Lab 5: Sparse Vector (Embedding) — Natural Language Processing

Author:

# Program Description

This lab implements classical sparse vector methods for text: TF–IDF scoring, Euclidean normalization, cosine-similarity word neighborhoods, and Pointwise Mutual Information (PMI/PPMI). A Streamlit app provides interactive exploration across three tasks.

Problem as implemented:

* Compute TF–IDF weights for a small term–document table and score user queries by summing per-term TF–IDF.
* Apply Euclidean (L2) normalization to raw TF values.
* Build word embeddings from a small corpus using simple count-based vectors (TF or TF–IDF), then find nearest neighbors by cosine similarity and visualize with TruncatedSVD in 2D.
* Replace TF/TF–IDF with PMI/PPMI values to score queries when co-occurrences are informative.

# Program Logic

Main concepts and algorithms:

1. Vectorization: scikit-learn CountVectorizer/TfidfVectorizer with English stop-word removal and min\_df filtering.
2. TF–IDF: tfidf(t, d) = tf(t, d) × idf(t), with sublinear TF and smoothed IDF for robustness.
3. L2 Normalization: For each document vector v, use v / ||v||₂ to compare fairly across different lengths.
4. Cosine Similarity: sim(x, y) = (x · y) / (||x||₂ ||y||₂); since rows are normalized, this is just the dot product.
5. Dimensionality Reduction: TruncatedSVD projects high-dimensional sparse word vectors into 2D for plotting.
6. PMI/PPMI: PMI(w, d) = log2((tf(w,d)·N)/(count(w)·count(d))); PPMI = max(0, PMI). Zeros and non-finite values are set to 0.
7. Caching and interactivity: Streamlit cache functions to keep UI responsive.

# Question 1: TF–IDF and Euclidean Normalization

**Term Frequency (TF) Table**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Doc1 | Doc2 | Doc3 |
| car | 27 | 4 | 24 |
| auto | 3 | 33 | 0 |
| insurance | 0 | 33 | 29 |
| best | 14 | 0 | 17 |

**Inverse Document Frequency (IDF) Table**

|  |  |
| --- | --- |
|  | idf |
| car | 1.65 |
| auto | 2.08 |
| insurance | 1.62 |
| best | 1.50 |

**Computed TF–IDF Matrix (tf × idf)**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Doc1 | Doc2 | Doc3 |
| car | 44.55 | 6.60 | 39.60 |
| auto | 6.24 | 68.64 | 0.00 |
| insurance | 0.00 | 53.46 | 46.98 |
| best | 21.00 | 0.00 | 25.50 |

**Query scoring (sum of TF–IDF for query terms):**

**Query: 'car insurance'**

|  |  |
| --- | --- |
|  | Score |
| Doc3 | 86.58 |
| Doc2 | 60.06 |
| Doc1 | 44.55 |

**Query: 'best car'**

|  |  |
| --- | --- |
|  | Score |
| Doc1 | 65.55 |
| Doc3 | 65.10 |
| Doc2 | 6.60 |

Conclusion for 'car insurance': Doc3 is most relevant.

Conclusion for 'best car': Doc1 is most relevant.

**Document L2 Norms (TF)**

|  |  |
| --- | --- |
|  | L2 Norm (TF) |
| Doc1 | 30.56 |
| Doc2 | 46.84 |
| Doc3 | 41.30 |

**L2-Normalized TF Matrix**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Doc1 | Doc2 | Doc3 |
| car | 0.88 | 0.09 | 0.58 |
| auto | 0.10 | 0.70 | 0.00 |
| insurance | 0.00 | 0.70 | 0.70 |
| best | 0.46 | 0.00 | 0.41 |

# Question 2: Cosine Similarity — Nearest Neighbors

Vectorization: TF–IDF with stop\_words="english", min\_df=2 on the default 5-document corpus.

Top-3 nearest neighbors (by cosine similarity) for selected words:

**Neighbors of 'best'**

|  |  |  |
| --- | --- | --- |
|  | Neighbor | Similarity |
| 0 | car | 0.8165 |
| 1 | coverage | 0.8165 |
| 2 | new | 0.8165 |

**Neighbors of 'car'**

|  |  |  |
| --- | --- | --- |
|  | Neighbor | Similarity |
| 0 | coverage | 1.0000 |
| 1 | new | 1.0000 |
| 2 | best | 0.8165 |

Interpretation: words that co-occur in the same subset of documents (e.g., best with coverage/new/car) tend to have high cosine similarity.

