Lesson 38: BoW and TF-IDF

Data Science with Python – BSc Course

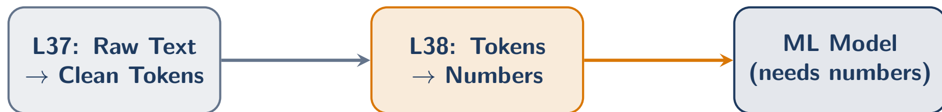
Data Science Program

BSc Course

45 Minutes

Previously on L37...

We learned to clean text. Now we need numbers.



- L37 output: [q3, `revenue`, `exceed`, `expectation`, `grow`]
- **Problem:** `LogisticRegression().fit(X, y)` needs a **number matrix** X
- Today: two classic methods to convert tokens → numbers

The gap between text and ML: tokens are strings, models need float arrays

Learning Objectives

The Problem: We have clean tokens, but ML models need numbers. How do we convert a collection of words into numerical feature vectors?

After this lesson, you will be able to:

1. **Explain** Bag of Words and why word order is discarded (Remember)
2. **Calculate** TF-IDF weights by hand for small examples (Apply)
3. **Compare** BoW vs TF-IDF strengths and weaknesses (Analyze)
4. **Build** a text classification pipeline with sklearn (Create)

Key Insight: Counting words is surprisingly powerful for many NLP tasks.

Finance Application: Converting news articles and filings into features for sentiment models

The One-Word Problem

How do you feed TEXT to a model?

Consider three earnings headlines:

1. "Revenue exceeded analyst expectations"
2. "Revenue missed analyst expectations"
3. "Dividend announced after strong quarter"

A model needs:

- Document 1: $[x_1, x_2, x_3, \ldots, x_n]$ – a vector of numbers
- Document 2: $[x_1, x_2, x_3, \ldots, x_n]$ – same length!
- Document 3: $[x_1, x_2, x_3, \ldots, x_n]$ – same length!

Simplest idea: Build a vocabulary of all unique words, then count each one.

That is **Bag of Words**.

Key constraint: every document must become a vector of the SAME length

Bag of Words: The Concept

Count how many times each word appears

- Ignore word order – just count occurrences (a “bag” of words)
- “The stock rose. The market rose.” → {the:2, stock:1, rose:2, market:1}

Bag of Words: Count Word Occurrences

`Document: "Apple stock rose. Stock is up."`

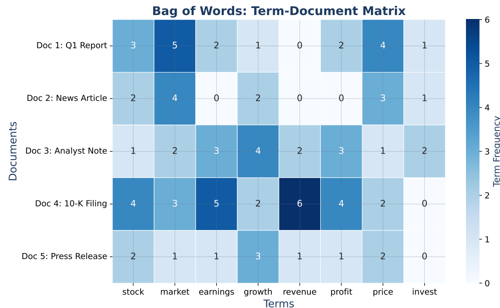
`Vector: [1, 2, 1, 1, 1]`

BoW ignores word order but captures word importance through frequency

The Document-Term Matrix

Rows = documents, Columns = vocabulary

- Each cell = how many times word j appears in document i
- Result: a matrix where every document has the same length

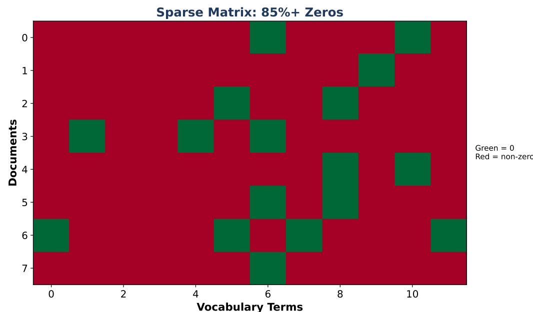


This is the matrix X that sklearn models consume directly

The Sparsity Problem

Most cells in the matrix are zero

- Vocabulary: 10,000+ unique words across all documents
- Each document uses only 50–200 words → 98%+ zeros
- Use `scipy.sparse` format to save memory



Sparse matrix: stores only non-zero entries – essential for large corpora

BoW Limitations: What Gets Lost

Three things BoW ignores:

- **Word order:** "Dog bites man" = "Man bites dog" (same counts!)
- **Semantics:** "good" and "excellent" are unrelated in BoW
- **Common words dominate:** "the" always has the highest count

Bag of Words Limitations

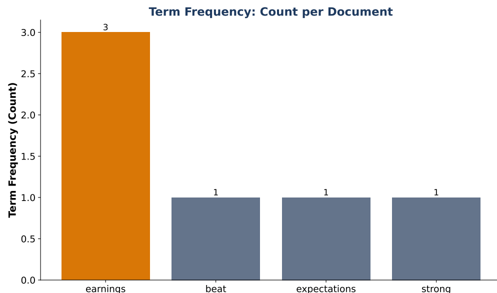
Word Order Lost	*"not good" = "good not"*
No Semantics	*"car" and "automobile" are different*
No Context	*"bank" (river) = "bank" (finance)*
Stopwords Dominate	*"the", "is", "and" are most frequent*

BoW limitation #3 is exactly what TF-IDF solves – coming next

Term Frequency (TF)

How often does a word appear in THIS document?

