1 Introduction

Every model we have built so far in this course takes numbers as input — returns, prices, financial ratios, portfolio weights. But the vast majority of information produced in financial markets is not numerical at all. It is text: earnings call transcripts, 10-K filings, analyst reports, central bank minutes, financial news articles, Twitter and Reddit posts. Humans have always read these documents to form views about companies and markets. The question this lecture addresses is whether machines can do the same thing, systematically and at scale.

Three features make text data particularly valuable for finance. First, there is a **scale** problem. A single quarter produces thousands of earnings call transcripts across publicly traded companies. No human analyst can read them all in real time, but an algorithm can process the entire batch in minutes. Second, text often contains **forward-looking information** — management guidance, risk disclosures, shifts in tone — that is not captured in historical price or accounting data. An executive who suddenly starts using more hedging language ("approximately," "uncertain," "contingent") may be signalling trouble before the numbers reflect it. Third, there is a **speed** advantage. Algorithmic systems can process a news article in milliseconds, well before most human traders have finished reading the headline. The finance literature has confirmed that text-derived signals predict returns, volatility, and corporate events (Tetlock 2007, Loughran and McDonald 2011, Ke, Kelly, and Xiu 2019).

The core challenge is converting words into numbers — finding a numerical representation $\mathbf{x} \in \mathbb{R}^p$ for each document that preserves enough information for a downstream ML model to work with. The problem is that p is enormous. The Oxford English Dictionary lists roughly 170,000 English words in current use (OED 2nd Edition, 1989). After cleaning, the vocabulary for a financial corpus can be very large. Each document becomes a point in a space with tens of thousands of dimensions, and most coordinates are zero because any given article uses only a tiny fraction of the full vocabulary. This is exactly the kind of high-dimensional, sparse data that machine learning is designed for, and the tools we have studied throughout the course — regularization (Lecture 5), clustering (Lecture 4), classification (Lectures 7–8) — all become relevant here.

The pipeline for any text-based analysis follows four steps: (1) collect raw text (the *corpus*), (2) clean and preprocess it (lowercase, remove punctuation, stem or lemmatize, remove stop words), (3) represent it numerically (bag of words, TF-IDF, embeddings), and (4) model it (classification, regression, clustering, topic modeling). This lecture works through each stage of that pipeline.

1.1 Our Running Example

Throughout this chapter we work with a small corpus of Wall Street Journal articles from 2026, covering different financial topics: gas prices, European monetary policy, a food industry merger, prediction market regulation, and a bank settlement. Having articles about clearly different subjects makes it easy to see how the various text representations separate documents by topic. Here is the opening of one article:

> **Gas Just Hit $4 a Gallon. Is That Really as Bad as It Sounds?** — Wall Street Journal, March 31, 2026
>
> There is an argument that Americans shouldn't worry much about $4 gasoline. The people filling their tanks might not be convinced by it. The average price for a gallon of regular gasoline hit $4.02 on Tuesday, according to AAA. That was the first time it has been above $4 since August 2022, when the world was still struggling to absorb the oil shock from the Russia-Ukraine war.

Each article uses only a small fraction of the English vocabulary. This extreme sparsity — most words do not appear in any given document — is typical of text data.

We begin by loading the corpus into Python:

Python

```
import os
```

```python
# Read all articles from our corpus
corpus = {}
for filename in sorted(os.listdir("../Slides/10_text")):
    if filename.endswith(".txt"):
        with open(f"../Slides/10_text/{filename}") as f:
            corpus[filename] = f.read()

# Display article names and word counts
for name, text in corpus.items():
    words = text.split()
    print(f"{name:45s} {len(words):>4} words")
```

Output

```
bofasettlement_wsj.txt                        538 words
europemonetarypolicy_wsj.txt                  878 words
foodmerger_wsj.txt                            798 words
gas_wsj.txt                                   643 words
sportsbettingprediction_wsj.txt               646 words
```

The articles range from a few hundred to over a thousand words. Even in this tiny corpus, the vocabulary will contain hundreds of unique terms, most of which appear in only one or two articles.

2 From Text to Numbers

2.1 Bag of Words

The simplest way to represent a document numerically is to count how many times each word appears. Suppose our dictionary has V words. For each document d, we create a vector $\mathbf{x}_d \in \mathbb{R}^V$ where entry j is the count of word j in document d. This is called the **bag of words** (BOW) representation.

Consider the headline: **"Gas prices rise as energy prices rise"**

If our dictionary is {gas:1, prices:2, rise:3, as:4, energy:5}, then the bag-of-words vector is:

$$\mathbf{x} = (1, 2, 2, 1, 1)$$

"Prices" has count 2 because it appears twice, and so does "rise." The representation completely ignores the **order** of words — "prices rise" and "rise prices" produce the same vector. That is why it is called a "bag" of words: imagine dumping all the words into a bag and shaking them up. The order is lost, but the frequency information is preserved. This seems like a drastic simplification, and it is. But it turns out to be surprisingly effective for many tasks. The reason is that word frequencies alone carry a great deal of information about a document's topic and tone. An article that mentions "oil," "barrel," and "crude" many times is almost certainly about energy markets, regardless of the order in which those words appear.

One way to partially recover word order is to count **sequences** of consecutive words rather than individual words. These sequences are called n-grams:

- **Unigram** (1-gram): individual words — "gas", "prices", "rise"
- **Bigram** (2-gram): pairs — "gas prices", "prices rise", "rise as", "as energy", "energy prices", "prices rise"
- **Trigram** (3-gram): triples — "gas prices rise", "prices rise as", …

Bigrams distinguish "prices rise" from "rise prices." The phrase "prices rise" describes inflation, while "rise prices" means something different (to increase prices). Word order matters for meaning, and bigrams partially capture it. Formally, an n-gram model decomposes the probability of a word sequence as:

$$p(w_1, w_2, \ldots, w_m) = p(w_1) \times p(w_2|w_1) \times \cdots \times p(w_m|w_1, \ldots, w_{m-1})$$

The trade-off is that the dictionary size explodes. With V unique words, there are up to V^2 bigrams and V^3 trigrams. In practice, most text analyses use unigrams or unigrams combined with bigrams.