# Question 3: Pointwise Mutual Information (PMI/PPMI)

**PMI Matrix (signed)**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Doc1 | Doc2 | Doc3 |
| car | 1.04 | -2.39 | 0.20 |
| auto | -1.52 | 1.27 | 0.00 |
| insurance | 0.00 | 0.48 | 0.30 |
| best | 0.92 | 0.00 | 0.53 |

**PPMI Matrix (non-negative)**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Doc1 | Doc2 | Doc3 |
| car | 1.04 | 0.00 | 0.20 |
| auto | 0.00 | 1.27 | 0.00 |
| insurance | 0.00 | 0.48 | 0.30 |
| best | 0.92 | 0.00 | 0.53 |

**Query (PMI): 'car insurance'**

|  |  |
| --- | --- |
|  | Score |
| Doc1 | 1.038 |
| Doc3 | 0.496 |
| Doc2 | -1.903 |

**Query (PPMI): 'car insurance'**

|  |  |
| --- | --- |
|  | Score |
| Doc1 | 1.038 |
| Doc3 | 0.496 |
| Doc2 | 0.484 |

PMI ranking: ['Doc1', 'Doc3', 'Doc2']

PPMI ranking: ['Doc1', 'Doc3', 'Doc2'] (ties broken by value)

# Program: Python Source Code (key procedures / full app)

import streamlit as st

import pandas as pd

import numpy as np

from sklearn.feature\_extraction.text import TfidfVectorizer, CountVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

from sklearn.decomposition import TruncatedSVD

import matplotlib.pyplot as plt

from io import StringIO

from sklearn.preprocessing import normalize

import time

# --- Page Configuration ---

st.set\_page\_config(layout="wide", page\_title="Lab 5: Sparse Vectors")

# --- Default Data ---

TF\_DEFAULT\_DATA = {

'term': ['car', 'auto', 'insurance', 'best'],

'Doc1': [27, 3, 0, 14],

'Doc2': [4, 33, 33, 0],

'Doc3': [24, 0, 29, 17]

}

TF\_DEFAULT\_DF = pd.DataFrame(TF\_DEFAULT\_DATA).set\_index('term')

IDF\_DEFAULT\_DATA = {'term': ['car', 'auto', 'insurance', 'best'], 'idf': [1.65, 2.08, 1.62, 1.5]}

IDF\_DEFAULT\_DF = pd.DataFrame(IDF\_DEFAULT\_DATA).set\_index('term')

CORPUS\_DEFAULT = [

"The best car insurance provides great coverage for your new auto.",

"Find cheap auto insurance by comparing quotes from many providers.",

"A new car is a major investment; protect it with the best coverage.",

"The best recipe for apple pie is a family secret.",

"Good food makes for a good mood, especially a sweet pie."

]

# =======================

# CACHING HELPERS

# =======================

@st.cache\_data(show\_spinner=False)

def cache\_vectorize(corpus, rep, min\_df, max\_features):

if rep == "TF counts":

vectorizer = CountVectorizer(stop\_words='english', min\_df=min\_df,

max\_features=(None if max\_features == 0 else max\_features))

else:

vectorizer = TfidfVectorizer(stop\_words='english', norm=None, sublinear\_tf=True, smooth\_idf=True,

min\_df=min\_df, max\_features=(None if max\_features == 0 else max\_features))

matrix = vectorizer.fit\_transform(corpus) # docs x terms (sparse)

vocab = vectorizer.get\_feature\_names\_out()

# Build word vectors (terms x docs) and L2-normalize each word vector

word\_vectors = normalize(matrix.T, norm='l2', axis=1)

return matrix, word\_vectors, vocab

@st.cache\_data(show\_spinner=False)

def cache\_svd(word\_vectors, n\_components=2, random\_state=42):

svd = TruncatedSVD(n\_components=n\_components, random\_state=random\_state)

vectors\_2d = svd.fit\_transform(word\_vectors)

# Center for readability

vectors\_2d = vectors\_2d - vectors\_2d.mean(axis=0, keepdims=True)

return vectors\_2d

@st.cache\_data(show\_spinner=False)

def cache\_cosine\_sim(selected\_vector, word\_vectors):

sims = cosine\_similarity(selected\_vector, word\_vectors).ravel()

return sims

@st.cache\_data(show\_spinner=False)

def cache\_pmi\_matrix(tf\_df: pd.DataFrame, use\_ppmi: bool):

tf = tf\_df.astype(float).values # T x D

N = tf.sum()

if N == 0:

return pd.DataFrame(0, index=tf\_df.index, columns=tf\_df.columns)

term\_counts = tf.sum(axis=1, keepdims=True) # T x 1

doc\_counts = tf.sum(axis=0, keepdims=True) # 1 x D

denom = term\_counts @ doc\_counts # T x D

with np.errstate(divide='ignore', invalid='ignore'):

pmi = np.log2((tf \* N) / denom)

