ECON 7377 Microeconometrics **Spring 2026**

Machine Learning

Contents

Part I

Basic Concepts in Machine Learning

1 Prediction and Projection

1.1 Conditional Expectation

- Supervised learning uses a function $g(x)$ to predict y. The prediction error $y - g(x)$ depends on the choice of g. Among all possible g, which one is the best?
- This question is agnostic about the data generating process (DGP). We seek a rule for accurate prediction of y given x, regardless of how (y, x) is generated.
- To compare different g, we need a *loss function* $L(y, g(x))$. A convenient choice is the *quadratic loss*:

$$L(y, g(x)) = (y - g(x))^2.$$

- Since the data are random, $L(y, g(x))$ is also random. To eliminate this uncertainty, we average the loss over the joint distribution of (y, x): $R(y, g(x)) = E[L(y, g(x))]$, called the *risk*. Risk is a deterministic quantity.
- For quadratic loss, the risk is

$$R(y, g(x)) = E\left[(y - g(x))^2\right],$$

  called the *mean squared error* (MSE). MSE is the most popular risk measure. An alternative is the *mean absolute error* (MAE) $E[|y - g(x)|]$. MSE is preferred for its closed-form tractability, which MAE lacks due to nondifferentiability.
- Our question becomes: What g minimizes the MSE?

Proposition 1. The CEF $m(x)$ minimizes MSE.

- Before proving this, we discuss properties of the conditional mean. We can write

$$y = m(x) + (y - m(x)) = m(x) + \epsilon,$$

  where $\epsilon := y - m(x)$ is the *regression error*. This decomposition holds for any joint distribution of (y, x), as long as $E[y|x]$ exists.

- The error term ϵ satisfies these properties:
  - $E[\epsilon|x] = E[y - m(x)|x] = E[y|x] - m(x) = 0$,
  - $E[\epsilon] = E[E[\epsilon|x]] = E[0] = 0$,
  - For any function $h(x)$, we have

$$E[h(x)\epsilon] = E[E[h(x)\epsilon|x]] = E[h(x)E[\epsilon|x]] = 0. \tag{1}$$

- The last property implies ϵ is uncorrelated with any function of x. In particular, setting $h(x) = x$ gives $E[x\epsilon] = \text{cov}(x, \epsilon) = 0$.

Proof of Proposition 1. We use "guess-and-verify." For arbitrary $g(x)$, decompose the MSE into three terms:

$$\begin{aligned} & E\left[(y - g(x))^2\right] \\ = \; & E\left[(y - m(x) + m(x) - g(x))^2\right] \\ = \; & E\left[(y - m(x))^2\right] + 2E\left[(y - m(x))(m(x) - g(x))\right] + E\left[(m(x) - g(x))^2\right]. \end{aligned}$$

The first term does not depend on $g(x)$. The second term equals

$$2E[\epsilon(m(x) - g(x))] = 0$$

by (1) with $h(x) = m(x) - g(x)$. The third term is minimized at $g(x) = m(x)$. □

- Unlike many econometric textbooks that assume $y = g(x) + \epsilon$ with $E[\epsilon|x] = 0$ for some unknown $g(\cdot)$, we take a predictive approach agnostic to the DGP. We observe y and x and seek $g(x)$ that predicts y as accurately as possible under MSE.

1.2 Linear Projection

- The CEF $m(x)$ minimizes the MSE, but $m(x) = E[y|x]$ depends on the joint distribution of (y, x), which is typically unknown.

- Suppose we restrict to linear functions $h(x; b) = x'b$ for $b \in \mathbb{R}^K$. The minimization problem becomes

$$\min_{b \in \mathbb{R}^K} E\left[(y - x'b)^2\right]. \tag{2}$$

- The first-order condition is

$$\frac{\partial}{\partial b} E\left[(y - x'b)^2\right] = E\left[\frac{\partial}{\partial b}(y - x'b)^2\right] = -2E\left[x\left(y - x'b\right)\right],$$

  where the first equality holds when $E\left[(y - x'b)^2\right] < \infty$ (so differentiation and expectation are interchangeable), and the second follows from the chain rule and linearity of expectation.

- Setting this to 0 yields the closed-form solution

$$\beta = \left(E\left[xx'\right]\right)^{-1} E\left[xy\right]$$

  provided $E\left[xx'\right]$ is invertible. Here b is an arbitrary K-vector, while β is the minimizer.

