Classical Text Classification
Multimodal Computational Methods in Political Science

Tamara Grechanaya [1]

[1]LMU Munich — Computational Social Science MA Program

April, 2026

Course Roadmap

1. Text as Data: Foundations & Preprocessing
2. **Classical Text Classification** ← *Today*
3. Word Embeddings & Vector Spaces
4. Document Representations & Topic Models
5. Neural Networks & Sequence Models
6. Attention & the Transformer Architecture
7. Transfer Learning & Fine-Tuning BERT

Last week we learned to *represent* text as numbers. Today we learn to *classify* text into categories — the most common supervised task in political text analysis.

Today's Outline

Part I: The Supervised Learning Framework

- What is supervised learning?
- Training, validation, and test sets
- From text to features (recap)

Part II: Logistic Regression

- Intuition and the sigmoid function
- From features to predictions
- Training: cost function & gradient descent
- Worked political example

Part III: Naive Bayes

- Bayes' theorem refresher
- The "naive" assumption
- Laplacian smoothing
- Log-likelihood for stable computation

Part IV: Evaluation & Practice

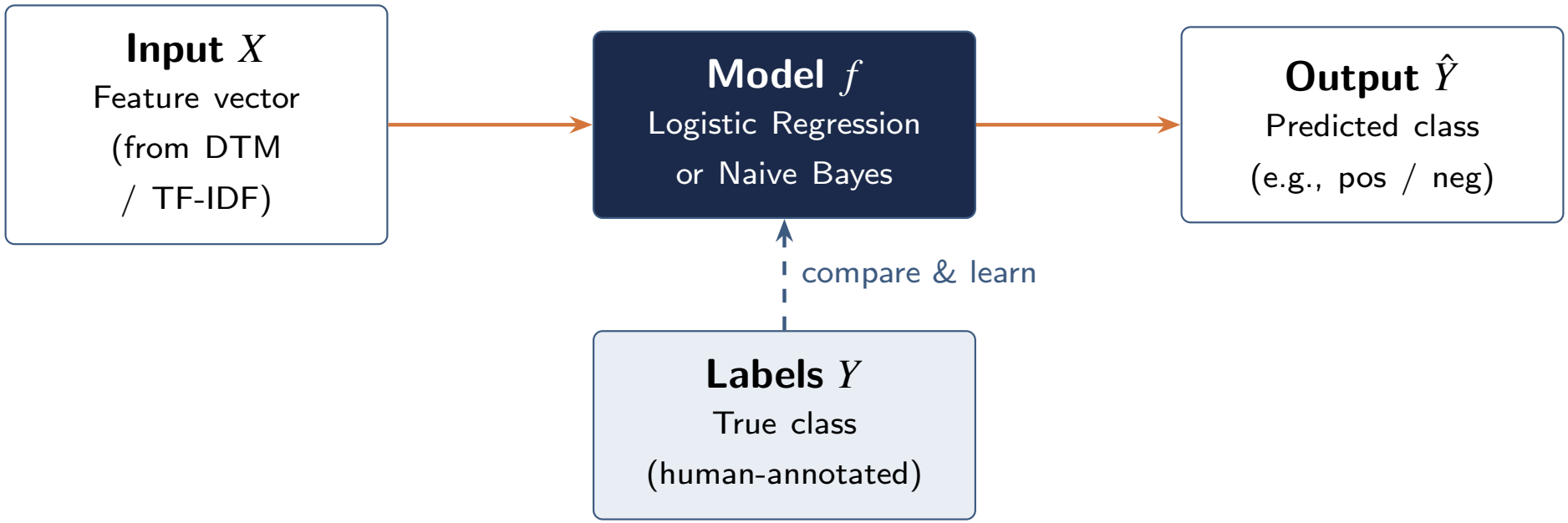
- Accuracy, precision, recall, F1
- The confusion matrix
- Why class balance matters
- Practical: classify Bundestag speeches

Table of Contents

What Is Supervised Learning?

Definition

Supervised learning: Given a set of **labeled examples** (input–output pairs), learn a function that maps new, unseen inputs to the correct output.



The model learns from labeled data during **training**, then makes predictions on new data during **inference**. The labels come from human annotation — which is why the quality of your labeled data is critical.

Supervised Learning in Political Science: Examples

Task	Input (X)	Labels (Y)	Source of labels
Sentiment analysis	Tweet text	Positive / Negative	Crowdworkers
Policy topic coding	Manifesto sentence	Economy / Social / Foreign / ...	Manifesto Project coders
Stance detection	Political statement	Pro / Against / Neutral	Expert annotation
Hate speech detection	Social media post	Hate / Not hate	Trained annotators
Government vs. opposition	Parliamentary speech	Gov. / Opp.	Parliamentary metadata
Left–right position	Party press release	Left / Center / Right	Expert surveys

The Labeling Bottleneck

Labels are expensive. A typical project hand-codes 1,000–5,000 texts, then trains a classifier to label the remaining millions. The quality ceiling of your classifier is the quality of your training labels.

Train / Validation / Test Split

Why split the data? We need to evaluate how the model performs on data it has *never seen during training*. Otherwise we're just measuring memorization.

Practical guidelines:

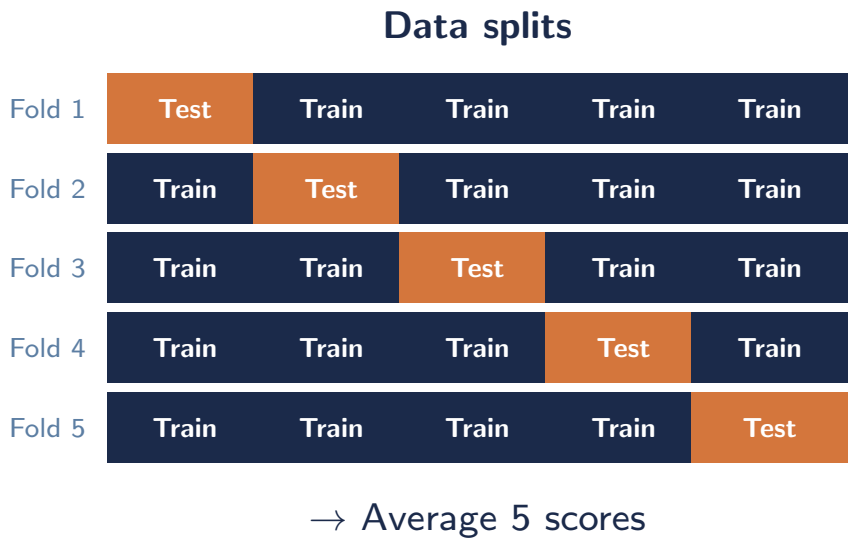
- With ∼2,000 labeled texts: a 70/15/15 split is reasonable.
- With very small datasets (< 500): use k-**fold cross-validation** instead of a fixed split. Each document serves as both training and validation data across different folds.
- **Never** use the test set for any decision-making during development. It's your final exam — you only take it once.
- **Stratify** the split: ensure each class is proportionally represented in all three sets.