# Zero-out invalids and zero cells

pmi[~np.isfinite(pmi)] = 0.0

pmi[tf == 0] = 0.0

if use\_ppmi:

pmi = np.maximum(0.0, pmi)

return pd.DataFrame(pmi, index=tf\_df.index, columns=tf\_df.columns)

# =================================================================================

# --- Question 1: TF-IDF and Normalization Page ---

# =================================================================================

def page\_question\_1():

st.header("Question 1: TF-IDF and Euclidean Normalization")

st.markdown("---")

st.subheader("1a. TF-IDF Calculation and Query Scoring")

col1, col2 = st.columns(2)

with col1:

st.write("Term Frequency (TF) Table")

st.dataframe(TF\_DEFAULT\_DF)

with col2:

st.write("Inverse Document Frequency (IDF) Table")

st.dataframe(IDF\_DEFAULT\_DF)

tfidf\_df = TF\_DEFAULT\_DF.multiply(IDF\_DEFAULT\_DF['idf'], axis=0)

st.write("Calculated TF-IDF Matrix")

st.dataframe(tfidf\_df.style.format("{:.2f}"))

query = st.text\_input("Enter a query (e.g., 'car insurance' or 'best car')", "car insurance").lower()

query\_terms = [t for t in query.split() if t in tfidf\_df.index]

if not query\_terms:

st.warning("None of the query terms are in the vocabulary.")

else:

scores = tfidf\_df.loc[query\_terms].sum(axis=0)

score\_df = pd.DataFrame(scores, columns=['Score']).sort\_values(by='Score', ascending=False)

st.dataframe(score\_df.style.format("{:.2f}"))

st.success(f"Conclusion: Document {score\_df.index[0]} is the most relevant.")

st.markdown("---")

st.subheader("1b. Euclidean Normalization (on TF)")

tf\_norms = np.sqrt(np.sum(TF\_DEFAULT\_DF\*\*2, axis=0))

norm\_df = pd.DataFrame(tf\_norms, columns=['L2 Norm (TF)'])

st.dataframe(norm\_df.style.format("{:.2f}"))

normalized\_tf = TF\_DEFAULT\_DF.div(tf\_norms, axis=1)

st.write("L2-Normalized TF Matrix")

st.dataframe(normalized\_tf.style.format("{:.2f}"))

# =================================================================================

# --- Question 2: Cosine Similarity Page ---

# =================================================================================

def page\_question\_2():

st.header("Question 2: Cosine Similarity")

st.markdown("---")

st.subheader("Data Source")

data\_source = st.radio("Choose a corpus:", ("Use default corpus", "Input your own corpus"))

if data\_source == "Input your own corpus":

user\_corpus\_text = st.text\_area(

"Enter your documents, one per line.",

height=150,

placeholder="The quick brown fox...\nAnother document about cats and dogs...\nAnd a third one here."

)

corpus = [doc.strip() for doc in user\_corpus\_text.split('\n') if doc.strip()]

if not corpus:

st.info("Please enter at least one document to proceed.")

return

else:

corpus = CORPUS\_DEFAULT

st.subheader("Vector representation")

rep = st.radio("Choose representation:", ("TF counts", "TF-IDF"))

min\_df = st.number\_input("Min document frequency (filter rare words)", min\_value=1, max\_value=10, value=2, step=1)

max\_feat = st.number\_input("Max features (0 = unlimited)", min\_value=0, max\_value=10000, value=0, step=100)

with st.spinner("Vectorizing corpus..."):

t0 = time.perf\_counter()

try:

matrix, word\_vectors, vocab = cache\_vectorize(corpus, rep, min\_df, max\_feat)

except ValueError:

st.error("Could not process the corpus. Please ensure it's not empty or contains only stop words.")

return

st.caption(f"Vectorization time: {(time.perf\_counter()-t0)\*1000:.1f} ms")

with st.expander("View matrix (may be slow for large vocabularies)"):

show\_matrix = st.checkbox("Render full dense matrix (docs × terms)", value=False)

if show\_matrix:

max\_cells = 5000

cells = matrix.shape[0] \* matrix.shape[1]

if cells > max\_cells:

st.warning(f"Matrix too big to render densely ({cells} cells > {max\_cells}). "