- The function $x'\beta$ is the *best linear projection* (BLP) of y on x, and β is the *linear projection coefficient*.

Remark 1. Linearity is less restrictive than it appears. It can capture nonlinear effects by re-defining x. For example, if

$$y = x_1\beta_1 + x_2\beta_2 + x_1^2\beta_3 + e,$$

then $\frac{\partial}{\partial x_1} m\left(x_1, x_2\right) = \beta_1 + 2x_1\beta_3$, which is nonlinear in x_1, while it is still linear in the parameter $\beta = (\beta_1, \beta_2, \beta_3)$ if we define a set of new regressors as $(\tilde{x}_1, \tilde{x}_2, \tilde{x}_3) = (x_1, x_2, x_1^2)$.

Remark 2. Even though in general $m\left(x\right) \neq x'\beta$, the linear form $x'\beta$ is still useful in approximating $m\left(x\right)$. That is, $\beta = \arg\min_{b \in \mathbb{R}^K} E\left[(m(x) - x'b)^2\right]$.

Proof. The first-order condition gives $\frac{\partial}{\partial b} E\left[(m(x) - x'b)^2\right] = -2E[x(m(x) - x'b)] = 0$. Rearrange the terms and obtain $E[x \cdot m(x)] = E[xx']b$. When $E[xx']$ is invertible, we solve

$$\left(E\left[xx'\right]\right)^{-1} E[x \cdot m(x)] = \left(E\left[xx'\right]\right)^{-1} E[E[xy|x]] = \left(E\left[xx'\right]\right)^{-1} E[xy] = \beta.$$

Thus β is also the best linear approximation to $m\left(x\right)$ under MSE. □

- We may rewrite the linear regression model, or the *linear projection model,* as

$$\begin{aligned} y &= x'\beta + e \\ E[xe] &= 0, \end{aligned}$$

  where $e = y - x'\beta$ is called the *linear projection error*, to be distinguished from $\epsilon = y - m(x)$.

2 Cross-Validation

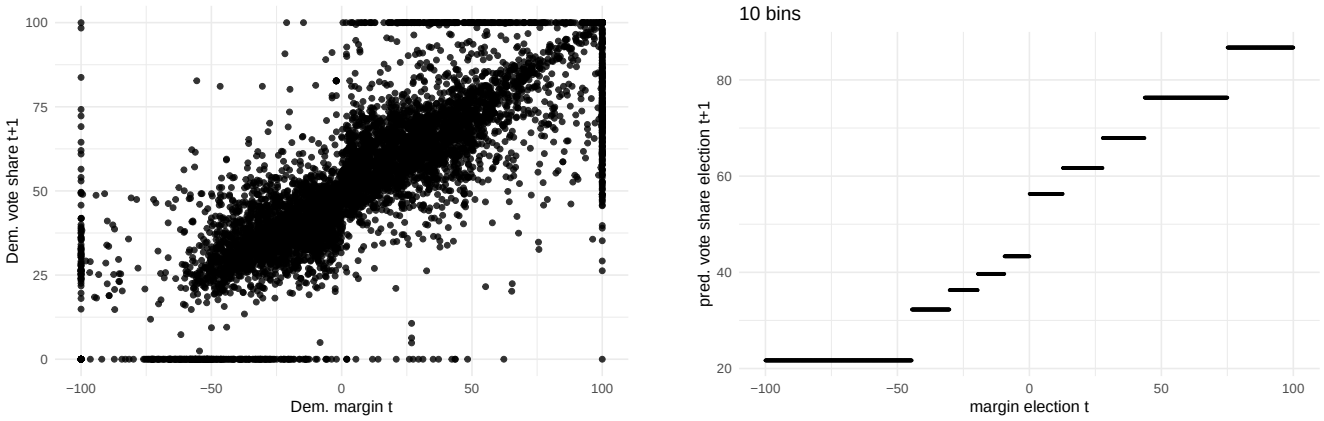
2.1 Overfitting

- For linear regression and logistic regression, we defined a generalization of R^2 based on the deviance. This is a measure of *in-sample fit*, how well the model fits your particular sample.
- However, the goal of prediction is to find a model that fits *new* (unseen) data well — *out-of-sample prediction*.
- If you have a set of candidate models, how do you pick the one that fits new data well without actually seeing that data?
- Should you pick based on R^2? No. Fitting past data too well usually makes the model fit poorly on new data. This is called *overfitting*, and it is one of the main things we try to avoid in machine learning (ML).
- Every model is split into a signal ($X_i'\beta$) and noise (ε_i) component, and when predicting, you just want the former (e.g. estimated using $X_i'\hat{\beta}$).
- Intuitively, overfitting occurs because you've fit the model too well to your particular sample such that it's being driven by the noise part. This part is poorly predictive of new data since ε_i is just a random mean-zero error.
- That can end up yielding worse predictions than no model at all (meaning just using the no-covariate prediction $\bar{Y}$).
- This is quite clear in the context of OLS and R^2. A well-known problem with R^2 is that it weakly increases if you add more covariates. You can get a perfect fit by having the same number of covariates as observations.

