional, no semantics

Embedding: Dense, low-dimensional, captures meaning

Parameters: vocabulary_size × embedding_dim

Word2Vec

Skip-gram: Predict context from word

CBOW: Predict word from context

Key property: $\vec{\text{king}} - \vec{\text{man}} + \vec{\text{woman}} \approx \vec{\text{queen}}$

GloVe

Approach: Learn from global co-occurrence matrix

Objective: $\vec{w}_i \cdot \vec{w}_j \approx \log(\text{co-occurrence})$

Result: Similar quality to Word2Vec

Cosine Similarity

$$\cos(\theta) = \frac{a \cdot b}{\|a\| \|b\|}$$

For unit vectors: $\cos(\theta) = a \cdot b$

Used to measure word/document similarity

9 Glossary

Term	Definition
CBOW	Continuous Bag of Words: Word2Vec variant predicting word from context
Co-occurrence Matrix	Matrix counting how often word pairs appear together
Cosine Similarity	Similarity measure based on angle between vectors
Document Frequency	Number of documents containing a term
Embedding	Dense vector representation of discrete items
GloVe	Global Vectors: embedding method using co-occurrence statistics
IDF	Inverse Document Frequency: $\log(\text{total docs} / \text{docs with term})$
L2 Normalization	Scaling vector to unit length using Euclidean norm
One-hot Encoding	Sparse vector with single 1 indicating category
Skip-gram	Word2Vec variant predicting context from word
TF	Term Frequency: count of term / document length
TF-IDF	Term Frequency-Inverse Document Frequency
Weight Sharing	Using same weights at different positions (RNN, CNN)
Window Size	Number of context words considered around target
Word2Vec	Neural embedding method learning from local context
