Lecture 2 — Classical Text Classification

Logistic Regression & Naive Bayes for Political Text

Tamara Grechanaya
Multimodal Computational Methods in Political Science

Summer Semester 2026

Contents

1 Learning objectives

By the end of this lecture, you should be able to:

1. Explain the supervised learning framework: features, labels, training, and prediction.
2. Understand why and how to split data into training, validation, and test sets.
3. Explain logistic regression intuitively — what it does, what the sigmoid function is, and what the learned weights mean.
4. Explain Naive Bayes intuitively — how it uses word counts and Bayes' rule to classify text.
5. Evaluate a classifier using accuracy, precision, recall, and F1 — and explain why accuracy alone is misleading with imbalanced classes.
6. Train both classifiers on a political text corpus in R and compare their performance.

---

2 Part I — The supervised learning framework

2.1 What is supervised learning?

In the broadest terms, supervised learning is the task of learning a rule from examples. You show an algorithm a set of inputs (called *features*) along with the correct answers (called *labels*), and the algorithm figures out a pattern that maps inputs to outputs. Once it has learned the pattern, it can predict the label for new inputs it has never seen before.

Think of it like studying for a multiple-choice exam with an answer key. You study the questions and the correct answers (the training phase). Then, during the actual exam, you encounter new questions you have never seen and apply the patterns you learned to answer them (the prediction phase). The hope is that you learned *generalizable* patterns, not just memorized the specific answer-key pairs.

In our context, supervised learning for text works like this:

- **Input (features)**: a numerical representation of a political text — for example, the TF-IDF vector we built in Lecture 1.
- **Label**: the category we want to predict — for example, the party of the speaker, the policy topic, or whether the sentiment is positive or negative.
- **Training**: the algorithm sees many text–label pairs and learns which words (or combinations of words) are associated with which labels.
- **Prediction**: given a new, unseen text, the algorithm uses the patterns it learned to predict the label.

2.2 Examples in political science

To make this concrete, here are several real applications of supervised text classification in political science:

Task	Input	Labels	Who provides labels?
Sentiment analysis	Tweet text	Positive / Negative	Crowdworkers or trained annotators
Policy topic coding	Manifesto sentence	Economy / Social / Foreign / …	Manifesto Project coders
Stance detection	Political statement	Pro / Against / Neutral	Expert annotators
Hate speech detection	Social media post	Hate speech / Not hate speech	Trained annotators

In each case, the researcher first produces a set of hand-labeled examples (typically 1,000 to 5,000 documents), trains a classifier on those examples, and then uses the classifier to automatically label the remaining thousands or millions of documents. This is the core bargain of supervised learning: you invest a fixed amount of human effort in labeling, and the machine scales it up to the full corpus.

The quality ceiling of your classifier is the quality of your training labels. If your human annotators disagree about what counts as "hate speech," the classifier will also be confused. Garbage labels in, garbage predictions out. This is why careful annotation design is just as important as the choice of algorithm.

2.3 Why you need to split your data

Suppose you train a classifier on 2,000 labeled speeches and it correctly predicts the label for 95% of them. Is it a good classifier? You might think so, but the answer is: *you have no idea*. The 95% accuracy is measured on the same data the model learned from. The model might have simply memorized the training data (this is called *overfitting*) rather than learning generalizable patterns. It might predict perfectly on training data but fail miserably on new speeches it has never seen.

This is why we split our labeled data into separate sets:

Training set (60–80% of data): The model learns from this data. It sees both the texts and their labels and adjusts its internal parameters to minimize errors on this set.

Validation set (10–20%): After training, we evaluate the model on this set. The model has *never seen* these documents during training, so performance on the validation set tells us how well the model generalizes. We use the validation set to make decisions like: should we use TF-IDF or raw counts? Should we add bigrams? Should we increase regularization? Every time we try a different configuration, we check its performance on the validation set.

Test set (10–20%): This set is used *exactly once*, at the very end, to report the final performance. You never make any decisions based on test-set performance. Think of it as the final exam — you take it once, and the result is what you report. If you keep checking the test set and adjusting your model based on what you see, you are effectively "training" on the test set, and your reported performance will be optimistic.

A crucial detail is **stratification**: make sure each class is proportionally represented in all three sets. If your corpus is 70% government speeches and 30% opposition speeches, each set should also be 70/30. Without stratification, you might accidentally put all the opposition speeches in the training set and have none to test on.