2.2 Example: Binned Sample Mean in Elections Data

- Let's examine this phenomenon with a simple estimator, the binned sample mean:
  - Place observations $(X_1, Y_1), \ldots (X_n, Y_n)$ into d bins $(t_0, t_1], (t_1, t_2], \ldots, (t_{d-1}, t_d]$, according to the value of X_i.
  - Bin j contains observations (X_i, Y_i) such that $t_{j-1} < X_i \leq t_j$.
  - The binned sample mean $\hat{f}(x)$ is the sample average of Y_i over observations with X_i in the same bin $(t_{j-1}, t_j]$ as x.

- Let's see this on the Lee (2008) elections data. X_i = Democratic margin of victory in election t and Y_i is Democratic vote share in election $t+1$.



(a) Raw data

(b) Binned sample mean (10 bins)

Figure 1: Lee Elections data: Democratic vote share in $t+1$ vs. margin of victory in t.

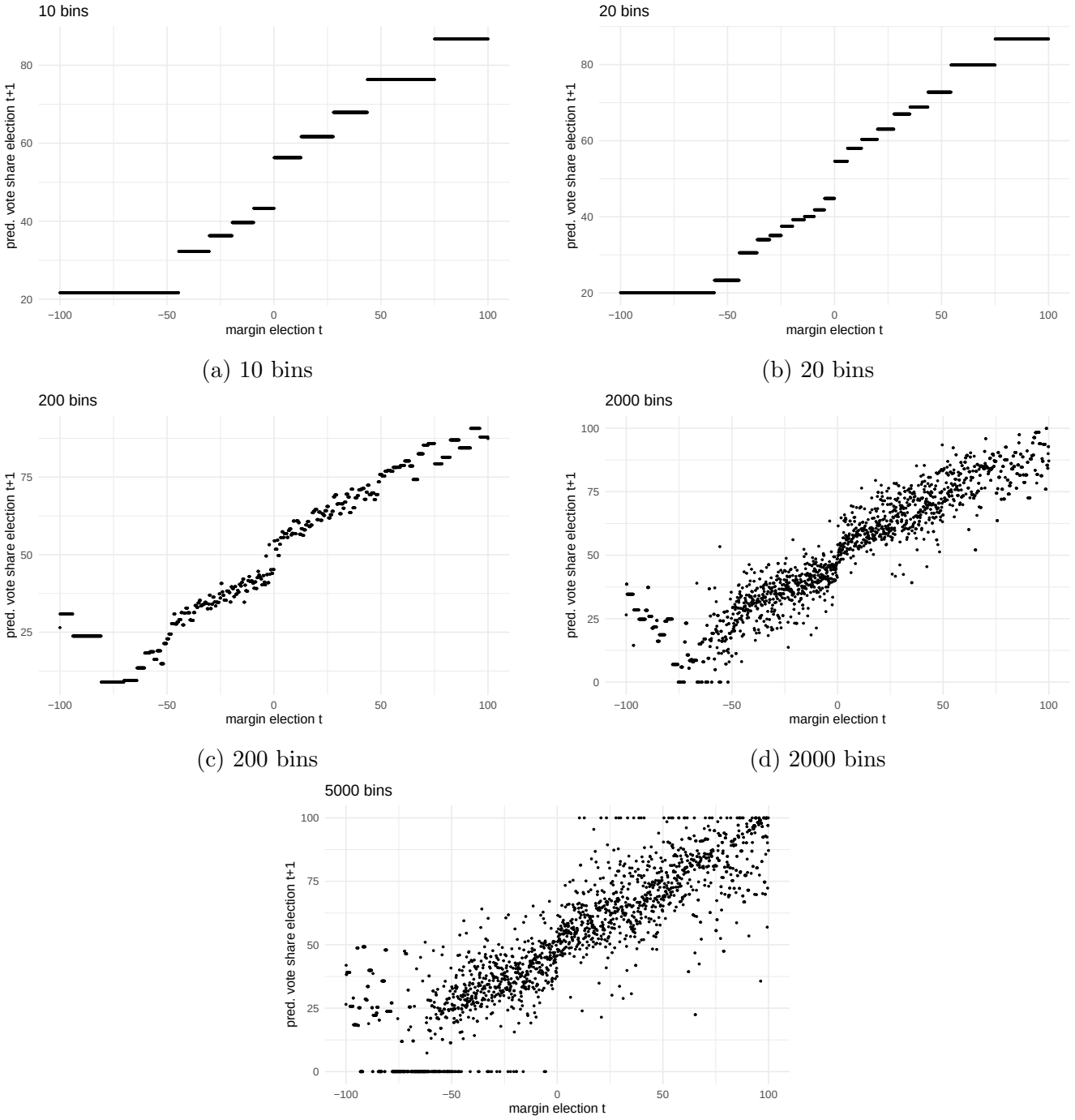
- With too many bins, we end up with one bin per data point, so that the estimates are just the raw Y_i's. Probably not the best estimate! See Figure 2.
- This is an example of *overfitting*: with too many bins, the model is too complex, and it just extrapolates the data.
- In OLS, model complexity corresponds to the number of covariates. Here, the number of bins plays the same role.
  - In fact, our binned estimator can be shown to be equivalent to a regression on indicator variables for each bin (try checking this!).
- What is the optimal choice of the number of bins? Let's examine this using *out of sample fit*.