Cross-Validation: When You Have Limited Labeled Data

k-Fold Cross-Validation:

1. Split data into k equal parts ("folds")
2. For each fold i: train on all other folds, evaluate on fold i
3. Average the k evaluation scores

Common choices: $k = 5$ or $k = 10$. With $k = 5$, each fold uses 80% for training and 20% for evaluation.



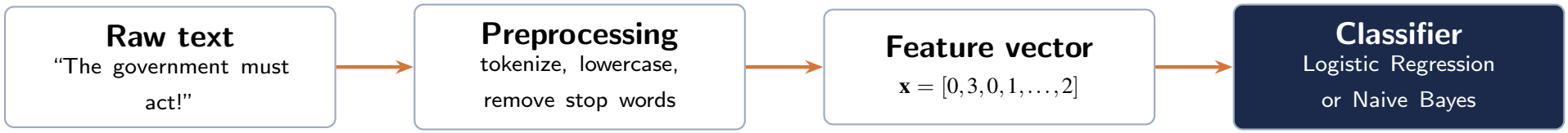
→ Average 5 scores

Why use it?

- Every document gets used for both training and evaluation
- More reliable performance estimate than a single split
- Especially important with small political science datasets ($< 1{,}000$ labeled examples)

From Text to Feature Vectors (Recap from Lecture 1)

Before we can classify, we need to convert each document into a numerical vector.



What feature representation to use?

Bag of Words (raw counts)

x_j = count of word j in document

Simple; sensitive to document length

TF-IDF

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

Down-weights common words; often better than raw counts

Binary (presence/absence)

$x_j = \mathbb{I}[word\ \ j \in d]$

Ignores frequency; works well for short texts like tweets

The choice of feature representation *is itself a modeling decision*. Try multiple options and compare performance.

Table of Contents

Why Start with Logistic Regression?

Logistic regression is the **most widely used baseline** for text classification in social science.

Why it's important:

- Simple, fast, and interpretable
- Often performs surprisingly well — competitive with much more complex models
- Coefficients tell you *which words matter* and in which direction
- Natural baseline: always train LR before trying neural networks
- You already (should) know it from stats courses — now we apply it to text

Limitations:

- Assumes a linear decision boundary
- Cannot capture word interactions (unless you engineer features manually)
- Treats each word independently

The Strong Baseline Rule

In computational social science, you should **always** report logistic regression as a baseline. If a complex deep learning model only marginally improves over LR, the simpler model may be preferable — it's more interpretable and requires less data.

This principle will become very concrete in Lecture 7 when we compare LR, simple neural networks, and fine-tuned BERT on the same task.

Logistic Regression: The Intuition

Goal: Given a feature vector $\mathbf{x}$ (e.g., TF-IDF scores of a speech), predict the probability that the document belongs to class 1 (e.g., "positive sentiment").

Step 1: Linear combination

Compute a weighted sum of the features:

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n = \theta^\top \mathbf{x}$$

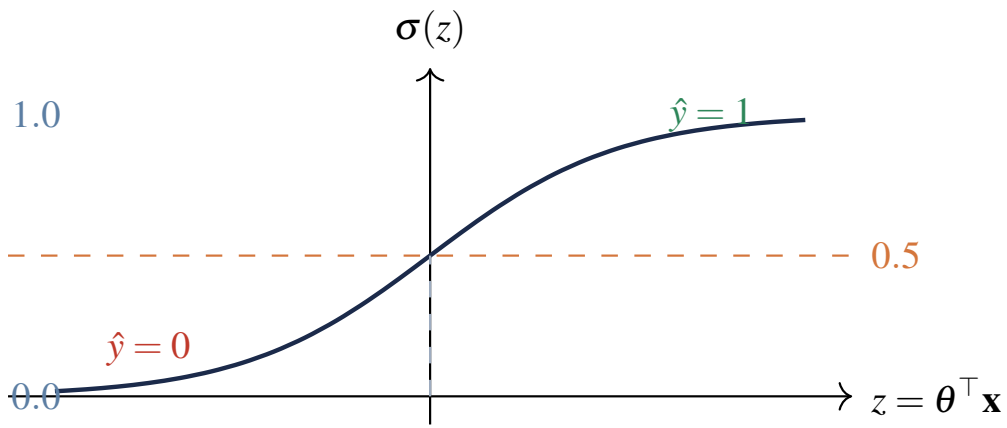
Each weight θ_j captures how much word j pushes toward class 1 (if $\theta_j > 0$) or class 0 (if $\theta_j < 0$).

Problem: z can be any real number ($-\infty$ to $+\infty$). We need a *probability* between 0 and 1.

Step 2: Apply the sigmoid function

Squash z into the range $[0, 1]$:

$$h(\mathbf{x}, \theta) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



The Sigmoid Function in Detail

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Key properties:

- Output is always between 0 and 1 $\Rightarrow$ interpretable as a **probability**
- $\sigma(0) = 0.5$ (the decision boundary)
- As $z \to +\infty$: $\sigma(z) \to 1$
- As $z \to -\infty$: $\sigma(z) \to 0$
- Smooth and differentiable everywhere (important for gradient descent)

The classification rule:

$$\hat{y} = \begin{cases} 1 & \text{if } \sigma(\theta^\top \mathbf{x}) \geq 0.5 \\ 0 & \text{if } \sigma(\theta^\top \mathbf{x}) < 0.5 \end{cases}$$

Equivalently: predict class 1 when $\theta^\top \mathbf{x} \geq 0$.

The 0.5 threshold is a default. For imbalanced classes (e.g., only 5% hate speech), you may want to adjust the threshold to improve recall.

Building a Feature Vector from Training Counts

Idea: instead of using the full vocabulary as features, compress each document into **three numbers**: a bias, the total positive frequency, and the total negative frequency of its words.

Step 1: From the labeled training corpus, count how often each word appears in positive vs. negative tweets.

Word	Freq in Pos class	Freq in Neg class
happy	5	1
learning	2	1
nlp	1	1
sad	1	5
terrible	0	3
⋮	⋮	⋮

Step 2: To encode a new tweet, look up each of its words in this table and sum.

Encoding "happy learning nlp"

Word in tweet	Pos freq	Neg freq
happy	5	1
learning	2	1
nlp	1	1
Sum	**8**	**3**

$$\Rightarrow \quad \mathbf{x} = [\underbrace{1}_{\text{bias}}, \underbrace{8}_{\text{pos sum}}, \underbrace{3}_{\text{neg sum}}]$$

Logistic Regression for Sentiment: Classification

Now we classify the tweet using the feature vector we just built.