2.2 Preprocessing

Before building the term-document matrix, we clean the raw text. Each step reduces the vocabulary size and removes noise. **Lowercasing** ensures that "IBM" and "ibm" and "Ibm" are treated as the same token (with the occasional exception of preserving proper nouns like ticker symbols). **Removing punctuation and numbers** strips commas, periods, dollar signs, and standalone numbers like "13.5%" that typically add noise rather than signal. **Stop word removal** eliminates high-frequency words with little semantic content — "the," "a," "is," "and," "of," "to," "in," and so on. The NLTK English stop word list contains 179 such words, and removing them can cut 30–50% of all tokens in a document. Finally, **stemming or lemmatization** reduces words to their root form so that variants count together. Stemming is faster but cruder: "declining," "declined," and "declines" all become "declin." Lemmatization is slower but smarter: the same words become "decline." The choice between them depends on whether you prioritize speed or linguistic accuracy; for most financial text applications, either works.

After preprocessing, we arrange the word counts into a **term-document matrix** $\mathbf{X} \in \mathbb{R}^{D \times V}$, where D is the number of documents and V is the vocabulary size:

$$\mathbf{X} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1V} \\ x_{12} & x_{22} & \cdots & x_{2V} \\ \vdots & \vdots & \ddots & \vdots \\ x_{D1} & x_{D2} & \cdots & x_{DV} \end{pmatrix}$$

Each row is one document. Each column is one word. Entry x_{dj} is the count of word j in document d. This matrix is almost always very sparse: most entries are zero because any given document uses only a tiny fraction of the full vocabulary. For a corpus of 10,000 news articles with a large vocabulary, the matrix could have billions of entries, but 99%+ of them are zero. Sparse matrix representations in Python handle this efficiently by storing only the non-zero entries.

In Python, `sklearn.feature_extraction.text.CountVectorizer` builds this matrix directly from raw text, handling tokenization, stop word removal, and n-gram construction in one step. Let us build it for our corpus:

Python

```python
from sklearn.feature_extraction.text import CountVectorizer
import pandas as pd

# Build the term-document matrix (removing English stop words)
vectorizer = CountVectorizer(stop_words="english")
X = vectorizer.fit_transform(corpus.values())

# Short names for display
names = [n.replace("_wsj.txt", "").replace(".txt", "") for n in corpus.keys()]

print(f"Documents:        {X.shape[0]}")
print(f"Vocabulary size:  {X.shape[1]}")
print(f"Non-zero entries: {X.nnz} out of {X.shape[0] * X.shape[1]}")
print(f"Sparsity:         {1 - X.nnz / (X.shape[0] * X.shape[1]):.1%}")
```

Output

```
Documents:        5
Vocabulary size:  1090
Non-zero entries: 1275 out of 5450
Sparsity:         76.6%
```

The sparsity number confirms what we expected: the vast majority of entries in the term-document matrix are zero. Each article uses only a small fraction of the total vocabulary. This is a general feature of text data, not an artifact of our small corpus — it holds even with millions of documents.

We can examine the word counts for a few hand-picked terms that should differ across articles:

Python

```python
# Convert to a DataFrame
words = vectorizer.get_feature_names_out()
df_bow = pd.DataFrame(X.toarray(), index=names, columns=words)

# Pick some words that should differ across articles
sample_words = ["prices", "oil", "gas", "rate", "inflation",
                "bank", "merger", "betting", "lawsuit"]
sample_words = [w for w in sample_words if w in df_bow.columns]
df_bow[sample_words]
```

	prices	oil	gas	rate	inflation	bank	merger	betting	lawsuit
bofasettlement	0	0	0	0	0	14	0	0	11
europemonetarypolicy	14	4	3	12	9	6	0	0	0
foodmerger	3	0	0	0	0	0	2	0	0
gas	9	1	1	0	6	0	0	0	0
sportsbettingprediction	0	0	0	0	0	0	0	4	1

Each row is an article, each column is a word. Most entries are zero — a word like "oil" appears frequently in the gas article but not at all in the bank settlement article. This is exactly the kind of structure that makes text data amenable to machine learning: the pattern of which words appear and how often they appear encodes what the document is about.

2.3 TF-IDF: Weighting Words by Importance

Raw word counts treat all words equally. But some words are more informative than others. Consider two words that both appear 5 times in an earnings call transcript: "revenue" appears in nearly every earnings call — it has a high count, but it is not distinctive. "Restructuring," on the other hand, appears in only a few transcripts. It has a high count *and* it is distinctive. The word "restructuring" tells us much more about what makes this particular document different from others. We want a weighting scheme that rewards words that are frequent **in this document** but rare **across the corpus**.

TF-IDF (Term Frequency–Inverse Document Frequency) is the standard weighting scheme that addresses this problem. It has two components that work together.

Term Frequency (TF) measures how often word j appears in document d, normalized by the document's total word count:

$$\text{TF}(j, d) = \frac{\text{count of word } j \text{ in document } d}{\text{total words in document } d}$$

This normalization is important. Without it, a 1,000-word article with 5 occurrences of "revenue" and a 100-word article with 5 occurrences of "revenue" would look the same, even though "revenue" is much more

prominent in the shorter article. Dividing by total words puts documents on an equal footing regardless of length.

Inverse Document Frequency (IDF) measures how rare word j is across the entire corpus:

$$\text{IDF}(j) = \ln\left(\frac{D}{D_j}\right)$$

Here D is the total number of documents and D_j is the number of documents containing word j. The logarithm dampens the effect so that very rare words do not receive disproportionately extreme scores. When a word appears in every document, $D_j = D$, so IDF $= \ln(1) = 0$ — the word gets zero weight because it is completely uninformative about which document we are looking at. When a word appears in only one document, IDF $= \ln(D)$, the maximum possible value — this word is maximally distinctive.