2.3 Out-of-Sample Fit

- Recall that $R^2 = 1 - \text{residual deviance/null deviance}$, where deviance is

$$\text{dev}_{IS}(\hat{\beta}) = \sum_{i=1}^{n} (Y_i - X_i'\hat{\beta})^2$$

for OLS. The "IS" subscript stands for *in-sample*.



(a) 10 bins

(b) 20 bins

(c) 200 bins

(d) 2000 bins

(e) 5000 bins

Figure 2: Varying model complexity: as the number of bins increases, the binned sample mean overfits.

- Suppose we gather m new observations, labeled $n+1, \ldots, n+m$. We can then define *out-of-sample deviance*

$$\text{dev}_{OOS}(\hat{\beta}) = \sum_{i=n+1}^{n+m} (Y_i - X_i'\hat{\beta})^2.$$

- Note! $\hat{\beta}$ is only calculated using the past data $i = 1, \ldots, n$. If it instead used all $n+m$ observations, then we're back to in-sample deviance, just for a larger dataset.

- We can now define *out-of-sample* R^2:

$$R^2_{OOS} = 1 - \frac{\text{dev}_{OOS}(\hat{\beta})}{\text{dev}_{OOS}(\hat{\beta}_{\text{null}})},$$

  where $\hat{\beta}_{\text{null}}$ is estimate from the no-covariate model, i.e. $\text{dev}_{OOS}(\hat{\beta}_{\text{null}}) = \sum_{i=n+1}^{n+m}(Y_i - \bar{Y})^2$.

- This is a much better measure of predictive performance, since it's literally how well our model predictions fare on new data.

- How different can this be from in-sample R^2? To see this in our data:

  1. Take 2/3 of our original sample and use this for estimation. Compute in-sample R^2 on this sample.
  2. Use the remaining 1/3 of our original sample to compute out-of-sample R^2.

2.4 K-Fold Cross-Validation

- We just saw that we can get an idea of out-of-sample error by artificially splitting our sample and only using part of it for estimation. This is the idea behind K-fold cross validation.

- Hold out a random subsample of our data (the *test set* or *validation set*), estimating the model with the remaining data (the *training set*), and then computing $\text{dev}_{OOS} = \sum_{\text{test data}}(Y_i - \text{estimate}_i)^2$ with the test data.

- Problem: different random subsamples give you different values of dev_{OOS}. Solution: repeat this procedure K times, each with a different random subsample, to get a sample of K estimates of dev_{OOS}. Then we take the average.

- K-fold CV algorithm:

  1. Split the data into K evenly-sized subsets (*folds*) at random.
  2. For $k = 1, \ldots, K$:

(a) Estimate the model using all but the kth fold to get a predictor $\hat{f}_{-k}(x)$.