Classifying "happy learning nlp"

Feature vector (from previous slide): $\mathbf{x} = [1, 8, 3]$

Learned parameters: $\theta = [0.0001, 0.0015, -0.0012]$

Compute z:

$$z = \underbrace{0.0001}_{\theta_0} \cdot \underbrace{1}_{\text{bias}} + \underbrace{0.0015}_{\theta_1} \cdot \underbrace{8}_{\text{pos}} + \underbrace{(-0.0012)}_{\theta_2} \cdot \underbrace{3}_{\text{neg}} = 0.0001 + 0.012 - 0.0036 = 0.0085$$

Apply sigmoid:

$$\sigma(0.0085) = \frac{1}{1 + e^{-0.0085}} \approx 0.502$$

$\sigma(z) \geq 0.5 \Rightarrow$ **Predict: Positive**

Caution

With only 3 features, the prediction is barely above the 0.5 threshold — the model is almost guessing. In practice, with a full TF-IDF vector of 5,000+ features, the signal is much stronger. This toy example is for understanding the **mechanics**, not for real classification.

What the Weights Tell Us: Interpretability

One of LR's biggest advantages: the learned weights θ_j are directly interpretable.

Hypothetical Weights for Sentiment Classification

Positive sentiment ($\theta_j > 0$):		Negative sentiment ($\theta_j < 0$):	
`excellent`	$+2.31$	`terrible`	-2.45
`great`	$+1.87$	`worst`	-2.12
`happy`	$+1.65$	`hate`	-1.89
`love`	$+1.52$	`disappointing`	-1.63
`recommend`	$+1.34$	`awful`	-1.41

In political science applications:

- Classifying pro- vs. anti-EU stance: large positive weights on "sovereignty", "cooperation"; large negative weights on "Brussels", "bureaucracy"
- The weights reveal *which features drive the prediction* — this is a finding itself
- Compare with neural networks (Lecture 5–7): much harder to interpret

Training: The Cost Function (Loss Function)

How does the model learn the weights θ?
By minimizing a **cost function** that measures how wrong the predictions are.

Binary Cross-Entropy Loss:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h(\mathbf{x}^{(i)},\theta) + (1-y^{(i)})\log(1-h(\mathbf{x}^{(i)},\theta))\right]$$

Unpacking this:

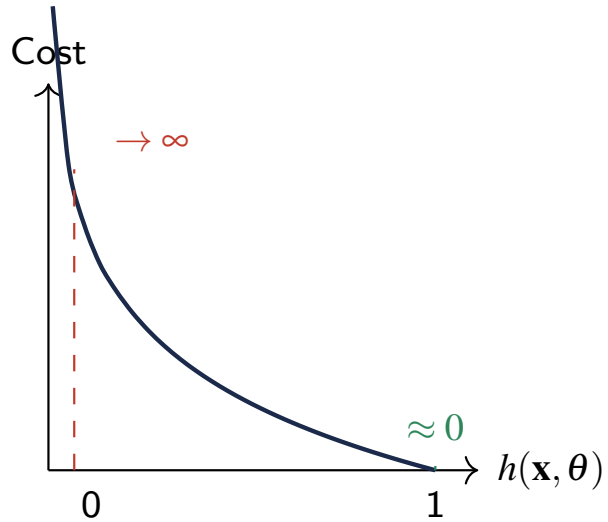
- m = number of training documents
- $y^{(i)}$ = true label for document i (0 or 1)
- $h(\mathbf{x}^{(i)},\theta) = \sigma(\theta^\top \mathbf{x}^{(i)})$ = model's predicted probability
- The negative sign ensures the cost is **positive** (since $\log$ of a probability is negative)
- We **minimize** $J(\theta)$: lower cost = better predictions

Why This Cost Function Makes Sense

The cost function penalizes confident wrong predictions heavily.

When the true label $y = 1$:

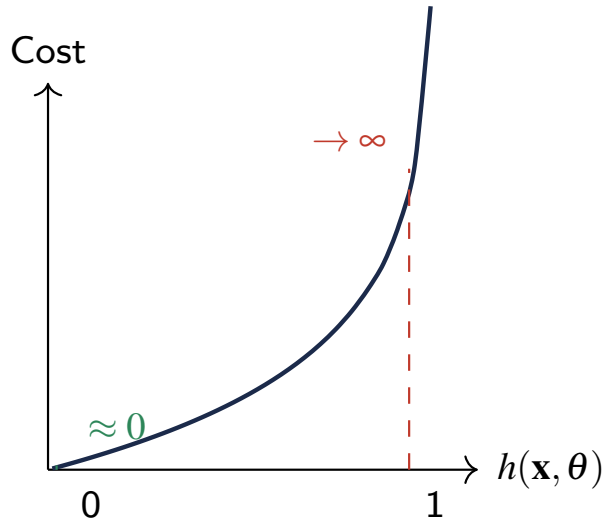
Only the first term matters:
$-\log h(\mathbf{x}, \theta)$



If $h \approx 1$: cost ≈ 0 ✔ (correct!)
If $h \approx 0$: cost $\to \infty$ ✘ (very wrong!)

When the true label $y = 0$:

Only the second term matters:
$-\log(1 - h(\mathbf{x}, \theta))$



If $h \approx 0$: cost ≈ 0 ✔ (correct!)
If $h \approx 1$: cost $\to \infty$ ✘ (very wrong!)

Key insight: The cost is asymmetric — confidently wrong predictions are punished *much more* than uncertain ones. This encourages the model to be calibrated.

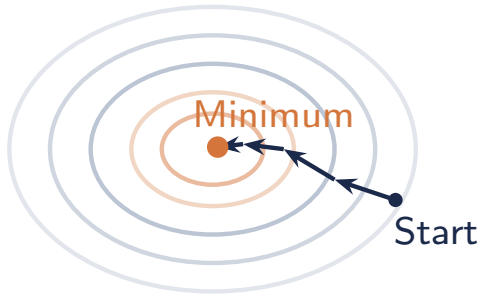
Training: Gradient Descent

How do we find the θ that minimizes the cost $J(\theta)$?

Gradient Descent Algorithm:

1. **Initialize** θ (often to zeros)
2. **Repeat** until convergence:
   a. Compute predictions: $\mathbf{h} = \sigma(\mathbf{X}\theta)$
   b. Compute gradient: $\nabla = \frac{1}{m}\mathbf{X}^\top(\mathbf{h} - \mathbf{y})$
   c. Update weights: $\theta := \theta - \alpha\nabla$
   d. Compute cost $J(\theta)$
3. **Return** learned θ

α = **learning rate** (step size). Too large $\rightarrow$ overshoot. Too small $\rightarrow$ slow convergence.