2.4 Cross-validation for small datasets

In political science, we often have limited labeled data — perhaps only 500 to 1,000 hand-coded documents. A fixed 70/15/15 split would leave only 75–150 documents for testing, which is a very noisy estimate of performance. In these situations, we use **k-fold cross-validation**.

The idea is simple. Divide your data into k equal-sized parts (called *folds*). For each fold: hold that fold out as the test set, train on the remaining k−1 folds, and evaluate on the held-out fold. After k rounds, every document has been used exactly once as a test document. Average the k performance scores to get your estimate.

Common choices are $k = 5$ or $k = 10$. With $k = 5$, each round uses 80% for training and 20% for testing, and every document gets tested. This gives a much more reliable performance estimate than a single split, especially when data is scarce.

2.5 From text to feature vectors (recap)

Before we can classify anything, we need to convert each document into a numerical vector. This is what we did in Lecture 1. The classifier never sees raw text — it only sees numbers.

The three options we covered:

- **Bag of words (raw counts)**: each feature is the count of a particular word in the document. Simple; sensitive to document length (longer documents have bigger counts).
- **TF-IDF**: each feature is the TF-IDF weight of a word. Automatically down-weights common words and up-weights distinctive ones. Often performs better than raw counts.
- **Binary (presence/absence)**: each feature is 1 if the word appears in the document and 0 if not, regardless of how often it appears. Useful for short texts like tweets where frequency is not informative.

Which representation you choose is itself a modeling decision — and it can affect your results. We will compare them in the practical session.

---

3 Part II — Logistic regression for text

3.1 Why start with logistic regression?

Logistic regression is the single most important baseline for text classification. It is simple, fast, well-understood, and it performs surprisingly well — often competitive with methods that are far more complex. In many published political science papers using text classification, logistic regression is either the main method or the baseline that everything else is compared against.

The key advantage of logistic regression over more complex methods (like neural networks or BERT, which we will cover in later lectures) is *interpretability*. Logistic regression assigns a weight to each word, and that weight tells you directly how much that word pushes the prediction toward one class or the other. If you train a classifier to distinguish government from opposition speeches, you can look at the weights and immediately see which words are most associated with each side. This is a finding in itself.

3.2 The intuition: from words to probabilities

Here is the core idea of logistic regression in plain language. We want to predict the probability that a document belongs to a particular class — say, the probability that a speech is by a Labour MP (class 1) rather than a Conservative MP (class 0).

The model works in two steps:

Step 1: Compute a score. Take the feature vector of the document (for example, its TF-IDF values) and compute a weighted sum. Each word has a weight, and we multiply each word's TF-IDF value by its weight and add them all up:

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

Here, $x_1, x_2, \ldots, x_n$ are the feature values (TF-IDF scores for each word), and $\theta_0, \theta_1, \ldots, \theta_n$ are the weights that the model will learn. The term θ_0 is called the *intercept* or *bias* — it captures the baseline probability when all features are zero.

What does this weighted sum mean? Each weight θ_j captures how much the corresponding word pushes the prediction toward class 1 (if the weight is positive) or class 0 (if the weight is negative). If "NHS" has a

weight of +1.5 and "enterprise" has a weight of −0.8, then documents containing "NHS" are pushed toward Labour and documents containing "enterprise" are pushed toward Conservative.

The problem is that this score z can be any number from $-\infty$ to $+\infty$. We need a probability — a number between 0 and 1.

Step 2: Convert the score to a probability using the sigmoid function.

3.3 The sigmoid function explained

The sigmoid function is a mathematical function that takes any real number and squeezes it into the range between 0 and 1. Its formula is:

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

If you have not seen the number e before: it is a mathematical constant approximately equal to 2.718. It is the base of the natural logarithm and appears everywhere in probability and statistics. For our purposes, you just need to know that e^{-z} gets very large when z is very negative and gets very close to zero when z is very positive.

Let us trace through a few values to build intuition:

- When $z = 0$: $\sigma(0) = \frac{1}{1+e^0} = \frac{1}{1+1} = 0.5$. Right in the middle — maximum uncertainty.
- When $z = 5$ (a large positive number): $e^{-5} \approx 0.0067$, so $\sigma(5) = \frac{1}{1+0.0067} \approx 0.993$. Very close to 1 — the model is confident this is class 1.
- When $z = -5$ (a large negative number): $e^5 \approx 148.4$, so $\sigma(-5) = \frac{1}{1+148.4} \approx 0.0067$. Very close to 0 — the model is confident this is class 0.
- When $z = 2$: $\sigma(2) \approx 0.88$. Fairly confident this is class 1, but not certain.
- When $z = -1$: $\sigma(-1) \approx 0.27$. Leaning toward class 0.