(b) Compute dev_{OOS} for this model using the kth fold as the test set. For squared error, this is

$$\text{dev}_{OOS,k} = \sum_{i \text{ in fold } k} (Y_i - \hat{f}_{-k}(X_i))^2$$

3. Take the average of these K estimates as our K-fold CV estimate of the out-of-sample deviance:

$$\text{dev}_{CV} = \frac{1}{K} \sum_{k=1}^{K} \text{dev}_{OOS,k}.$$

- Illustration of K-fold CV with $K = 5$:

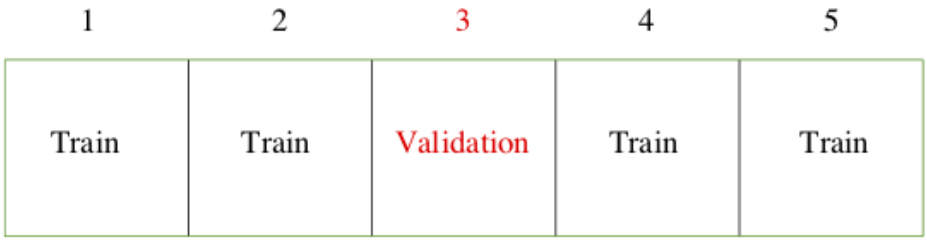
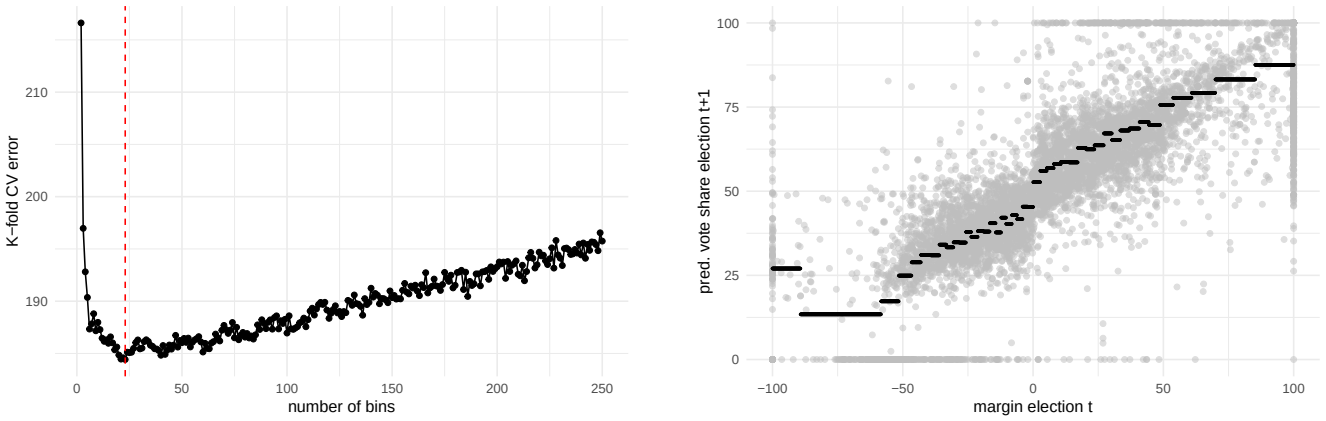


Figure 3: Source: *Elements of Statistical Learning*, Hastie et al. (2009).

- If we have multiple candidate models (e.g. binned sample mean estimators with different numbers of bins), we can use K-fold CV for *model selection*: we estimate out-of-sample error using K-fold CV for each model, and pick the one where our out-of-sample error estimate is smallest.

- Let's apply 10-fold CV to the elections data, searching over $1, \ldots, 250$ bins per side.



(a) K-fold CV error vs. number of bins

(b) CV-optimal binned estimate (46 bins)

Figure 4: K-fold cross-validation on the Lee Elections data. The red dashed line marks the CV-optimal choice.

- But is using cross-validation a good idea when estimating the cutoff point?

3 Bias-Variance Trade-off

- We'll discuss some of the theory behind why cross-validation works.
- This will involve introducing some population (rather than sample) objects: *population mean squared error* and its decomposition into *bias* and *variance*.
- We'll focus on the least squares case, but similar ideas apply to logit, etc.
- Understanding these concepts is also helpful for understanding the motivation behind *information criteria*, which we'll discuss next.

3.1 Population Mean-Squared Error

- In least-squares, we try to predict a continuous outcome Y using a linear function of the covariates $X'\beta$.
- More generally in ML, the goal is to predict Y using some (possibly nonlinear) function of the covariates, $f(X)$. Here we are positing the model

$$Y = f(X) + \varepsilon,$$

where ε is a mean-zero noise.