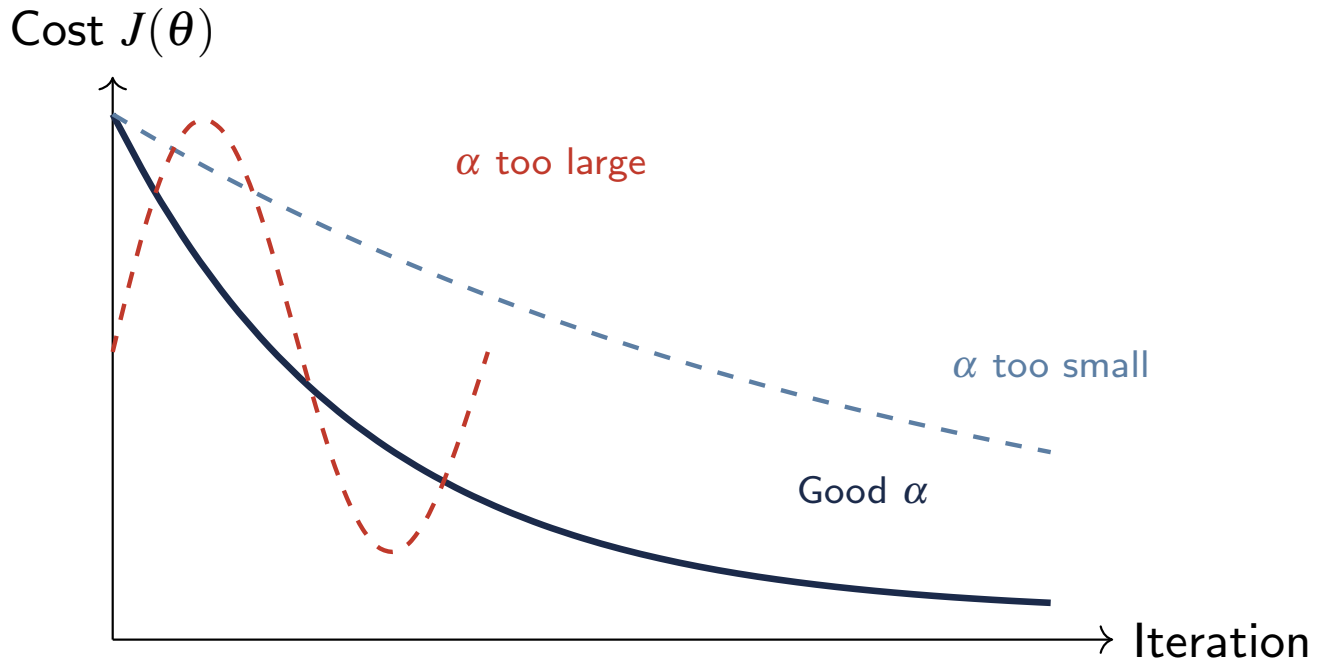


Cost surface (contour plot)

Each step moves θ in the direction that reduces the cost most quickly. The gradient ∇ points "uphill"; we go in the opposite direction.

Gradient Descent: The Learning Curve

Monitoring training: Plot the cost $J(\theta)$ at each iteration. It should decrease.



In practice, you rarely implement gradient descent yourself. Libraries like `scikit-learn` (Python) or `glmnet` (R) use optimized solvers. But understanding the mechanics helps you:

- Diagnose convergence problems
- Understand why some feature representations train faster than others
- Appreciate what changes in Lectures 5–7 when we move to neural networks

Regularization: Preventing Overfitting

Problem: With 10,000+ features (words) and only 2,000 training documents, the model can *memorize* the training data by assigning large weights to rare words.

Overfitting

A model **overfits** when it performs well on training data but poorly on new data. It has learned noise instead of signal. Rare words that happen to co-occur with one class in training may not generalize.

Solution: Add a penalty for large weights.

L2 Regularization (Ridge):

$$J_{\text{reg}}(\theta) = J(\theta) + \lambda \sum_{j=1}^{n} \theta_j^2$$

Shrinks all weights toward zero. Keeps all features but makes them smaller.

L1 Regularization (Lasso):

$$J_{\text{reg}}(\theta) = J(\theta) + \lambda \sum_{j=1}^{n} |\theta_j|$$

Pushes many weights to exactly zero $\Rightarrow$ automatic feature selection. Sparse models.

λ = regularization strength. Larger λ = more regularization. Tune λ using the validation set or cross-validation. In `scikit-learn`, the parameter is called `C` $= 1/\lambda$.

Regularization: A Worked Example

Suppose two weight configurations produce the **same prediction error**: $J(\theta) = 0.5$

Config A: spread weights

$\theta_1 = 3$ (`economy`)
$\theta_2 = 3$ (`welfare`)

Config B: one big weight

$\theta_1 = 5$ (`economy`)
$\theta_2 = 0$ (`welfare`)

Both predict equally well ($J = 0.5$). Which does regularization prefer? Set $\lambda = 0.1$:

L2 (Ridge): penalizes $\theta_1^2 + \theta_2^2$

Config A: $0.5 + 0.1 \times (3^2 + 3^2) = 0.5 + 1.8 = \mathbf{2.3}$ ✔ lower!

Config B: $0.5 + 0.1 \times (5^2 + 0^2) = 0.5 + 2.5 = 3.0$

$\rightarrow$ L2 **prefers A**: spreading the weight across two features is cheaper than concentrating it in one. **All features stay, but get smaller.**

L1 (Lasso): penalizes $|\theta_1| + |\theta_2|$

Config A: $0.5 + 0.1 \times (|3| + |3|) = 0.5 + 0.6 = 1.1$

Config B: $0.5 + 0.1 \times (|5| + |0|) = 0.5 + 0.5 = \mathbf{1.0}$ ✔ lower!

$\rightarrow$ L1 **prefers B**: zeroing out a feature costs nothing, so L1 produces **sparse models** — automatic feature selection.

Table of Contents

A Different Approach: Naive Bayes

Instead of learning weights for a linear boundary, Naive Bayes takes a **probabilistic** approach:

The Core Question

Given that I observe certain words in a document, what is the **probability** that it belongs to each class?

$$P(\text{class} \mid \text{words in document}) = ?$$

Why learn Naive Bayes when we already have logistic regression?

- Extremely fast to train — no iterative optimization, just counting
- Works well even with very small training sets
- Provides a natural probabilistic interpretation
- Strong baseline for text classification; competitive in many settings
- Conceptually different from LR — good to understand both perspectives

Think of it as: LR is a **discriminative** model (learns the boundary directly). NB is a **generative** model (learns how each class generates words, then inverts using Bayes' rule).