The TF-IDF score is simply the product:

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

A word gets a high TF-IDF score when it is frequent in the current document *and* rare across the corpus. It gets a low score when it is common everywhere (like "the" or "company") or when it barely appears in the current document. Think of TF as measuring "how much does this document care about this word?" and IDF as measuring "how special is this word in general?" The product captures both dimensions simultaneously.

We can compute TF, IDF, and TF-IDF by hand for a few words in the gas article to see the mechanics:

Python

```python
import numpy as np

# Pick three words to compare in the gas article
target_words = ["prices", "gasoline", "gallon"]

# Get the gas article's row index
doc_idx = names.index("gas")
n_docs = X.shape[0]

for word in target_words:
    if word not in words:
        continue
    col = list(words).index(word)
    count = X[doc_idx, col]                          # raw count in this article
    total_words = X[doc_idx].sum()                   # total words in this article
    docs_with_word = (X[:, col].toarray() > 0).sum()  # docs containing this word
    tf = count / total_words
    idf = np.log(n_docs / docs_with_word)
    tfidf = tf * idf
    print(f'"{word:10s}"  count={count}  docs_with_word={docs_with_word}/{n_docs}'
          f'  TF={tf:.4f}  IDF={idf:.2f}  TF-IDF={tfidf:.4f}')
```

Output

```
"prices    "  count=9  docs_with_word=3/5  TF=0.0267  IDF=0.51  TF-IDF=0.0136
"gasoline  "  count=21  docs_with_word=1/5  TF=0.0623  IDF=1.61  TF-IDF=0.1003
"gallon    "  count=7  docs_with_word=1/5  TF=0.0208  IDF=1.61  TF-IDF=0.0334
```

Words that appear in many articles get a low IDF (they are not distinctive). Words concentrated in one article get a high IDF and therefore a high TF-IDF score. This is exactly the behaviour we want: the representation highlights what makes each document unique rather than what all documents share.

In practice, `sklearn.feature_extraction.text.TfidfVectorizer` builds the full TF-IDF matrix directly and is typically the default starting point for any text analysis:

Python
```python
from sklearn.feature_extraction.text import TfidfVectorizer

# Build the TF-IDF matrix
tfidf = TfidfVectorizer(stop_words="english")
X_tfidf = tfidf.fit_transform(corpus.values())

# Compare TF-IDF scores for the same words as before
words_tfidf = tfidf.get_feature_names_out()
df_tfidf = pd.DataFrame(X_tfidf.toarray(), index=names, columns=words_tfidf)
df_tfidf[sample_words].round(3)
```

	prices	oil	gas	rate	inflation	bank	merger	betting	lawsuit
bofasettlement	0.000	0.000	0.000	0.000	0.000	0.343	0.000	0.000	0.270
europemonetarypolicy	0.263	0.091	0.068	0.337	0.204	0.136	0.000	0.000	0.000
foodmerger	0.059	0.000	0.000	0.000	0.000	0.000	0.059	0.000	0.000
gas	0.192	0.026	0.026	0.000	0.154	0.000	0.000	0.000	0.000
sportsbettingprediction	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.121	0.025

Compare this to the raw counts above. Words that are common across many articles (like "prices") get lower TF-IDF scores, while words unique to one article (like "betting" or "lawsuit") get higher scores. TF-IDF is a simple but remarkably effective transformation that forms the backbone of most text-based models in finance.

3 Sentiment Analysis

3.1 Dictionary-Based Approaches

Sentiment analysis tries to measure the **tone** of a document — is it positive, negative, or neutral? In finance, this is directly useful. Does an earnings call transcript sound optimistic or pessimistic? Is a news article about a company positive or negative? Is the tone of Fed minutes hawkish or dovish? If we can reliably quantify tone, we can use it as a feature in trading models, risk assessment, or event studies.

The simplest approach uses pre-built **sentiment dictionaries**: lists of words pre-classified as positive or negative. Count the positive and negative words in a document, and compute a score. The most widely used finance-specific sentiment dictionary was developed by Loughran and McDonald (2011). Their contribution was to point out that general-purpose sentiment dictionaries (like the Harvard General Inquirer) misclassify many financial terms. The word "liability," for instance, is negative in everyday English but routine in financial text — it simply describes one side of a balance sheet. The word "outstanding" is positive in everyday English but means "currently owed" in financial text ("outstanding shares," "outstanding debt"). Applying a general dictionary to 10-K filings introduces systematic noise because so many common financial terms are falsely flagged as negative.

The Loughran-McDonald dictionary contains six word lists tailored to 10-K filings:

Category	Example words	Count
Negative (Fin-Neg)	loss, impairment, decline, litigation, default	2,337
Positive (Fin-Pos)	achieve, attain, improve, profitable, upturn	353
Uncertainty (Fin-Unc)	approximate, contingency, depend, fluctuate, uncertain	285

Category	Example words	Count
Litigious (Fin-Lit)	claimant, deposition, interlocutory, testimony, tort	731
Strong Modal (MW-Strong)	always, highest, must, will	19
Weak Modal (MW-Weak)	could, depending, might, possibly	27

The asymmetry in size is striking: the negative list contains nearly seven times as many words as the positive list. This reflects the fact that negative language in financial disclosures is far more varied than positive language. Companies have many ways to describe problems — impairment, restructuring, litigation, default, termination — but relatively few ways to describe good news.

Given a dictionary, the simplest sentiment score for document d is:

$$\text{Sentiment}_d = \frac{N_d^+ - N_d^-}{N_d^+ + N_d^-}$$

where N_d^+ is the count of positive words and N_d^- is the count of negative words. This score ranges from -1 (entirely negative) to $+1$ (entirely positive). When $N_d^+ = N_d^-$, the score is zero — the document is balanced. When all sentiment-charged words are negative ($N_d^+ = 0$), the score is -1. The denominator normalizes by the total number of sentiment words, so the score reflects the *proportion* of positive versus negative language rather than the raw count.