"Increase filters or uncheck this option.")

else:

st.dataframe(pd.DataFrame(matrix.toarray(), columns=vocab,

index=[f"Doc {i+1}" for i in range(len(corpus))]))

if len(vocab) < 2:

st.warning("Vocabulary too small (<2 words). Provide a richer corpus.")

return

col1, col2 = st.columns([1, 2])

with col1:

selected\_word = st.selectbox("Select a word:", sorted(vocab))

word\_idx = list(vocab).index(selected\_word)

t0 = time.perf\_counter()

sims = cosine\_similarity(word\_vectors[word\_idx], word\_vectors).ravel()

st.caption(f"Similarity time: {(time.perf\_counter()-t0)\*1000:.1f} ms")

sim\_df = pd.DataFrame({'word': vocab, 'similarity': sims})

sim\_df = sim\_df[sim\_df['word'] != selected\_word].sort\_values(by='similarity', ascending=False)

max\_k = max(2, min(20, len(vocab) - 1))

k = st.slider("Number of neighbors to show", 2, max\_k, min(5, max\_k))

st.write(f"Top {k} nearest neighbors to '{selected\_word}':")

st.dataframe(sim\_df.head(k).reset\_index(drop=True))

with col2:

st.write("2D Visualization (TruncatedSVD on row-normalized word vectors)")

with st.spinner("Projecting to 2D..."):

t0 = time.perf\_counter()

#vectors\_2d = cache\_svd(word\_vectors, n\_components=2, random\_state=42)

svd = TruncatedSVD(n\_components=2, random\_state=42)

vectors\_2d = svd.fit\_transform(word\_vectors)

vectors\_2d = vectors\_2d - vectors\_2d.mean(axis=0, keepdims=True)

st.caption(f"SVD time: {(time.perf\_counter()-t0)\*1000:.1f} ms")

top\_neighbors = set(sim\_df.head(k)['word'].tolist())

idx\_map = {w: i for i, w in enumerate(vocab)}

colors = ['red' if w == selected\_word else ('orange' if w in top\_neighbors else 'lightgray') for w in vocab]

fig, ax = plt.subplots(figsize=(10, 8))

ax.scatter(vectors\_2d[:, 0], vectors\_2d[:, 1], c=colors, alpha=0.85)

for w in [selected\_word] + list(top\_neighbors):

i = idx\_map[w]

ax.annotate(w, (vectors\_2d[i, 0], vectors\_2d[i, 1]), fontsize=10,

color=('red' if w == selected\_word else 'black'))

ax.set\_xlabel("Component 1")

ax.set\_ylabel("Component 2")

ax.set\_title("Word Embeddings in 2D Space")

st.pyplot(fig)

# =================================================================================

# --- Question 3: Pointwise Mutual Information (PMI) Page ---

# =================================================================================

def page\_question\_3():

st.header("Question 3: Pointwise Mutual Information (PMI)")

st.markdown("---")

st.subheader("Data Source")

data\_source = st.radio(

"Choose a TF table:",

(

"Use default TF table from Q1",

"Upload CSV/Excel file",

"Paste CSV text"

)

)

tf\_df = None

if data\_source == "Upload CSV/Excel file":

uploaded = st.file\_uploader(

"Upload a TF table file (must include a 'term' column). Supported: .csv, .tsv, .txt, .xlsx",

type=["csv", "tsv", "txt", "xlsx"]

)

if uploaded is None:

st.info("Waiting for file upload...")

return

try:

name\_lower = uploaded.name.lower()

if name\_lower.endswith(".xlsx"):

df = pd.read\_excel(uploaded)

else:

df = pd.read\_csv(uploaded, sep=None, engine="python")

except Exception as e:

st.error(f"Error reading the uploaded file: {e}")

return

if 'term' not in df.columns:

st.error("Uploaded file must contain a 'term' column.")

return

tf\_df = df.set\_index('term')

st.success(f"Loaded file: {uploaded.name}")

with st.expander("Preview loaded data"):

st.dataframe(tf\_df.head())

elif data\_source == "Paste CSV text":

st.info("Please provide data in CSV format. The first column should be 'term'.")

user\_tf\_text = st.text\_area(

"Paste your CSV data here:",

height=150,

placeholder="term,DocA,DocB,DocC\nword1,10,4,0\nword2,0,5,12\nword3,1,1,1"