Bayes' Theorem: A Quick Refresher

Bayes' Theorem:

$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B)}$$

Political Intuition

You read a news article. You see the word "Klimaschutz" (climate protection). How likely is it that the speaker is from the Grüne party?

$$\begin{aligned} P(\text{Grüne} \mid \text{“Klimaschutz”}) &= \text{probability the speaker is Grüne, given we see this word} \\ P(\text{“Klimaschutz”} \mid \text{Grüne}) &= \text{how often Grüne MPs say “Klimaschutz”} \\ P(\text{Grüne}) &= \text{overall proportion of Grüne speeches in corpus} \\ P(\text{“Klimaschutz”}) &= \text{how often anyone says “Klimaschutz”} \end{aligned}$$

$$P(\text{Grüne} \mid \text{“Klimaschutz”}) = \frac{P(\text{“Klimaschutz”} \mid \text{Grüne}) \cdot P(\text{Grüne})}{P(\text{“Klimaschutz”})}$$

Applying Bayes to Text Classification

For a document with words $w_1, w_2, \ldots, w_n$:

$$P(\text{class} \mid w_1, w_2, \ldots, w_n) = \frac{P(w_1, w_2, \ldots, w_n \mid \text{class}) \cdot P(\text{class})}{P(w_1, w_2, \ldots, w_n)}$$

Problem: Computing $P(w_1, w_2, \ldots, w_n \mid \text{class})$ requires estimating the joint probability of all word combinations. With a vocabulary of 50,000 words, this is impossible — we'd need more data than exists.

Solution: The **"naive" conditional independence assumption**:

$$P(w_1, w_2, \ldots, w_n \mid \text{class}) \approx P(w_1 \mid \text{class}) \cdot P(w_2 \mid \text{class}) \cdots P(w_n \mid \text{class}) = \prod_{i=1}^{n} P(w_i \mid \text{class})$$

This assumes words occur *independently* of each other, given the class. Obviously false ("European" and "Union" co-occur), but works well in practice. This is why it's called "naive."

The Naive Bayes Classifier

For binary classification (e.g., positive vs. negative), we compare:

$$\frac{P(\text{pos} \mid \text{doc})}{P(\text{neg} \mid \text{doc})} = \frac{P(\text{pos})}{P(\text{neg})} \cdot \prod_{i=1}^{n} \frac{P(w_i \mid \text{pos})}{P(w_i \mid \text{neg})}$$

$$\underbrace{\frac{P(\text{pos})}{P(\text{neg})}}_{\text{Prior ratio}} \times \underbrace{\prod_{i=1}^{n} \frac{P(w_i \mid \text{pos})}{P(w_i \mid \text{neg})}}_{\text{Likelihood ratio (for each word)}}$$

How common is each class in the training data?
E.g., 60% positive, 40% negative $\Rightarrow$ ratio $= 1.5$

For each word in the document: how much more likely is this word in positive vs. negative texts?

"happy": $P(\text{happy} \mid \text{pos})/P(\text{happy} \mid \text{neg}) = 3.2$
"terrible": $P(\text{terrible} \mid \text{pos})/P(\text{terrible} \mid \text{neg}) = 0.1$

Decision rule: If the ratio > 1, predict positive. If < 1, predict negative.

Estimating the Probabilities

How do we compute $P(w \mid \textbf{class})$? Simply count!

Word Frequencies by Class

Positive tweets:
"I am happy because I am learning NLP"
"I am happy"

Negative tweets:
"I am sad, I am not learning NLP"
"I am sad"

Word	Count in Pos	Count in Neg
I	3	3
am	3	3
happy	2	0
because	1	0
learning	1	1
NLP	1	1
sad	0	2
not	0	1
Total words	**11**	**11**

Problem: Zero Probabilities

What if a word never appeared in one class during training?

The Zero Problem

"happy" appears 2 times in positive, **0 times** in negative.

$$P(\text{happy} \mid \text{neg}) = \frac{0}{11} = 0$$

Since we *multiply* all word probabilities together:

$$\prod_i P(w_i \mid \text{neg}) = \ldots \times 0 \times \ldots = 0$$

The entire product collapses to zero! A single unseen word wipes out all other evidence.

This is a critical flaw. A new test document containing "happy" would get $P(\text{neg} \mid \text{doc}) = 0$, regardless of all other words. We need a fix.

Solution: Laplacian (Add-1) Smoothing

Idea: Pretend every word was seen at least once in every class.

Smoothed probability:

$$P(w \mid \text{class}) = \frac{\text{count}(w, \text{class}) + 1}{\sum_{w' \in V} \text{count}(w', \text{class}) + |V|}$$

where $|V|$ is the vocabulary size (total number of unique words).

Smoothed Probabilities

Vocabulary size $|V| = 8$. Total words in positive class $= 11$.

Without smoothing:

$P(\text{happy} \mid \text{pos}) = 2/11 = 0.182, \quad P(\text{happy} \mid \text{neg}) = 0/11 = \mathbf{0}$ ✗

With smoothing:

$P(\text{happy} \mid \text{pos}) = (2+1)/(11+8) = 3/19 = 0.158$

$P(\text{happy} \mid \text{neg}) = (0+1)/(11+8) = 1/19 = 0.053$ ✓ — small but not zero!

From Products to Sums: Log-Likelihood

Practical problem: Multiplying many small probabilities $\Rightarrow$ numerical underflow.

$$P(\text{pos} \mid \text{doc}) \propto P(\text{pos}) \times \underbrace{P(w_1 \mid \text{pos}) \times P(w_2 \mid \text{pos}) \times \cdots \times P(w_n \mid \text{pos})}_{\text{Many numbers like } 0.003 \times 0.0001 \times \cdots \to 0.0000000\ldots}$$

With a document of 200 words, this product can be $< 10^{-200}$, smaller than a computer can represent.

Solution: Take the logarithm. Products become sums.

$$\log \frac{P(\text{pos} \mid \text{doc})}{P(\text{neg} \mid \text{doc})} = \underbrace{\log \frac{P(\text{pos})}{P(\text{neg})}}_{\text{log-prior}} + \underbrace{\sum_{i=1}^{n} \log \frac{P(w_i \mid \text{pos})}{P(w_i \mid \text{neg})}}_{\text{sum of log-likelihood ratios}}$$

Decision rule: If this sum > 0, predict positive. If < 0, predict negative.

Each word contributes a **log-likelihood ratio**: positive if the word is more common in positive texts, negative if more common in negative texts. Neutral words contribute ≈ 0.

Naive Bayes: Worked Example

Classify: "I am happy, not sad" as positive or negative.