- $\text{TF}(t, d) = \frac{\text{count of term } t \text{ in doc } d}{\text{total terms in doc } d}$
- Normalizes for document length – long docs do not get unfair advantage

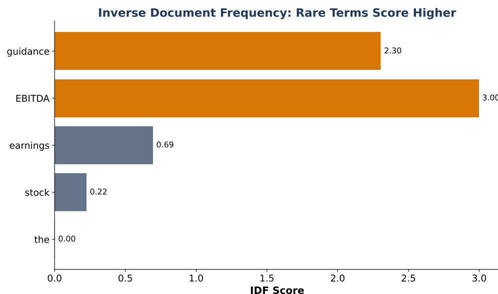


TF alone is just normalized BoW – the magic comes from multiplying by IDF

Inverse Document Frequency (IDF)

How rare is this word ACROSS all documents?

- $\text{IDF}(t) = \log\left(\frac{N}{\text{docs containing } t}\right)$
- "the" in every doc → IDF ≈ 0; "EBITDA" in 5 docs → IDF high

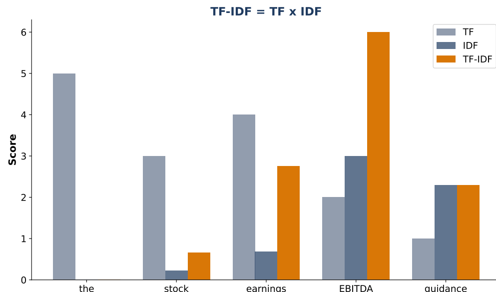


IDF penalizes common words and rewards distinctive terms automatically

TF-IDF = TF × IDF

Multiply frequency by distinctiveness

- High TF-IDF: frequent in this document AND rare overall
- Low TF-IDF: common everywhere ("the") or absent here



Analogy: every restaurant has tables (low TF-IDF), only one has a rooftop garden (high)

Checkpoint: Why Downweight Common Words?

Question: "the" appears 50 times. Why does TF-IDF score ≈ 0?

"the": TF $= 50/200 = 0.25$ (high) IDF $= \log(1000/1000) = 0$ TF-IDF $= \mathbf{0}$

"EBITDA": TF $= 3/200 = 0.015$ IDF $= \log(1000/5) = 5.3$ TF-IDF $= \mathbf{0.08}$

- "the" appears everywhere $\rightarrow$ IDF kills it
- "EBITDA" is rare $\rightarrow$ IDF amplifies it
- **Takeaway:** IDF zeros out noise, amplifies signal – automatically

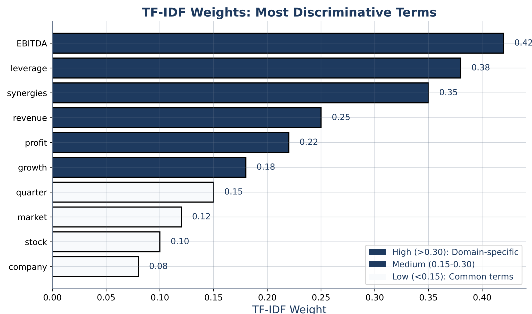
---

TF-IDF discovers informative words without any labels – unsupervised feature selection

TF-IDF Weights in Practice

Which words matter most?

- Domain-specific terms get highest weights automatically
- Common English words get near-zero weights



TF-IDF automatically discovers important vocabulary – no manual selection

sklearn: CountVectorizer & TfidfVectorizer

CountVectorizer (Bag of Words):

- `vec = CountVectorizer(max_features=5000, stop_words='english')`
- `X = vec.fit_transform(documents)`

TfidfVectorizer (TF-IDF – preferred):

- `vec = TfidfVectorizer(max_features=5000, ngram_range=(1,2))`
- `X = vec.fit_transform(documents)`

Key Parameters:

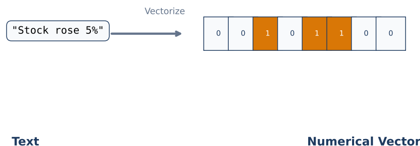
- `max_features`: Limit vocabulary; `ngram_range`: (1,2) = add bigrams
- `min_df`/`max_df`: Filter by document frequency

TfidfVectorizer = CountVectorizer + TF-IDF weighting in one step

Tuning Vectorizer Parameters

Small changes, big impact

- `max_features`: 1000 vs 5000 vs 50000 – vocabulary size trade-off
- `min_df=5`: removes words in <5 docs; `max_df=0.9`: removes words in >90% of docs



Start with defaults, then tune: max_features, min_df, max_df, ngram_range

N-grams: Capturing Phrases

Single words lose multi-word concepts

- “interest rate” – two words, one concept
- “not profitable” – negation changes meaning completely
- Bigrams preserve these relationships

N-grams: Word Sequences

Sentence: "interest rate hike"

Unigrams (n=1)

Bigrams (n=2)