The shape of the sigmoid is an S-curve (the word "sigmoid" comes from the Greek letter sigma, which is S-shaped). It starts near 0 for very negative inputs, passes through 0.5 at the midpoint (z = 0), and approaches 1 for very positive inputs. The transition from "probably class 0" to "probably class 1" happens around z = 0 — this is the *decision boundary*.

The classification rule is simple: if $\sigma(z) \geq 0.5$, predict class 1. If $\sigma(z) < 0.5$, predict class 0. Equivalently: predict class 1 whenever $z \geq 0$.

3.4 A concrete example

Suppose we have trained a sentiment classifier on political tweets. After training, the model has learned three weights:

- $\theta_0 = 0.0001$ (intercept)
- $\theta_1 = 0.0015$ (weight for the "sum of positive word frequencies" feature)
- $\theta_2 = -0.0012$ (weight for the "sum of negative word frequencies" feature)

A new tweet has features: bias = 1, sum of positive frequencies = 8, sum of negative frequencies = 3.

Step 1: Compute the score:

$$z = 0.0001 \times 1 + 0.0015 \times 8 + (-0.0012) \times 3 = 0.0001 + 0.012 - 0.0036 = 0.0085$$

Step 2: Apply sigmoid:

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

This is just barely above 0.5, so the model predicts: Positive. But the probability is very close to 0.5, meaning the model is not confident at all. With more informative features (like a full TF-IDF vector with thousands of words), the model would typically produce scores further from 0.5 and therefore more confident predictions.

3.5 What the weights tell us: interpretability

One of the biggest practical advantages of logistic regression is that the learned weights are directly interpretable. After training, you can inspect which words have the largest positive weights (most predictive of class 1) and the largest negative weights (most predictive of class 0).

For example, in a classifier trained to distinguish government from opposition speeches, you might find:

Words with large *positive* weights (predict government): "investment", "delivering", "committed", "progress", "ensuring"

Words with large *negative* weights (predict opposition): "failure", "crisis", "shameful", "cuts", "demand"

This makes intuitive sense: government speeches use more positive, forward-looking language, while opposition speeches use more critical, negative language. The fact that we can *see* this directly in the weights is valuable — it means we can interpret not just what the model predicts, but *why* it predicts what it does. This interpretability is lost with more complex models like neural networks, where the thousands of parameters interact in ways that cannot be easily inspected. Always start with logistic regression as a baseline, and only move to more complex models if the performance improvement justifies the loss of interpretability.

3.6 Training: how the model learns the weights

How does the model find the right values for the weights $\theta_0, \theta_1, \ldots, \theta_n$? It does this by *optimizing* — finding the weights that make the model's predictions as accurate as possible on the training data. This optimization happens through two components: a cost function that measures how wrong the predictions are, and an algorithm (gradient descent) that iteratively adjusts the weights to reduce the cost.

3.6.1 The cost function

The cost function (also called the *loss function*) quantifies how far off the model's predictions are from the true labels. For logistic regression, the standard cost function is called **binary cross-entropy**:

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

This looks intimidating, but let us take it apart piece by piece.

- m is the number of training documents.
- $y^{(i)}$ is the true label for document i: either 0 or 1.
- $h^{(i)}$ is the model's predicted probability for document i: a number between 0 and 1 (the output of the sigmoid function).
- log is the natural logarithm.
- The negative sign at the front ensures that the overall cost is positive (since the logarithms of numbers between 0 and 1 are always negative).

The formula has two terms inside the sum, and for any given document only one of them is active:

When the true label is $y = 1$ (e.g., the speech is actually Labour): the $(1 - y^{(i)})$ factor becomes 0, so the second term vanishes. We are left with $-\log h^{(i)}$. If the model correctly predicts a high probability (say $h = 0.95$), then $-\log(0.95) \approx 0.05$ — a small cost, because the model got it right. But if the model wrongly

predicts a low probability (say $h = 0.05$), then $-\log(0.05) \approx 3.0$ — a large cost, because the model was confidently wrong.