)

if user\_tf\_text:

try:

df = pd.read\_csv(StringIO(user\_tf\_text), sep=None, engine="python")

except Exception as e:

st.error(f"Error parsing your CSV data: {e}")

return

if 'term' not in df.columns:

st.error("CSV must have a 'term' column.")

return

tf\_df = df.set\_index('term')

st.success("Successfully loaded your TF table.")

else:

st.info("Waiting for user-provided CSV.")

return

else:

tf\_df = TF\_DEFAULT\_DF

st.write("Active Term-Frequency (TF) Table")

st.dataframe(tf\_df)

total\_corpus\_words = tf\_df.sum().sum()

if total\_corpus\_words == 0:

st.warning("The TF table is empty. Cannot calculate PMI/PPMI.")

return

use\_ppmi = st.checkbox("Use Positive PMI (PPMI)", value=True,

help="PPMI = max(0, PMI). Uncheck to see signed PMI values.")

with st.spinner("Computing PMI matrix..."):

t0 = time.perf\_counter()

pmi\_df = cache\_pmi\_matrix(tf\_df, use\_ppmi)

st.caption(f"PMI compute time: {(time.perf\_counter()-t0)\*1000:.1f} ms")

st.subheader("PMI/PPMI Matrix")

st.dataframe(pmi\_df.style.format("{:.2f}"))

st.write("Query Scoring using PMI/PPMI")

query\_pmi = st.text\_input("Enter a query to score (space-separated terms)", key="pmi\_query").lower()

query\_terms\_pmi = [t.strip() for t in query\_pmi.split() if t.strip() in pmi\_df.index]

if query\_pmi and not query\_terms\_pmi:

st.warning("None of the query terms are in the vocabulary.")

elif query\_terms\_pmi:

scores\_pmi = pmi\_df.loc[query\_terms\_pmi].sum(axis=0)

score\_df\_pmi = pd.DataFrame(scores\_pmi, columns=['Score']).sort\_values(by='Score', ascending=False)

st.dataframe(score\_df\_pmi.style.format("{:.2f}"))

st.success(f"Conclusion: Based on {'PPMI' if use\_ppmi else 'PMI'}, Document {score\_df\_pmi.index[0]} is the most relevant.")

# =================================================================================

# --- Main App Navigation ---

# =================================================================================

st.title("Lab 5: Sparse Vector (embedding) - Interactive Solution")

st.markdown("---")

st.sidebar.title("Lab Sections")

option = st.sidebar.radio(

"Choose a question to view:",

(

"Question 1: TF-IDF and Normalization",

"Question 2: Cosine Similarity",

"Question 3: Pointwise Mutual Information (PMI)"

)

)

if option == "Question 1: TF-IDF and Normalization":

page\_question\_1()

elif option == "Question 2: Cosine Similarity":

page\_question\_2()

elif option == "Question 3: Pointwise Mutual Information (PMI)":

page\_question\_3()

# Test Cases with Actual and Expected I/O

1) TF–IDF Query Scoring

- Input: query = 'car insurance' | Expected: Doc3 highest (because both terms are strong in Doc3).

- Actual scores: {'Doc3': 86.58, 'Doc2': 60.06, 'Doc1': 44.55}

- Actual winner: Doc3

- Input: query = 'best car' | Expected: Doc1 slightly higher than Doc3.

- Actual scores: {'Doc1': 65.55, 'Doc3': 65.1, 'Doc2': 6.6}

- Actual winner: Doc1

2) Cosine Similarity (TF–IDF, min\_df=2)

- Input: word = 'best', k=3 | Expected neighbors: a subset of {coverage, car, new}.

- Actual: [('car', 0.8165), ('coverage', 0.8165), ('new', 0.8165)]

- Input: word = 'car', k=3 | Expected neighbors: a subset of {coverage, best, new, auto, insurance}.

- Actual: [('coverage', 1.0), ('new', 1.0), ('best', 0.8165)]

3) PMI/PPMI Query Scoring on TF table

- Input: query = 'car insurance', metric = PMI | Expected: Doc1 ≥ Doc3 ≫ Doc2 (Doc2 has negative PMI for 'car').

- Actual (PMI): {'Doc1': 1.0376581260585247, 'Doc3': 0.4959453501612865, 'Doc2': -1.9026040263289887}

- Input: query = 'car insurance', metric = PPMI | Expected: Doc1 highest; Doc3 slightly > Doc2.

- Actual (PPMI): {'Doc1': 1.0376581260585247, 'Doc3': 0.4959453501612865, 'Doc2': 0.4844767480836245}