Step 1: Log-prior
Training set: 2 positive, 2 negative tweets $\Rightarrow \log(2/2) = \log(1) = 0$

Step 2: Log-likelihood ratios (using smoothed counts, $|V| = 8$):

Word	$P(w \mid \text{pos})$	$P(w \mid \text{neg})$	**Ratio**	$\log(\text{ratio})$
I	$4/19 = 0.211$	$4/19 = 0.211$	1.00	0.00
am	$4/19 = 0.211$	$4/19 = 0.211$	1.00	0.00
happy	$3/19 = 0.158$	$1/19 = 0.053$	3.00	$+1.10$
not	$1/19 = 0.053$	$2/19 = 0.105$	0.50	-0.69
sad	$1/19 = 0.053$	$3/19 = 0.158$	0.33	-1.10

Step 3: Sum it up
$\text{Score} = 0 + (0.00 + 0.00 + 1.10 + (-0.69) + (-1.10)) = -0.69$

$\text{Score} < 0 \Rightarrow$ **Predict: Negative**

Interpretation: "happy" pushes positive ($+1.10$), but "not" and "sad" together push more strongly negative (-1.79). The negative words win. Note that NB handles "not" correctly here because "not" is empirically more common in negative texts.

Naive Bayes: Training Summary

The entire training procedure for Naive Bayes:

1. **Count** the number of documents in each class $\Rightarrow$ compute the prior $P(\text{class})$
2. **Count** how often each word appears in each class
3. **Apply smoothing**: add 1 (or α) to all counts
4. **Compute** $P(w \mid \text{class})$ for every word and class
5. **Store** the log-likelihood ratios: $\lambda_w = \log \frac{P(w|\text{pos})}{P(w|\text{neg})}$ for each word

To classify a new document:

$$\text{Score} = \log \frac{P(\text{pos})}{P(\text{neg})} + \sum_{w \in \text{document}} \lambda_w$$

If score > 0: predict positive. Otherwise: predict negative.

Key Concept

No gradient descent! No iteration! Just **counting and dividing**. This is why NB is so fast — it "trains" in a single pass through the data.

Logistic Regression vs. Naive Bayes: Comparison

	Logistic Regression	**Naive Bayes**
Type	Discriminative (learns boundary)	Generative (models $P(w \mid \text{class})$)
Training	Iterative optimization (gradient descent)	Single-pass counting
Speed	Moderate (seconds to minutes)	Very fast (milliseconds)
Features	Works with any features; naturally handles correlated features	Assumes feature independence ("naive")
Small data	Needs regularization	Works well out of the box
Large data	Usually wins	Often competitive, sometimes worse
Interpretability	Weights show word importance	Log-likelihood ratios show word importance
Probabilities	Well-calibrated	Often poorly calibrated (overconfident)

Practical Recommendation

Try both. Report both. If they agree, you can be more confident in your results. If they disagree, investigate why — the disagreement itself is informative about your data.

Table of Contents

Why Accuracy Is Not Enough

A Misleading Classifier

Task: Detect hate speech in tweets. Your dataset: 95% non-hate, 5% hate speech.

"Classifier": Always predict "not hate speech" (never flags anything).

Accuracy: 95%! Sounds great, right?

Reality: This classifier is *completely useless*. It catches zero hate speech. It has learned nothing — it just exploits the class imbalance.

This is why we need multiple evaluation metrics.

Accuracy only tells you the overall proportion of correct predictions. It doesn't tell you *what kind of errors* the model makes. In political science, different errors have different costs:

- Missing hate speech (false negative) is dangerous
- Flagging legitimate speech as hate (false positive) raises free-speech concerns

The Confusion Matrix

A 2×2 table that shows exactly what the classifier gets right and wrong.

	Predicted: Pos	**Predicted: Neg**
Actual:Pos	**True Positive** (TP) Correctly identified	**False Negative** (FN) Missed
Actual:Neg	**False Positive** (FP) False alarm	**True Negative** (TN) Correctly rejected

In political science terms:

- **TP**: Correctly flagged hate speech
- **FP**: Flagged legitimate speech as hate
- **FN**: Missed actual hate speech
- **TN**: Correctly passed legitimate speech

Different costs for different tasks:

- Content moderation: **FN is dangerous** (missed hate speech)
- Policy coding: **FP and FN matter equally** (miscoding in either direction)
- Screening: **FP is acceptable** if FN is very costly (cast a wide net)

Precision, Recall, and F1 Score

Precision: Of all documents the model *predicted* as positive, how many are actually positive?

$$\text{Precision} = \frac{TP}{TP + FP}$$

"When the model says positive, how often is it right?"
High precision = few false alarms.

Recall (Sensitivity): Of all *actually* positive documents, how many did the model find?

$$\text{Recall} = \frac{TP}{TP + FN}$$

"Of all the positives out there, how many did we catch?"
High recall = few misses.

F1 Score: The harmonic mean of precision and recall.

$$F_1 = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Balances both errors. $F_1 = 1$ is perfect. The harmonic mean punishes extreme imbalance: if either precision or recall is near 0, F_1 will also be near 0.

Accuracy: Overall correctness.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Only useful when classes are balanced.

Precision vs. Recall: A Worked Example

Hate Speech Detection

Test set: 200 tweets (180 normal, 20 hate speech).

Model's predictions:

	Predicted: Hate	**Predicted: Normal**
Actual: Hate	TP $= 12$	FN $= 8$
Actual: Normal	FP $= 5$	TN $= 175$

Accuracy $= (12+175)/200 = 93.5\%$

Looks good! But...

Precision $= 12/(12+5) = 70.6\%$

Of flagged tweets, 70.6% are actually hate.

Recall $= 12/(12+8) = 60.0\%$

We *missed* 40% of hate speech!

F1 $= 2 \times 0.706 \times 0.600/(0.706+0.600) = 64.9\%$

Takeaway: 93.5% accuracy masks the fact that we miss 40% of hate speech. Always report precision, recall, and F1 — especially with imbalanced classes.