When the true label is $y = 0$ (e.g., the speech is actually Conservative): the $y^{(i)}$ factor becomes 0, so the first term vanishes. We are left with $-\log(1 - h^{(i)})$. If the model correctly predicts a low probability (say $h = 0.05$, meaning it thinks this is probably class 0), then $-\log(0.95) \approx 0.05$ — small cost, correct prediction. But if the model wrongly predicts a high probability (say $h = 0.95$), then $-\log(0.05) \approx 3.0$ — large cost, wrong prediction.

The key insight is that the cost function penalizes *confidently wrong* predictions much more than *uncertain* ones. Getting something wrong with probability 0.55 (just barely wrong) costs much less than getting it wrong with probability 0.99 (very confidently wrong). This encourages the model to be well-calibrated — to assign high probabilities only when it is genuinely confident.

We average over all m training documents to get the overall cost. The goal of training is to find the weights θ that minimize this cost.

3.6.2 Gradient descent

How do we actually find the weights that minimize the cost? We use an algorithm called *gradient descent*. The intuition is like walking downhill in fog. You cannot see the bottom of the valley, but you can feel which direction slopes downward under your feet. If you always take a step in the downhill direction, you will eventually reach the bottom.

More precisely:

1. **Initialize** the weights to some starting values (often all zeros).
2. **Repeat until convergence:**
   a. Compute the current cost $J(\theta)$ on the training data.
   b. Compute the *gradient* — the direction of steepest increase in the cost. The gradient is a vector that points "uphill."
   c. Update the weights by taking a small step in the *opposite* direction (downhill): $\theta := \theta - \alpha \nabla J(\theta)$
3. **Stop** when the cost stops decreasing (convergence).

The parameter α is the *learning rate* — the size of each step. If α is too large, you overshoot the minimum and the algorithm bounces around. If α is too small, the algorithm converges very slowly. In practice, good libraries choose α adaptively.

You will not need to implement gradient descent yourself. Libraries like `glmnet` in R and `scikit-learn` in Python do this for you with highly optimized code. But understanding the concept helps you diagnose problems: if a model fails to converge, it might be because the learning rate is wrong or the features need to be rescaled.

3.6.3 Regularization: preventing memorization

With text data, you typically have thousands of features (one per word) but only hundreds or thousands of training documents. This means the model has more "knobs to turn" (weights) than there are data points to learn from. In this situation, the model can easily *overfit*: it assigns large weights to rare words that happen to co-occur with one class in the training data, but these patterns don't generalize to new data.

Regularization prevents overfitting by adding a penalty for large weights. The idea is: the model should not only minimize the cost, but also keep the weights small. Two common forms:

L2 regularization (Ridge): adds a penalty proportional to the *squared* weights: $J_{\text{reg}} = J(\theta) + \lambda \sum_j \theta_j^2$. This shrinks all weights toward zero but rarely sets any to exactly zero. It keeps all features but makes them smaller.

L1 regularization (Lasso): adds a penalty proportional to the *absolute value* of the weights: $J_{\text{reg}} = J(\theta) + \lambda \sum_j |\theta_j|$. This pushes many weights to exactly zero, effectively performing automatic feature selection. The resulting model uses only a subset of words — those most informative for the classification.

The parameter λ controls how much regularization to apply. Larger λ means more regularization (simpler model, lower risk of overfitting, but potentially worse fit to the data). In R's `glmnet`, this is tuned automatically using cross-validation: the package tries many values of λ and picks the one that gives the best performance on held-out data.

---

4 Part III — Naive Bayes for text

4.1 A different approach: thinking probabilistically

Logistic regression learns a direct mapping from features to class labels — it is a *discriminative* model. Naive Bayes takes a different, *generative* approach. Instead of asking "which class is most likely given these words?" directly, it asks: "If this document were written by a Labour MP, how likely would we be to see these specific words? And if it were written by a Conservative MP?" Then it uses Bayes' theorem to flip this around and get the answer we actually want.

The beauty of this approach is that it requires no iterative optimization, no gradient descent, no learning rate. It learns by *counting*. This makes it extremely fast and surprisingly effective, especially when training data is scarce.

4.2 Bayes' theorem: a refresher

Bayes' theorem is a fundamental result in probability theory that tells you how to update your beliefs when you see new evidence. The formula is:

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

In plain English: the probability of A given that you observed B equals the probability of B given A, times the prior probability of A, divided by the overall probability of B.