An alternative normalization divides by the total word count in the document:

$$\text{Sentiment}_d = \frac{N_d^+ - N_d^-}{\text{Total words in } d}$$

This adjusts for document length and allows comparison across short news articles and long 10-K filings. A 200-word article with 10 negative words is more intensely negative than a 10,000-word filing with 10 negative words, and this normalization captures that difference.

We can apply a simple dictionary approach to our corpus. The word lists below are a small sample inspired by Loughran and McDonald (2011):

Python

```python
# Small finance-oriented sentiment word lists
negative = {"loss", "decline", "fell", "fall", "risk", "lawsuit", "penalty",
            "recession", "inflation", "debt", "default", "failure", "hurt",
            "erode", "drop", "sank", "slipped", "charges", "convicted",
            "tumbled", "worried", "weary", "disrupted", "ban", "illegal"}
positive = {"growth", "gain", "profit", "improve", "success", "benefit",
            "rose", "growing", "boost", "excelled", "opportunity", "strong"}

# Score each article
for name, text in corpus.items():
    tokens = text.lower().split()
    neg = sum(1 for w in tokens if w in negative)
    pos = sum(1 for w in tokens if w in positive)
    total = pos + neg
    if total > 0:
        score = (pos - neg) / total
    else:
        score = 0.0
    label = name.replace("_wsj.txt", "").replace(".txt", "")
```



```
    print(f"{label:35s}  pos={pos:2d}  neg={neg:2d}  score={score:+.2f}")
```

Output

```
bofasettlement                      pos= 0  neg=12  score=-1.00
europemonetarypolicy                pos= 1  neg= 9  score=-0.80
foodmerger                          pos= 6  neg= 5  score=+0.09
gas                                 pos= 1  neg= 4  score=-0.60
sportsbettingprediction             pos= 1  neg= 4  score=-0.60
```

The scores give a rough sense of each article's tone. Articles about settlements, lawsuits, and economic anxiety tend to score negatively, while articles with growth or profit language score more positively. The approach is crude — we are using a tiny dictionary and ignoring context entirely — but even this simple method captures meaningful variation.

3.2 Limitations and Data-Driven Alternatives

Dictionary-based sentiment is simple and interpretable, but it has clear weaknesses. **Negation** is the most obvious: "Revenue did *not* decline" contains the negative word "decline" but the sentence is actually positive. A simple word count misses this entirely. **Context** is another problem: "The risk of loss is *minimal*" contains the negative words "risk" and "loss," but the sentence is reassuring. **Sarcasm and nuance** pose difficulties as well — "What a *great* quarter for shareholders" might be sarcastic in context. **Domain specificity** means that even finance-specific dictionaries cannot cover every context; "volatility" is negative for a risk manager but could be positive for an options trader. And **fixed vocabularies** do not adapt to new language — financial jargon evolves, and terms like "meme stock" or "SPAC" would not appear in a dictionary built from pre-2010 filings.

These limitations motivate more sophisticated approaches that learn sentiment from data rather than specifying it in advance. Ke, Kelly, and Xiu (2019) developed **SESTM** — Sentiment Extraction via Screening and Topic Modeling — which learns which words carry sentiment from the data itself.

The approach has two stages. In **Stage 1 (Screening)**, of the many thousands of words in the vocabulary, most are irrelevant to sentiment. SESTM screens for words whose frequency correlates with subsequent stock returns, reducing the vocabulary from V to a much smaller set V_S of "sentiment-charged" words. In **Stage 2 (Topic model)**, the screened word counts are modelled as a mixture of two topics: a positive topic O_+ and a negative topic O_-. Each document gets a sentiment score $p_i \in [0, 1]$ based on how much it loads on the positive versus negative topic.

Recall that a Multinomial distribution models the outcome of drawing n items from a set of categories, each with a fixed probability — like rolling a weighted die n times and counting how often each face appears. Here, the "die" has one face per sentiment-charged word, and the probabilities come from a blend of the positive and negative topics:

$$d_{iS} \sim \text{Multinomial}(s_i,\ p_i O_+ + (1 - p_i) O_-)$$

Each symbol deserves attention. d_{iS} is the vector of sentiment-charged word counts in article i. s_i is the total number of sentiment-charged words in article i — the number of "draws" from the Multinomial. O_+ and O_- are probability vectors over the vocabulary — the positive and negative topics. And $p_i \in [0, 1]$ is the sentiment score: p_i close to 1 means the article's words are drawn mostly from the positive topic, p_i close to 0 means mostly negative.

The model estimates three things: the positive topic O_+ (which words are associated with good news), the negative topic O_- (which words are associated with bad news), and a sentiment score p_i for every article. The topics O_+ and O_- are estimated once from the entire corpus. The p_i are estimated separately for each article — they are the output of interest, giving each article a data-driven sentiment score without ever specifying a dictionary.

Applied to Dow Jones newswires (2004–2017), SESTM produces a daily long-short portfolio with a Sharpe ratio of 4.29 (equal-weighted) — substantially higher than dictionary-based approaches. The improvement comes from letting the data decide which words matter, rather than relying on a human-curated list that may miss subtle or domain-specific language.

4 Topic Modeling with LDA

Sentiment analysis reduces a document to a single number (positive or negative). But documents contain richer information. An earnings call might discuss revenue growth, cost cutting, regulatory risk, and new product launches — all in the same transcript. **Topic modeling** is an unsupervised method that discovers the latent themes (topics) in a collection of documents. It answers "What is this corpus about?" without any labels.

Recall from Lecture 4 that unsupervised learning discovers structure in data without labels. Topic modeling is the text equivalent of clustering: instead of grouping data points into clusters, we discover groups of words that tend to co-occur. The most widely used topic model is **Latent Dirichlet Allocation** (LDA), introduced by Blei, Ng, and Jordan (2003). Despite the shared acronym, this LDA is unrelated to the Linear Discriminant Analysis from Lecture 7.