- Let $\hat{f}$ denote an estimator (e.g. $\hat{f}(x) = x'\hat{\beta}$ where $\hat{\beta}$ is a lasso estimator).
- We measure the predictive ability of $\hat{f}$ using the *population mean-squared error* (MSE)

$$\mathbb{E}[(Y_{n+1} - \hat{f}(X_{n+1}))^2]$$

  where
  - $\hat{f}$ is formed from the sample $(X_1, Y_1), \ldots, (X_n, Y_n)$
  - (X_{n+1}, Y_{n+1}) is a new observation drawn randomly from the same population
- Recall the out-of-sample error for least squares objectives was

$$\text{dev}_{OOS} = \sum_{i=n+1}^{n+m} (Y_i - \hat{f}(X_i))^2.$$

  Note that

$$\mathbb{E}\left[\frac{1}{m}\text{dev}_{OOS}\right] = \frac{1}{m}\sum_{i=n+1}^{n+m} \mathbb{E}\left[(Y_i - \hat{f}(X_i))^2\right] = \text{pop. MSE}.$$

- Thus, the population MSE tells us what the out-of-sample error will be "on average" over draws of the sample and a new observation from the same population.

3.2 MSE and CV

- The expectation of the average OOS deviance is exactly the population MSE.
- K-fold CV gives us K values of the OOS deviance:
  1. Split the data into K folds.
  2. For each $k = 1, \ldots, K$, estimate $\hat{f}$ on all but the kth fold and take the average of $(Y_i - \hat{f}_{-k}(X_i))^2$ over i in the kth fold.
- By averaging over these K values, we get an approximation of the population MSE, since averages approximate expectations by the LLN. This is the mathematical reason why CV works.
  - Another way of understanding this: the standard error bars on our CV graph are small when K is large enough.
  - Technical note: CV is actually approximating MSE for $\hat{f}$ computed with sample size $n \cdot (K-1)/K$.

- Rewriting the population MSE also gives us further insights into what makes a good predictive model.

3.3 Bias-Variance Decomposition

- First, note that population MSE takes the population average over two sources of randomness:
  - *estimation error* in how well $\hat{f}(x)$ approximates $f(x)$
  - *irreducible error* in how well $f(X_{n+1})$ predicts Y_{n+1}.
- Next, we'll decompose estimation error into *bias* and *variance*.
- First step:

$$\begin{aligned}\mathbb{E}\left[(Y_{n+1} - \hat{f}(X_{n+1}))^2\right] &= \mathbb{E}\left[(\varepsilon_{n+1} + f(X_{n+1}) - \hat{f}(X_{n+1}))^2\right] \\ &= \underbrace{\mathbb{E}(\varepsilon_{n+1}^2)}_{\text{irreducible error}} + \underbrace{\mathbb{E}\left[(\hat{f}(X_{n+1}) - f(X_{n+1}))^2\right]}_{\text{estimation error}}\end{aligned}$$

  since $\mathbb{E}[\varepsilon_{n+1}(\hat{f}(X_{n+1}) - f(X_{n+1}))] = 0$.
- Next, decompose estimation error. For technical reasons, we'll do this first for $\hat{f}(x) - f(x)$ with x fixed:

$$\begin{aligned}&\mathbb{E}\left[\left(\hat{f}(x) - f(x)\right)^2\right] \\ &= \mathbb{E}\left[\left(\hat{f}(x) - \mathbb{E}[\hat{f}(x)] + \mathbb{E}[\hat{f}(x)] - f(x)\right)^2\right] \\ &= \underbrace{\mathbb{E}\left[\left(\hat{f}(x) - \mathbb{E}[\hat{f}(x)]\right)^2\right]}_{\text{variance}} + \underbrace{\left(\mathbb{E}[\hat{f}(x)] - f(x)\right)^2}_{\text{bias}^2}\end{aligned} \tag{3}$$

  since $\mathbb{E}\left[\left(\hat{f}(x) - \mathbb{E}[\hat{f}(x)]\right)\left(\mathbb{E}[\hat{f}(x)] - f(x)\right)\right] = 0$.
- The estimation error $\mathbb{E}\left[(\hat{f}(X_{n+1}) - f(X_{n+1}))^2\right]$ averages this over the population distribution of X_{n+1}.
  - Formally, let $V(x)$ denote the variance term and $B(x)^2$ denote the squared bias term in the above display. Averaging over the distribution of X_{n+1} gives

$$\mathbb{E}\left[(\hat{f}(X_{n+1}) - f(X_{n+1}))^2\right] = \mathbb{E}[V(X_{n+1})] + \mathbb{E}[B(X_{n+1})^2].$$