Let us apply this to text classification. Suppose we see a speech containing the word "austerity." We want to know: how likely is it that this speech was given by a Labour MP?

$$P(\text{Labour} \mid \text{“austerity”}) = \frac{P(\text{“austerity”} \mid \text{Labour}) \cdot P(\text{Labour})}{P(\text{“austerity”})}$$

- $P(\text{Labour} \mid \text{“austerity”})$ is what we want: the probability that the speaker is Labour, given that we see "austerity."
- $P(\text{“austerity”} \mid \text{Labour})$ is something we can estimate from data: how often do Labour MPs say "austerity"? We count.
- $P(\text{Labour})$ is the prior: what fraction of speeches in our training data are by Labour MPs? We count.
- $P(\text{“austerity”})$ is the overall frequency of "austerity." We count.

So far so good. But a real document contains many words, not just one. We need:

$$P(\text{Labour} \mid w_1, w_2, \ldots, w_n)$$

where $w_1, w_2, \ldots, w_n$ are all the words in the document.

4.3 The "naive" assumption

To compute $P(w_1, w_2, \ldots, w_n \mid \text{Labour})$ — the probability that a Labour MP would produce exactly this sequence of words — we would need to know the joint probability of all words appearing together. With a vocabulary of 10,000 words, this is practically impossible to estimate: there are more possible word combinations than there are speeches in any corpus.

The "naive" in Naive Bayes is the assumption that *words occur independently of each other*, given the class. Under this assumption:

$$P(w_1, w_2, \ldots, w_n \mid \text{Labour}) \approx P(w_1 \mid \text{Labour}) \times P(w_2 \mid \text{Labour}) \times \cdots \times P(w_n \mid \text{Labour})$$

This is clearly false in reality. Words are not independent: if a speech contains "climate," it is more likely to also contain "emissions" and "carbon." But this assumption makes the math tractable, and in practice Naive Bayes classifiers work surprisingly well despite the wrong assumption. The reasons are somewhat subtle, but the key insight is: for *classification* (deciding which class has the highest probability), you don't need the probabilities to be perfectly calibrated — you just need the ranking to be correct. The naive independence assumption often preserves the ranking even when it distorts the absolute probabilities.

4.4 How Naive Bayes classifies a document

For a binary classification problem (e.g., Labour vs. Conservative), we compare the posterior probabilities of the two classes and pick the one that is higher. Using Bayes' theorem and the independence assumption, the ratio of posteriors is:

$$\frac{P(\text{Labour} \mid \text{doc})}{P(\text{Conservative} \mid \text{doc})} = \underbrace{\frac{P(\text{Labour})}{P(\text{Conservative})}}_{\text{prior ratio}} \times \underbrace{\prod_{i=1}^{n} \frac{P(w_i \mid \text{Labour})}{P(w_i \mid \text{Conservative})}}_{\text{likelihood ratio for each word}}$$

The denominator $P(\text{doc})$ cancels because it is the same for both classes.

The **prior ratio** reflects how common each class is in the training data. If 60% of training speeches are Labour and 40% are Conservative, the prior ratio is $60/40 = 1.5$, which gives Labour a head start before we even look at the words.

The **likelihood ratio** for each word tells us how much that word shifts the prediction. If "austerity" is 3 times more frequent in Labour speeches than in Conservative speeches, then $P(\text{austerity} \mid \text{Labour})/P(\text{austerity} \mid \text{Conservative}) = 3$, which multiplies the Labour probability by 3 for this word alone.

We multiply together the likelihood ratios for all words in the document. If the overall product (times the prior ratio) is greater than 1, we predict Labour. If less than 1, we predict Conservative.

4.5 Estimating the probabilities: just counting

How do we get $P(w \mid \text{class})$? We simply count how often each word appears in each class in the training data:

$$P(w \mid \text{Labour}) = \frac{\text{count of } w \text{ in all Labour speeches}}{\text{total words in all Labour speeches}}$$

This is the maximum likelihood estimate, and it requires nothing more than counting.

4.6 The zero-frequency problem and Laplacian smoothing

There is one serious problem with the counting approach. What if a word appears in Labour speeches but *never* in Conservative speeches? Then:

$$P(\text{word} \mid \text{Conservative}) = \frac{0}{\text{total Conservative words}} = 0$$

Since we multiply all the word probabilities together, one zero wipes out the entire product:

$$\prod_i P(w_i \mid \text{Conservative}) = ... \times 0 \times ... = 0$$

This is catastrophic: a single unseen word makes the model assign zero probability to an entire class, regardless of all other evidence. A new document containing one word never seen in Conservative training data would *never* be classified as Conservative, even if every other word screams "Conservative."