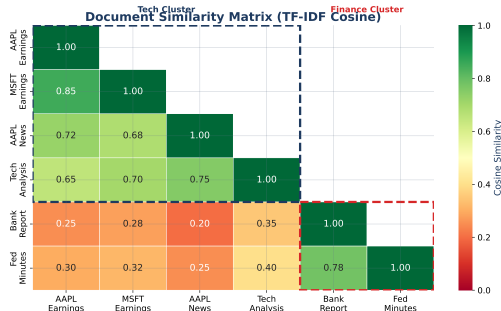
Trigrams (n=3)

Use ngram_range=(1,2) to include both unigrams and bigrams

BoW vs TF-IDF: When to Use Which

Head-to-head:

- **BoW:** Simple, interpretable, fast – good baseline
- **TF-IDF:** Finds distinctive words, usually more accurate

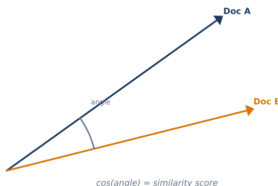


Default: TfidfVectorizer. Use CountVectorizer only for topic models (LDA).

Measuring Document Similarity

Cosine similarity measures the angle between document vectors

- Range: 0 (completely different) to 1 (identical content)
- Uses: document search, recommendation, plagiarism detection

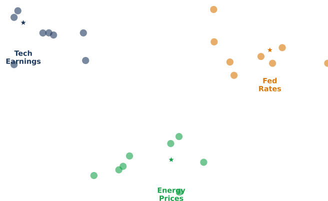


Cosine similarity works with both BoW and TF-IDF vectors

Finance: News Sentiment Features

From headlines to trading signals

- TF-IDF reveals sentiment words: "exceeded", "growth" (positive) vs "missed", "loss" (negative)
- These features become inputs to sentiment classifiers

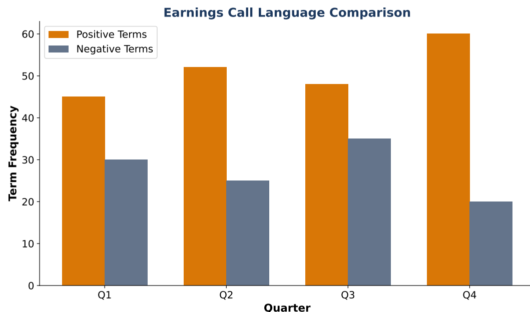


TF-IDF features are the foundation of most financial text analysis

Finance: Comparing Earnings Calls

Are two earnings calls saying the same thing?

- Convert transcripts to TF-IDF vectors
- Compute cosine similarity between quarters
- Track language changes: tone shift may signal risk



Language change across quarters can be an early warning signal

Complete Text Classification Pipeline

From Raw Text to Predictions – 5 Steps:

1. **Preprocess:** Clean text (L37 pipeline)
2. **Vectorize:** `TfidfVectorizer(max_features=5000, ngram_range=(1,2))`
3. **Split:** `train_test_split(X, y, test_size=0.2)`
4. **Train:** `LogisticRegression().fit(X_train, y_train)`
5. **Evaluate:** `classification_report(y_test, y_pred)`

Finance Example:

- Documents: 1000 earnings call transcripts
- Labels: "beat" or "miss" (earnings surprise)
- This pipeline achieves 70–80% accuracy as a baseline

TF-IDF + Logistic Regression: the strong baseline every NLP project should try first

Hands-On Exercise (25 min)

Task: Classify Financial News Sentiment

1. Load financial news headlines with positive/negative labels
2. Create TF-IDF features with `TfidfVectorizer`
3. Train Logistic Regression and Naive Bayes classifiers
4. Compare accuracy – which model performs better?
5. Inspect top features: which words predict positive vs negative?

Deliverable: Classification report + top 10 positive/negative features.

Extension: Try bigrams (ngram_range=(1,2)) – does accuracy improve?

Key Takeaways

Problem Solved: We can now convert text documents into numerical feature vectors for ML.

What You Learned:

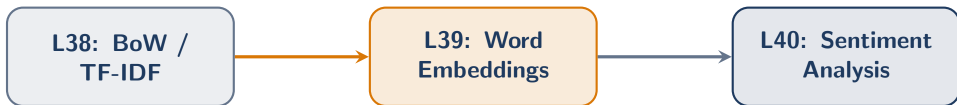
1. **Bag of Words:** count word occurrences (simple, fast, interpretable)
2. **TF-IDF:** weight by distinctiveness (high for rare, low for common)
3. **Document-term matrix:** rows = docs, columns = vocab, mostly zeros
4. **Cosine similarity:** measure how similar two documents are
5. **sklearn:** `TfidfVectorizer` does it all in one line

Default Choice: TF-IDF beats raw counts in nearly every scenario.

Memory: BoW = count words. TF-IDF = weight by distinctiveness. TfidfVectorizer in sklearn.

Next: Word Embeddings (L39)

TF-IDF limitation: "good" and "excellent" are unrelated vectors.



L39 Preview – What if words had meaning in vector space?

- Word2Vec: "king" − "man" + "woman" ≈ "queen"
- Dense vectors (300 dims) instead of sparse (10,000+ dims)
- Similar words → similar vectors (semantic similarity)

Next lesson: from counting words to understanding meaning