The Precision–Recall Trade-off

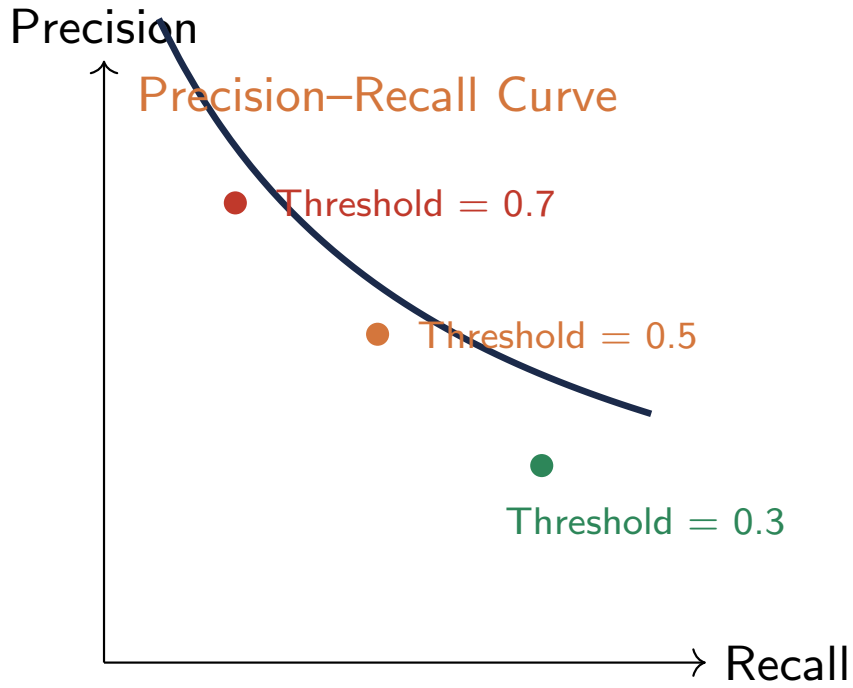
You can usually improve one at the cost of the other.

Lowering the classification threshold (e.g., from 0.5 to 0.3):

→ The model is *more willing* to predict positive
→ Catches more true positives (recall ↑)
→ But also produces more false alarms (precision ↓)

Raising the classification threshold (e.g., from 0.5 to 0.7):

→ The model is *more conservative*
→ Fewer false alarms (precision ↑)
→ But misses more true positives (recall ↓)



The curve bows toward the top-right corner for better models. Choose the operating point based on your task's cost structure.

Dealing with Class Imbalance

In political science, many classification tasks have imbalanced classes.

Examples of imbalance:

- Hate speech: $< 5\%$ of tweets
- Populist rhetoric: $< 10\%$ of speeches
- Crisis-related news: $< 3\%$ of articles
- Manifesto sentences about EU: $< 8\%$

Why it's a problem:

- Models learn to always predict the majority class
- High accuracy but useless predictions
- Gradient descent sees mostly one class during training

Solutions:

1. **Stratified sampling**: Ensure both classes are proportionally represented in train/val/test splits
2. **Class weighting**: Tell the model to penalize errors on the minority class more heavily
   In `scikit-learn`:
   `class_weight='balanced'`
3. **Oversampling**: Duplicate minority class examples (or use SMOTE)
4. **Undersampling**: Use only a subset of the majority class
5. **Evaluate with F1**, not accuracy

Multi-class Classification

Many political science tasks have more than two classes.

Policy Topic Coding

Classify each manifesto sentence into one of 7 policy domains:
Economy, Social Policy, Foreign Affairs, Environment, Immigration, Security, Governance.

Extending our binary classifiers:

- **Logistic Regression**: Replace sigmoid with **softmax**:

$$P(\text{class}_k \mid \mathbf{x}) = \frac{e^{\theta_k^\top \mathbf{x}}}{\sum_{j=1}^{K} e^{\theta_j^\top \mathbf{x}}}$$

  Gives a probability distribution over all K classes. Uses cross-entropy loss.

- **Naive Bayes**: Naturally extends to K classes. Compute $P(\text{class}_k \mid \text{doc})$ for each class; predict the class with the highest score.

- **One-vs-rest (OvR)**: Train K binary classifiers, each distinguishing one class from all others. Predict the class whose classifier has the highest confidence.

Multi-class Evaluation Metrics

With $K > 2$ classes, we get a $K \times K$ confusion matrix.

How to compute precision, recall, F1?

Macro-averaging:

1. Compute P, R, F1 for *each class separately* (treating it as "positive" vs. "all others")
2. Average across classes

$$F_1^{\text{macro}} = \frac{1}{K} \sum_{k=1}^{K} F_1^{(k)}$$

Treats all classes equally, even rare ones.
Use when all classes matter equally.

Weighted averaging:

1. Compute P, R, F1 for each class
2. Weight by class size (number of examples)

$$F_1^{\text{weighted}} = \sum_{k=1}^{K} \frac{n_k}{N} F_1^{(k)}$$

Gives more influence to frequent classes.
Better reflects overall performance.

Which to report?

Report **both** macro and weighted F1. If they differ substantially, that means the model performs much worse on rare classes. Also report the per-class metrics for transparency.

Table of Contents

Practical Session: Overview

Classifying Bundestag Speeches by Party

Task: Given a preprocessed parliamentary speech, predict the party of the speaker.

What we'll do:

1. Prepare the data (from Lecture 1's preprocessed corpus)
2. Train a Logistic Regression classifier
3. Train a Naive Bayes classifier
4. Evaluate both using the metrics we just learned
5. Compare: which performs better? Which words drive predictions?

We'll use: `quanteda` for text processing, `quanteda.textmodels` for Naive Bayes, and `glmnet` for regularized Logistic Regression. All in R.

```
install.packages(c("quanteda", "quanteda.textmodels",
  "glmnet", "caret", "tidyverse"))
```

Table of Contents

Key Takeaways

1. **Supervised text classification** is the most common quantitative text analysis task in political science. You need labeled data, a feature representation, and a classifier.
2. **Logistic Regression** is a strong, interpretable baseline. The learned weights tell you which words drive the prediction — this is a finding in itself.
3. **Naive Bayes** is fast, simple, and probabilistic. It works by counting word frequencies per class and applying Bayes' rule with the "naive" independence assumption.
4. **Evaluation must go beyond accuracy.** Use the confusion matrix, precision, recall, and F1. Especially with imbalanced classes, accuracy is misleading.
5. **Always split your data** into train/validation/test. Never evaluate on data the model has seen. Use cross-validation when data is scarce.
6. **Report multiple models.** If LR and NB agree, your findings are robust. If they disagree, investigate why.

Home Assignment (Optional)

Task: Build a text classifier for a political science application.

Instructions:

1. Choose a classification task: sentiment, policy topic, stance, government/opposition, or your own idea.
2. Preprocess the data (using your pipeline from Lecture 1).
3. Train **both** Logistic Regression and Naive Bayes.
4. Evaluate using **accuracy, precision, recall, and F1**. Report the confusion matrix.
5. Experiment with at least one variation:
   - Raw counts vs. TF-IDF
   - Ridge vs. Lasso regularization
   - With vs. without stop word removal
   - Different minimum document frequency thresholds
6. Write a **1.5-page report**: describe your task, results, which model won, what the most predictive features reveal substantively, and how sensitive the results are to your choices.