The standard fix is **Laplacian smoothing** (also called "add-one" smoothing). We pretend that every word was seen at least once in every class:

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

where $|V|$ is the vocabulary size. The $+1$ in the numerator ensures no word has zero probability. The $+|V|$ in the denominator ensures the probabilities still sum to 1 across the vocabulary.

This is a small adjustment, but it prevents the zero-probability catastrophe entirely. Words that genuinely never appeared in a class will get a very small probability (instead of zero), which is a much better reflection of our actual uncertainty: we haven't observed this word in this class, but we cannot rule it out.

4.7 From products to sums: log-likelihood

There is a practical computational problem with multiplying many small probabilities together. If a document has 200 words, and each word has a probability of, say, 0.003 in the Labour class, then the product is approximately 0.003^{200}, which is around 10^{-500}. This number is far smaller than a computer can represent — it *underflows* to exactly zero. All classes get zero probability, and the classifier breaks.

The solution is to work with logarithms. The logarithm of a product is the sum of the logarithms:

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

Instead of multiplying hundreds of tiny numbers, we add hundreds of manageable log-values. No underflow.

The **decision rule** becomes: if this sum is greater than 0, predict Labour. If less than 0, predict Conservative. Each word contributes a log-likelihood ratio: positive if the word is more common in Labour, negative if more common in Conservative, near zero if equally common in both.

This also gives us interpretability similar to logistic regression: we can inspect the log-likelihood ratios to see which words are most diagnostic of each class.

4.8 Logistic regression vs. Naive Bayes: when to use which

Both models are strong baselines for text classification. Their strengths differ:

	Logistic Regression	Naive Bayes
Training speed	Moderate (iterative optimization)	Very fast (single-pass counting)
Small data	Needs regularization to avoid overfitting	Works well out of the box
Large data	Usually wins	Often competitive, sometimes worse
Handles correlated features	Yes, naturally	No (independence assumption is violated)
Probability calibration	Well-calibrated	Often poorly calibrated (overconfident)
Interpretability	Weights show word importance	Log-likelihood ratios show word importance

Practical recommendation: try both. If they agree, you can be confident in the result. If they disagree, investigate why — the disagreement itself is informative about your data. And always report logistic regression as a baseline, even if you ultimately use a more complex model.

---

5 Part IV — Evaluating classifiers

5.1 Why accuracy is not enough

Consider this scenario. You are building a classifier to detect hate speech in political tweets. Your dataset is 95% normal tweets and 5% hate speech. You build a "classifier" that always predicts "not hate speech," regardless of what the tweet says. Its accuracy is 95% — it correctly classifies 95% of all tweets. Sounds good, right?

It is completely useless. It catches zero hate speech. It has learned nothing. It just exploits the fact that most tweets are not hate speech.

This example illustrates why **accuracy alone is misleading** when classes are imbalanced — and in political science, classes are almost always imbalanced. Hate speech is rare. Populist rhetoric is rare. Crisis-related news is rare. Manifesto sentences about the EU are a small fraction of all sentences. For all these tasks, a naive "always predict the majority class" strategy gives high accuracy, and yet it fails at the only thing we actually care about: finding the minority class.

We need metrics that tell us *what kinds of errors* the model makes.

5.2 The confusion matrix

The confusion matrix is a 2×2 table that completely describes the model's predictions:

	Predicted: Positive	**Predicted: Negative**
Actual: Positive	True Positive (TP)	False Negative (FN)
Actual: Negative	False Positive (FP)	True Negative (TN)

- **True Positive (TP)**: the model correctly identified a positive example (correctly flagged hate speech).

- **False Positive (FP)**: the model incorrectly flagged a negative example as positive (false alarm — flagged a normal tweet as hate speech).
- **False Negative (FN)**: the model missed a positive example (missed actual hate speech).
- **True Negative (TN)**: the model correctly identified a negative example (correctly passed a normal tweet).

In political science, different errors have different costs. Missing hate speech (FN) can be dangerous. Flagging legitimate speech as hate speech (FP) raises free-speech concerns. For policy coding, miscoding in either direction (FP or FN) is equally problematic. Your choice of evaluation metric should reflect which errors are most costly for your specific application.