4.1 The Dirichlet Distribution

Before we can describe LDA, we need to understand the **Dirichlet distribution** — a distribution that generates probability vectors (vectors of non-negative numbers that sum to 1).

Suppose we have $K = 3$ topics. A probability vector $\theta = (\theta_1, \theta_2, \theta_3)$ with $\theta_1 + \theta_2 + \theta_3 = 1$ describes how much of each topic a document contains. The Dirichlet distribution is a way to randomly generate such vectors. You can think of it as a machine that, every time you press a button, spits out a different probability vector — a different blend of topics. What makes the Dirichlet useful is that its single parameter α controls the *character* of the vectors it produces.

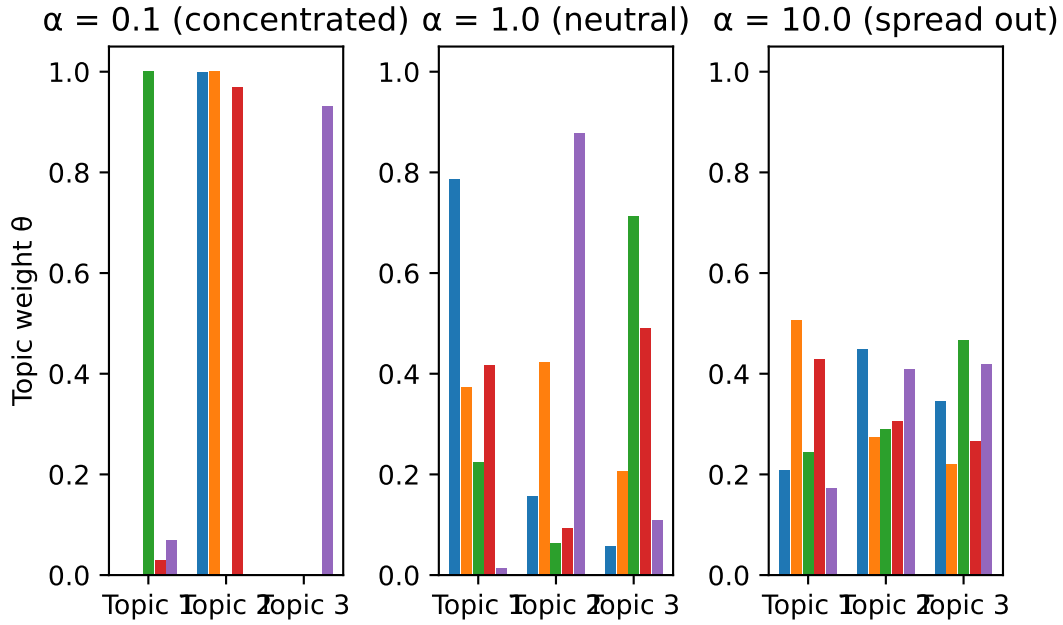
When all α_k are equal to some value α, this single number controls the **concentration** of the resulting vectors. Small α (e.g., 0.1) means most of the probability mass lands on one or two topics — documents are "about" one thing. When $\alpha = 1$, all probability vectors are equally likely — a uniform prior over mixtures. Large α (e.g., 10) means the probability is spread roughly evenly across all topics — documents are a balanced blend of everything. The analogy to portfolio allocation is useful here: small α is like a concentrated portfolio (all your money in one or two stocks), while large α is like a diversified portfolio (equal weights across many stocks).

We can visualize this by sampling topic mixtures from the Dirichlet with different α values:

Python

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)

# Sample topic mixtures from Dirichlet with different alpha values
fig, axes = plt.subplots(1, 3)
topics = ["Topic 1", "Topic 2", "Topic 3"]

for ax, alpha, title in zip(axes,
    [0.1, 1.0, 10.0],
    [" = 0.1 (concentrated)", " = 1.0 (neutral)", " = 10.0 (spread out)"]):
    # Sample 5 documents' topic mixtures
    samples = np.random.dirichlet([alpha, alpha, alpha], size=5)
```

```
    x = np.arange(3)
    for i, theta in enumerate(samples):
        ax.bar(x + i * 0.15, theta, width=0.14)
    ax.set_xticks(x + 0.3)
    ax.set_xticklabels(topics)
    ax.set_ylim(0, 1.05)
    ax.set_title(title)