Submit: R script + report. Due before Lecture 3.

Readings

Required:

- Jurafsky, D. & Martin, J.H. (2024). *Speech and Language Processing*. Chapter 4: Naive Bayes, Text Classification, and Sentiment. https://web.stanford.edu/~jurafsky/slp3/4.pdf

Recommended:

- Hopkins, D. & King, G. (2010). "A Method of Automated Nonparametric Content Analysis for Social Science." *American Journal of Political Science*, 54(1), 229–247.

Other relevant literature:

- Manning, C.D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Chapter 13: Text Classification and Naive Bayes.
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning*. Chapter 4.4: Logistic Regression (for the math behind regularization).

Next week: Word Embeddings & Vector Spaces

From counting words to understanding meaning: Word2Vec, GloVe, and political semantics

Logistic Regression: Tiny Worked Example

Task: classify a sentence as $1 =$ positive or $0 =$ negative.
Imagine that in our training set we have 2 sentences. *"It is so good!"*, has label 1 (positive), and *"That was too bad."*, has label 0 (negative)

Features

$x_1 = \mathbf{1}(\text{“good” appears})$,
$x_2 = \mathbf{1}(\text{“bad” appears})$

With a bias term: $\mathbf{x} = [1, x_1, x_2]$

Training examples

“good” $\rightarrow \mathbf{x}^{(1)} = [1,1,0],\ y^{(1)} = 1$

“bad” $\rightarrow \mathbf{x}^{(2)} = [1,0,1],\ y^{(2)} = 0$

Model

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2, \qquad h(\mathbf{x}) = \sigma(z) = \frac{1}{1+e^{-z}}$$

Interpretation of the weights:

- $\theta_1 > 0$: the word good pushes toward the positive class
- $\theta_2 < 0$: the word bad pushes toward the negative class
- θ_0: bias / intercept

Gradient Descent: One Update on "good"

Setup

Initialize weights:

$\theta = [0, 0, 0]$

Learning rate: $\alpha = 1$

Training example:

$\mathbf{x} = [1, 1, 0]$, $y = 1$

Five steps of one gradient update

Step 1 — Linear score: $z = \theta^\top \mathbf{x} = 0$

Step 2 — Prediction: $h = \sigma(0) = 0.5$

Step 3 — Error: $h - y = 0.5 - 1 = -0.5$

Step 4 — Gradient:

$\nabla = (h - y)\,\mathbf{x} = -0.5 \cdot [1, 1, 0] = [-0.5, -0.5, 0]$

Step 5 — Update: $\theta := \theta - \alpha \nabla = [0.5, 0.5, 0]$

Result

After seeing a **positive** example containing good, the weight θ_1 for good increased from 0 to 0.5. The model is starting to learn that good signals positive sentiment.

Gradient Descent: One Update on "bad"

Setup

Current weights:

$\theta = [0.5,\ 0.5,\ 0]$

Learning rate: $\alpha = 1$

Training example:

$\mathbf{x} = [1, 0, 1],\ y = 0$

Five steps of one gradient update

Step 1 — Linear score:

$z = 0.5 \cdot 1 + 0.5 \cdot 0 + 0 \cdot 1 = 0.5$

Step 2 — Prediction: $h = \sigma(0.5) \approx 0.622$

Step 3 — Error: $h - y = 0.622 - 0 = 0.622$

Step 4 — Gradient:

$\nabla = (h - y)\,\mathbf{x} = 0.622 \cdot [1, 0, 1] = [0.622,\ 0,\ 0.622]$

Step 5 — Update:

$\theta := \theta - \alpha \nabla := [0.5,\ 0.5,\ 0] - 1 * [0.622,\ 0,\ 0.622] = [-0.122,\ 0.5,\ -0.622]$

Result

After seeing a **negative** example containing bad, the weight θ_2 for bad dropped from 0 to -0.622. The model is learning that bad signals negative sentiment.

What the Model Learned

After two updates, the weights are: $\theta = [-0.122,\ 0.5,\ -0.622]$

$$z = -0.122 + 0.5x_1 - 0.622x_2$$

Weight interpretation

good: $+0.5$
$\Rightarrow$ pushes toward positive

bad: -0.622
$\Rightarrow$ pushes toward negative

bias: -0.122
$\Rightarrow$ baseline shift

Check: "good"

$\mathbf{x} = [1, 1, 0]$

$z = -0.122 + 0.5 = 0.378$

$h = \sigma(0.378) \approx 0.593$

Predict: Positive ✔

Check: "bad"

$\mathbf{x} = [1, 0, 1]$

$z = -0.122 - 0.622 = -0.744$

$h = \sigma(-0.744) \approx 0.322$

Predict: Negative ✔

After just two gradient steps, the model classifies both training examples correctly.

Main Takeaway: Weights and Gradient Descent

Weights are the parameters the model learns: $\theta = [\theta_0, \theta_1, \theta_2]$

Gradient descent is the procedure used to update them: $\theta := \theta - \alpha \nabla$

At each iteration

1. Start with the **current weights** θ
2. Compute the score: $z = \theta^\top \mathbf{x}$
3. Compute the prediction: $h = \sigma(z)$
4. Compute the gradient: $\nabla = (h - y)\,\mathbf{x}$
5. Update the weights: $\theta := \theta - \alpha \nabla$

Intuition

- If a word appears in a **positive** example, its weight tends to go **up**
- If a word appears in a **negative** example, its weight tends to go **down**
- After many iterations, the weights converge to values that minimize the cost function

This is the same algorithm used for 5,000-feature TF-IDF vectors — just more dimensions.

Probability Refresher

1. Simple probability

$$P(A) = \frac{\text{number of cases where } A \text{ happens}}{\text{total number of cases}}$$

Example: if 30 out of 100 speeches are by Labour, then

$$P(\text{Labour}) = \frac{30}{100} = 0.3$$

2. Conditional probability

$$P(A \mid B) = \frac{\text{number of cases where both } A \text{ and } B \text{ happen}}{\text{number of cases where } B \text{ happens}}$$

Example: if 20 speeches contain ‘‘`austerity`’’, and 8 of them are by Labour, then

$$P(\text{Labour} \mid \text{“austerity”}) = \frac{8}{20} = 0.4$$

3. Reverse conditional probability

$$P(\text{“austerity”} \mid \text{Labour}) = \frac{\text{number of Labour speeches containing ‘‘}\texttt{austerity}\text{’’}}{\text{total number of Labour speeches}}$$

If 8 of 30 Labour speeches contain ‘‘`austerity`’’, then

$$P(\text{“austerity”} \mid \text{Labour}) = \frac{8}{30} \approx 0.26$$

Key idea: Bayes’ theorem lets us connect these two conditional probabilities.