5.3 Precision, recall, and F1

Three metrics that are more informative than accuracy:

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

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

High precision means few false alarms. When the model says "this is hate speech," it is usually right.

Recall (also called *sensitivity*) answers: "Of all the documents that are *actually* positive, how many did the model find?"

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

High recall means few misses. The model catches most of the actual hate speech.

F1 score is the harmonic mean of precision and recall, giving a single number that balances both:

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

F1 = 1 is perfect. The harmonic mean has a useful property: if *either* precision or recall is near zero, F1 is also near zero. You cannot get a high F1 by being good at only one of the two.

A worked example. Suppose you have 200 test tweets: 180 normal, 20 hate speech. Your model predicts:

- TP = 12 (correctly flagged 12 hate speech tweets)
- FP = 5 (incorrectly flagged 5 normal tweets)
- FN = 8 (missed 8 hate speech tweets)
- TN = 175 (correctly passed 175 normal tweets)

Then:

- Accuracy = (12 + 175) / 200 = 93.5% — looks great
- Precision = 12 / (12 + 5) = 70.6% — when the model flags something, it's right 70.6% of the time
- Recall = 12 / (12 + 8) = 60.0% — but it misses 40% of actual hate speech
- F1 = 2 × 0.706 × 0.600 / (0.706 + 0.600) = 64.9%

The 93.5% accuracy masks the fact that we are missing 40% of hate speech. The F1 of 64.9% gives a much more honest picture of performance. Always report precision, recall, and F1.

5.4 The precision–recall trade-off

You can usually improve one at the cost of the other by adjusting the classification threshold. Remember, logistic regression outputs a probability between 0 and 1, and we classify as positive if the probability exceeds some threshold (default: 0.5).

Lowering the threshold (e.g., to 0.3): the model becomes more willing to predict positive. It catches more true positives (recall goes up), but also produces more false alarms (precision goes down).

Raising the threshold (e.g., to 0.7): the model becomes more conservative. Fewer false alarms (precision goes up), but it misses more true positives (recall goes down).

The right threshold depends on your application. For content moderation where missing hate speech is dangerous, you might accept lower precision to get higher recall. For legal applications where false accusations are costly, you might prefer higher precision and accept lower recall.

5.5 Dealing with class imbalance

When one class is much more common than the other (which is typical in political science), several strategies help:

1. **Stratified splitting**: ensure both classes are proportionally represented in train/validation/test sets.
2. **Class weighting**: tell the model to penalize errors on the minority class more heavily. In R's `glmnet`, set `weights` inversely proportional to class frequency. In `scikit-learn`, use `class_weight='balanced'`.
3. **Oversampling the minority class**: duplicate or synthetically generate minority-class examples so the model sees them more often.
4. **Evaluate with F1**, not accuracy, so you see the model's performance on the minority class.

---

6 Part V — Practical session

The practical session implements everything we have discussed: loading data, training both classifiers, evaluating them, and comparing their performance. See the separate R script (`Lecture2_Practical.R`) for the complete, commented code.

The workflow is:

1. Load the U.K. House of Commons corpus from Lecture 1
2. Define a binary classification task: Government vs. Opposition speeches
3. Build a DFM and split into training and test sets
4. Train Naive Bayes using `quanteda.textmodels`
5. Train regularized Logistic Regression using `glmnet`
6. Evaluate both using the confusion matrix, precision, recall, and F1
7. Inspect the most predictive words from the logistic regression weights
8. Compare: which model performs better? Which words drive predictions?

---

7 Key takeaways

1. **Supervised text classification** requires labeled data, a feature representation, and a classifier. The quality of your labels is the ceiling for your classifier's performance.

2. **Always split your data** into training and test sets. Never evaluate on training data. Use cross-validation when data is scarce.

3. **Logistic regression** is the most important baseline. It is fast, interpretable (you can read the weights), and surprisingly competitive with more complex methods.

4. **Naive Bayes** is a probabilistic alternative that learns by counting. It is faster than logistic regression and works well with small training sets, but assumes word independence.

5. **Evaluation must go beyond accuracy.** Use the confusion matrix, precision, recall, and F1. Accuracy is misleading with imbalanced classes — and almost all political science text classification tasks have imbalanced classes.

6. **Report multiple models.** If logistic regression and Naive Bayes agree, your findings are robust. If they disagree, the disagreement is informative.