axes[0].set_ylabel("Topic weight  ")
plt.tight_layout()
plt.show()
```



Each group of bars is one sampled document's topic mixture. With small α, each document is dominated by a single topic — the bars are all-or-nothing. With large α, every document is a roughly equal blend — the bars are nearly the same height. In practice, real text corpora tend to have small α — most documents focus on a few topics, not all of them equally. A news article about an oil price shock is mostly about energy, with perhaps a small amount of monetary policy mixed in.

4.2 The LDA Generative Story

LDA is a **generative model** — it tells a fictional story about how documents are created, then works backward to infer the hidden structure from the words we actually observe. Nobody believes documents are actually written this way, but the model captures the essential idea that documents are mixtures of topics and topics are clusters of related words.

Suppose we have K topics and a vocabulary of V words. Each topic k has its own word distribution ϕ_k (phi) — a vector of V probabilities that sum to 1. A "monetary policy" topic would put high probability on words like "rate," "inflation," "fed," and "monetary," and low probability on everything else. An "energy" topic would put high probability on "oil," "crude," "barrel," and "pipeline."

To "generate" document d, LDA imagines three steps.

Step 1 — Pick a topic mixture for this document:

$$\theta_d \sim \text{Dirichlet}(\alpha)$$

The vector θ_d (theta) has K entries that sum to 1. It describes this particular document's blend of topics. For instance, $\theta_d = (0.7, 0.2, 0.1)$ would mean the document is 70% topic 1, 20% topic 2, 10% topic 3. Different documents get different θ_d values, all drawn from the same Dirichlet distribution — so they share the same set of topics but in different proportions.

Step 2 — For each word position n in the document, pick a topic:

$$z_{dn} \sim \text{Multinomial}(\theta_d)$$

This is like rolling a weighted die with K faces. If $\theta_d = (0.7, 0.2, 0.1)$, then word n has a 70% chance of being assigned to topic 1, 20% to topic 2, and 10% to topic 3. Each word gets its own topic assignment — different words in the same document can come from different topics. This is what makes LDA more flexible than simple clustering: a document is not assigned to a single topic, but to a mixture.

Step 3 — Given the topic, pick a word:

$$w_{dn} \sim \text{Multinomial}(\phi_{z_{dn}})$$

Once we know word n was assigned to topic k, we draw the actual word from that topic's word distribution ϕ_k. If topic k is "monetary policy," we are likely to draw "rate" or "inflation" and unlikely to draw "merger" or "batting."

In LDA, we observe only the words in each document. Everything else is **latent** (hidden) and must be inferred: the topic assignments for each word z_{dn}, the topic mixture for each document θ_d, and the word distribution for each topic ϕ_k.

The joint probability of everything — words $\mathbf{w}$, topic assignments $\mathbf{z}$, document mixtures θ, and topic-word distributions ϕ — is:

$$p(\mathbf{w}, \mathbf{z}, \theta, \phi) = \prod_{k=1}^{K} p(\phi_k) \prod_{d=1}^{D} p(\theta_d) \prod_{n=1}^{N_d} p(z_{dn}|\theta_d)\, p(w_{dn}|z_{dn}, \phi)$$

Reading this from left to right: we first account for the prior probability of each topic's word distribution $p(\phi_k)$, then for each document d, the prior probability of its topic mixture $p(\theta_d)$, and then for each word position n in that document, the probability of the topic assignment given the mixture $p(z_{dn}|\theta_d)$ times the probability of the observed word given the topic $p(w_{dn}|z_{dn}, \phi)$.

The inference task is to compute the posterior $p(\mathbf{z}, \theta, \phi \mid \mathbf{w})$ — the most likely topic structure given the words we actually observed. This posterior is intractable to compute exactly, so LDA uses approximate inference algorithms — either Gibbs sampling (a Markov chain Monte Carlo method) or variational inference (an optimization-based approximation). The details of these algorithms are beyond our scope, but the key point is that they work backward from observed words to infer the hidden topic structure.

4.3 LDA in Practice

Bybee, Kelly, Manela, and Xiu (2023) applied LDA to 800,000 Wall Street Journal articles from 1984–2017, fitting a 180-topic model. The algorithm discovered topics that align with recognizable economic themes — without being told what to look for. The topic labels below are assigned by humans after inspecting the top words; LDA itself only outputs word distributions.

Discovered topic	Top words (from Table 7 of the paper)
Federal Reserve	greenspan, yellen, fomc, federalfunds, bernanke, rate hike, inflation
Profits	net profit, profit rose, profit fell, pretax, profit surge, profit decline
Oil market	opec, brent crude, oil demand, crude price, petroleum, oil exporter, oil supply
Real estate	reit, estate investment, commercial property, realty, estate developer
Elections	romney, mccain, obama, democratic primary, republican primary, gop, caucuses

Each article gets a topic mixture θ_d describing its composition. An article about the Fed raising rates amid an oil price shock would load heavily on the "Federal Reserve" and "Oil market" topics. By tracking how much attention the WSJ devotes to each topic over time, the authors showed that shifts in news attention predict macroeconomic variables like industrial production growth.

Like K in K-means clustering (Lecture 4), the number of topics K is a hyperparameter that must be chosen. There is no single best method. **Perplexity** measures how well the model predicts held-out documents — lower perplexity means better fit. Compute it on a validation set for different values of K and look for an elbow, similar to the elbow method in K-means. **Coherence score** measures whether the top words in each topic tend to co-occur in the same documents — higher coherence means more interpretable topics. And **human judgment** remains essential: run the model for several values of K, inspect the top 10 words per topic, and ask whether the topics are interpretable and useful for your application. For a broad survey of financial news, $K = 50$–200 might be appropriate. For analyzing earnings calls of a single company, $K = 5$–15 might suffice.

Our corpus is small (only a handful of articles), so the results will not be as clean as with thousands of documents. But we can still see LDA in action:

Python

```python
from sklearn.decomposition import LatentDirichletAllocation

# Fit LDA with 5 topics on the BOW matrix
lda = LatentDirichletAllocation(n_components=5, random_state=42)
lda.fit(X)

# Show top 8 words per topic
feature_names = vectorizer.get_feature_names_out()
for i, topic in enumerate(lda.components_):
    top_words = [feature_names[j] for j in topic.argsort()[-8:]]
    print(f"Topic {i+1}: {', '.join(top_words)}")
```

Output