3.4 Bias-Variance Tradeoff

- Can't control irreducible error, so focus on bias and variance.
- Generally, there is a *trade-off*:
  - More complex model (more predictors, more bins) $\Longrightarrow$ model can better approximate true $f(x) \Longrightarrow$ lower bias
  - But also: more complex model $\Longrightarrow$ more parameters to estimate $\Longrightarrow$ higher variance
- Penalization $\uparrow \Rightarrow$ model complexity $\downarrow \Rightarrow$ bias $\uparrow$, variance $\downarrow$
  - Extreme case: as $\lambda \to \infty$, $\hat{\beta} \to 0$, so we just guess that $Y_{n+1} = 0$ without looking at the data. No variance, but high bias (unless we're lucky enough that $f(x) = 0$)

3.5 Illustration with Binned Sample Mean

- To illustrate this in a simple setting, let's do some simulations with the binned sample mean.
- We generate $Y_i = \beta_1 + \beta_2 X_i + \varepsilon_i$ where $\beta_1 = 1$ and $\beta_2 = 1$.
  - We'll make all of the X_i's positive, so that we're not worrying about a discontinuity at zero.
- We then plot the binned sample mean estimate for different values of the number of bins.

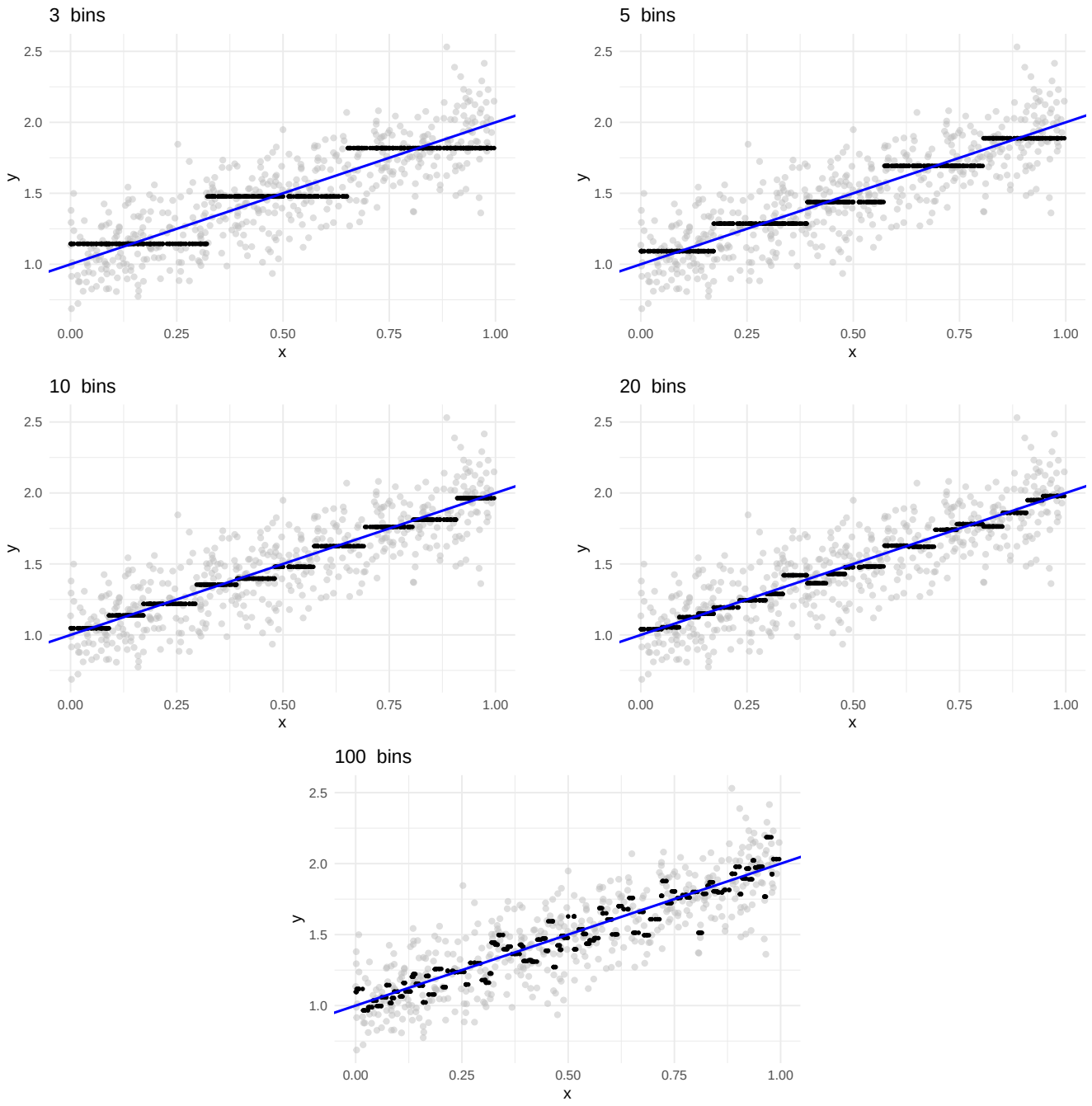


Figure 5: Bias-variance tradeoff illustration with binned sample mean. Blue line is the true conditional mean $E[Y|X] = 1 + X$.

- When the number of bins is small, the model is not very complex, and we have high bias and low variance.
  - The simple binning model forces us to overestimate (underestimate) the conditional mean at the left (right) end of a bin.

- When the number of bins is large, the model is more complex, and we have low bias and high variance.
  - Visually, we can see that the estimate "jumps around" the true conditional mean when the model is more complex.
- In this case, if we know a priori that the true model is linear, we can just impose this and estimate OLS on an intercept and the single coefficient X_i. Then the complexity is low (only two parameters) and there is no bias at all.
  - But, this is a risky strategy unless we have domain specific knowledge that tells us with great certainty that $f(x) = \mathbb{E}[Y_i|X_i]$ is linear. If we guess incorrectly that $\mathbb{E}[Y_i|X_i]$ is linear, then we will have high bias.