```
Topic 1: focused, cut, way, resulted, days, rose, recent, work
Topic 2: gallon, prices, people, america, lawsuit, bank, epstein, gasoline
Topic 3: deal, increase, pointing, triangle, unilever, brands, food, mccormick
Topic 4: likely, said, rise, workers, inflation, rate, energy, prices
Topic 5: states, contracts, markets, including, prediction, polymarket, kalshi, sports
```

With a larger corpus, the topics would be sharper. Bybee et al. (2023) used 800,000 articles and 180 topics — the scale makes all the difference. With only a handful of documents, LDA does not have enough data to cleanly separate themes, but even here you may see some recognizable groupings.

5 Measuring Similarity and Word Embeddings

5.1 Cosine Similarity

Before we move to embeddings, we need a way to measure how similar two vectors are. All of the representations we have built so far — BOW vectors, TF-IDF vectors — live in high-dimensional space. To compare two documents, we need a notion of distance or similarity between their vectors.

Recall from Lecture 1 that the **dot product** of two vectors $\mathbf{a}$ and $\mathbf{b}$ is:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{p} a_i b_i$$

The dot product is large when two vectors point in the same direction and zero when they are perpendicular. It is related to the angle θ between the vectors by:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|_2 \, \|\mathbf{b}\|_2 \cos\theta$$

where $\|\mathbf{a}\|_2 = \sqrt{a_1^2 + a_2^2 + \cdots + a_p^2}$ is the Euclidean norm — the length of the vector. Rearranging this equation gives us **cosine similarity**, a measure of similarity that depends only on the *angle* between two vectors, not their lengths:

$$\text{sim}(\mathbf{a}, \mathbf{b}) = \cos\theta = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|_2 \, \|\mathbf{b}\|_2}$$

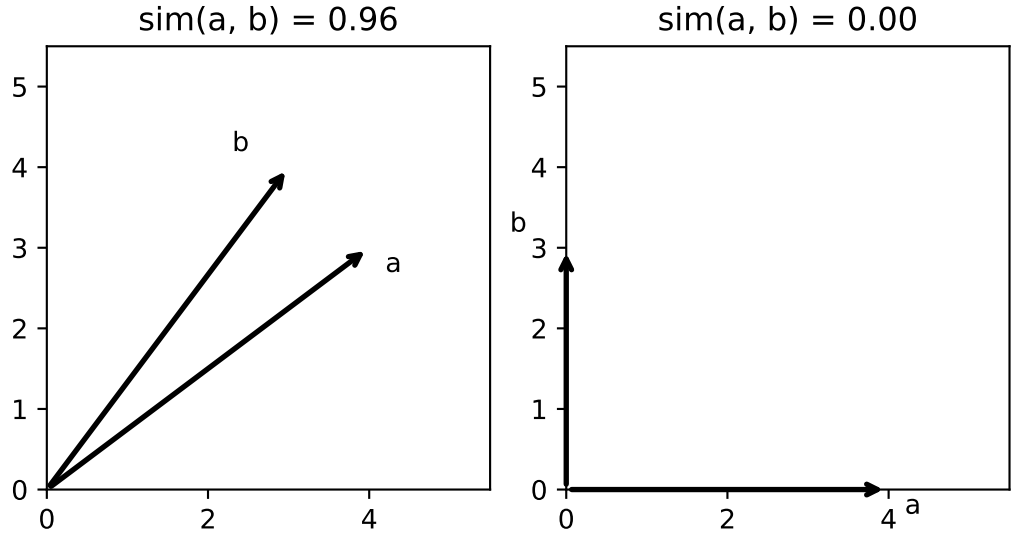
This ranges from -1 (opposite directions) through 0 (perpendicular, no similarity) to $+1$ (same direction, identical pattern). Normalizing by length is particularly useful for text: a long article and a short article about the same topic will have high cosine similarity even though their raw word counts differ substantially. What matters is the *pattern* of word usage — which words are present and in what proportions — not the absolute magnitudes.

When $\text{sim}(\mathbf{a}, \mathbf{b}) = 0$, the vectors are **orthogonal** — they share nothing in common. In a BOW representation, two documents are orthogonal if they share no words at all.

Python

```
import numpy as np
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 2)

# Left panel: two vectors pointing in similar directions
a1 = np.array([4, 3])
b1 = np.array([3, 4])
sim1 = np.dot(a1, b1) / (np.linalg.norm(a1) * np.linalg.norm(b1))

ax = axes[0]
ax.annotate("", xy=a1, xytext=(0, 0), arrowprops=dict(arrowstyle="->", lw=2))
ax.annotate("", xy=b1, xytext=(0, 0), arrowprops=dict(arrowstyle="->", lw=2))
ax.text(a1[0] + 0.2, a1[1] - 0.3, "a")
ax.text(b1[0] - 0.7, b1[1] + 0.2, "b")
ax.set_title(f"sim(a, b) = {sim1:.2f}")
ax.set_xlim(0, 5.5)
ax.set_ylim(0, 5.5)
ax.set_aspect("equal")
```

```
# Right panel: two perpendicular vectors
a2 = np.array([4, 0])
b2 = np.array([0, 3])
sim2 = np.dot(a2, b2) / (np.linalg.norm(a2) * np.linalg.norm(b2))

ax = axes[1]
ax.annotate("", xy=a2, xytext=(0, 0), arrowprops=dict(arrowstyle="->", lw=2))
ax.annotate("", xy=b2, xytext=(0, 0), arrowprops=dict(arrowstyle="->", lw=2))
ax.text(a2[0] + 0.2, a2[1] - 0.3, "a")
ax.text(b2[0] - 0.7, b2[1] + 0.2, "b")
ax.set_title(f"sim(a, b) = {sim2:.2f}")
ax.set_xlim(0, 5.5)
ax.set_ylim(0, 5.5)
ax.set_aspect("equal")

plt.tight_layout()
plt.show()
```



The left panel shows two vectors pointing in nearly the same direction — high cosine similarity. The right panel shows two perpendicular vectors — zero cosine similarity, meaning they share nothing in common. In BOW space, two documents that use completely different words are orthogonal like this, even if they discuss the same topic using synonyms. This is the core limitation that motivates word embeddings.

5.2 From Sparse to Dense: Word Embeddings

Every approach so far — BOW, TF-IDF, sentiment dictionaries, LDA — treats words as discrete symbols. Word j is a one-hot vector: all zeros except for a 1 in position j. This means "revenue" and "sales" are just as different as "revenue" and "giraffe" — both pairs are orthogonal in the word-count space. Synonyms get no credit for meaning the same thing, and the representation has no notion of semantic similarity. We want a representation where words with similar meanings are close together in vector space.

Mikolov et al. (2013) introduced **Word2Vec**, which learns a dense vector representation (an "embedding") for each word from a large text corpus. The core idea is captured by a famous linguistic principle:

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

Words that appear in similar contexts should have similar embeddings. "Revenue" and "sales" often appear

near the same words ("quarterly," "growth," "declined"), so their embeddings should be close. "Revenue" and "giraffe" almost never appear in similar contexts, so their embeddings should be far apart. Word2Vec uses a shallow neural network trained on a simple task: given a word, predict its surrounding context words (or vice versa). The learned weights of this network become the word embeddings.

Each word gets mapped to a dense vector, typically of dimension 100–300:

$$\text{"revenue"} \to [0.21, -0.08, 0.44, \ldots, -0.12] \in \mathbb{R}^{300}$$

Compare this to the one-hot BOW vector for "revenue," which is a sparse vector in $\mathbb{R}^{50{,}000}$ with a single 1. The embedding is dramatically lower-dimensional and captures meaning. Two words with similar meanings will have high cosine similarity in embedding space, even though they are orthogonal in BOW space.

Word embeddings encode semantic relationships as geometric relationships in vector space. The most famous example is:

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

The vector from "man" to "king" (the "royalty" direction) is approximately the same as the vector from "woman" to "queen." In finance, embeddings trained on financial text capture finance-specific relationships: "bearish" and "pessimistic" have similar embeddings, "revenue" is close to "sales," "income," and "turnover," and $\vec{\text{CEO}} - \vec{\text{company}} + \vec{\text{country}} \approx \vec{\text{president}}$.

To represent an entire document using embeddings, a simple approach is to average the embeddings of all words in the document:

$$\mathbf{x}_d = \frac{1}{N_d} \sum_{n=1}^{N_d} \mathbf{e}_{w_{dn}}$$

where $\mathbf{e}_{w_{dn}}$ is the embedding of the n-th word in document d. This gives a dense, fixed-length vector for each document that can be fed into any downstream model (regression, classification, etc.). It is a crude approach — averaging throws away word order and weighting — but it is often surprisingly competitive as a baseline.

Word2Vec produces **static** embeddings: each word gets one vector, regardless of context. But many words have multiple meanings. "The **bank** raised interest rates" (financial institution), "We walked along the river **bank**" (edge of a river), and "I want to **bank** on this trade" (rely on) all use the same word in different senses. In Word2Vec, all three uses of "bank" map to the same vector, which is a problem.

Contextual embeddings (ELMo, BERT, GPT) solve this by producing a different vector for each *occurrence* of a word, depending on the surrounding context. The vector for "bank" in "bank raised rates" is different from "bank" in "river bank." This is achieved by processing the entire sentence through a deep neural network (like the ones we saw in Lecture 9), so each word's representation incorporates information from all the other words around it.

6 Large Language Models

The progression of text representations mirrors the progression of ML models in this course — each step trades simplicity for expressiveness:

Representation	Dimension	Captures
Bag of words	V (~50,000)	Word presence
TF-IDF	V (~50,000)	Word importance
Word2Vec	~300	Semantic similarity

Representation	Dimension	Captures
BERT / GPT	~768–1,024 per token	Context-dependent meaning

Large language models (LLMs) like BERT, GPT, and LLaMA take the contextual embedding idea to its extreme. They are massive neural networks — from 110 million parameters (BERT-base) to hundreds of billions (GPT-4) — pre-trained on enormous text corpora. The key architectural innovation is the **attention mechanism** (Vaswani et al. 2017), which allows the model to weigh the relevance of every other word in a sentence when computing the representation of any given word. We will not cover the attention mechanism in detail, but the intuition matters: instead of processing text left-to-right, the model can "look at" the entire input simultaneously to resolve ambiguity.

Consider two sentences: "Swing the **bat**!" and "The **bat** flew at night." Traditional embeddings give "bat" the same vector in both sentences. The attention mechanism addresses this by computing context-dependent weights. For each word, the model asks: "Which other words in this sentence are most relevant to understanding me?" In "Swing the bat," high attention weight on "swing" pushes the representation toward the sports meaning. In "The bat flew at night," high attention weight on "flew" and "night" pushes toward the animal meaning. The result: "bat" gets a different vector in each sentence, one close to "baseball" and the other close to "animal."

In practice, finance researchers use pre-trained LLMs in two ways. The first is **feature extraction**: pass a news article through a pre-trained model (e.g., BERT, RoBERTa) and extract the output embeddings. These dense vectors become features for a downstream model — logistic regression, Random Forest, neural network — that predicts returns, default, or sentiment. Chen, Kelly, and Xiu (2024) use this approach to predict stock returns from news. They extract embeddings from several models and use them to construct long-short portfolios. Their best model (using RoBERTa embeddings) achieves an equal-weighted Sharpe ratio of 3.82, comparable to the SESTM approach but without requiring any feature engineering.

The second approach is **zero-shot or few-shot classification**: ask the LLM directly, "Is this article positive or negative for the stock price?" Modern LLMs can perform sentiment classification without any training data, using only the prompt. This bypasses the entire pipeline of tokenization → TF-IDF → model fitting.

The trade-off is real: LLMs are computationally expensive, require significant hardware, and can behave unpredictably. Simpler methods like TF-IDF + logistic regression remain competitive for many tasks and are far easier to deploy and interpret. For a class project or first pass at a research question, TF-IDF + a simple model (logistic regression, Lasso) is almost always the right starting point. Add complexity only when the simple approach fails.

The full NLP toolkit we have covered, from simplest to most complex:

Method	Type	What it does	Strengths	Weaknesses
BOW / TF-IDF	Representation	Count (or weight) words	Simple, interpretable, fast	Ignores word order and meaning
Sentiment dictionaries	Supervised (manual)	Score tone using word lists	Transparent, domain-specific	Rigid, misses context
SESTM	Supervised (learned)	Learn sentiment words from returns	Data-driven, high Sharpe	Requires labelled outcomes
LDA	Unsupervised	Discover latent topics	No labels needed, flexible	Hard to tune K, slow
Word2Vec	Representation	Dense vectors from context	Captures meaning	Static, one vector per word
BERT / LLMs	Representation + model	Context-dependent embeddings	State of the art	Expensive, black box