  - In practice, we usually can't get bias down to zero: there will typically be some reduction in bias each time we increase the complexity.

4 Information Criteria

4.1 MSE Estimation

- Cross-validation is one method of estimating MSE.
- It turns out that simpler estimators can be derived by adding a *degrees of freedom* (df) adjustment to the in-sample deviance.
- Basic idea:

$$\begin{aligned}\text{dev}_{IS} &= \sum_{i=1}^{n}(Y_i - \hat{f}(X_i))^2 = \sum_{i=1}^{n}(Y_i - f(X_i) + f(X_i) - \hat{f}(X_i))^2 \\ &= \sum_{i=1}^{n}(Y_i - f(X_i))^2 + \sum_{i=1}^{n}(\hat{f}(X_i) - f(X_i))^2 - 2\sum_{i=1}^{n}(Y_i - f(X_i))(\hat{f}(X_i) - f(X_i)). \quad (4)\end{aligned}$$

- Recall that overfitting makes dev_{IS} artificially small when $\hat{f}(X_i)$ depends too much on ε_i.
- This shows up in the last term: it tells us how much we're "cheating" by using Y_i in our in-sample prediction $\hat{f}(X_i)$.
  - CV gets rid of this term by using an estimate $\hat{f}$ computed from a different sample than Y_i.

  - In contrast, a degree of freedom adjustment estimates the last term and adds it back in.

- For least squares, we define degrees of freedom as

$$df = \text{ estimate of } \frac{1}{\sigma^2}\mathbb{E}\left[\sum_{i=1}^{n}(Y_i - f(X_i))(\hat{f}(X_i) - f(X_i))\right]$$

  where $\sigma^2 = \mathbb{E}[\varepsilon_i^2]$. We also use *df* for related quantities in likelihood settings.

- For MLE (OLS and logit), *df* = number of parameters. For lasso, *df* = number of nonzero $\hat{\beta}_j$'s (this is not obvious and follows from some deep theoretical derivations). For ridge, *df* is complicated.

  - Note that *df* from using `summary()` in R output actually gives $n - df$.

4.2 AIC

- For least squares objectives, adding *df* back leads to *Mallow's* C_p criterion:

$$\frac{1}{\hat{\sigma}^2}\sum_{i=1}^{n}(Y_i - \hat{f}(X_i))^2 + 2\,df$$

  where $\hat{\sigma}^2$ is computed with a low-bias model (e.g. including all regressors if possible).

- A generalization of C_p for likelihood models is *Akaike's information criterion* (AIC):

$$\text{AIC}(\hat{f}) = \text{dev}_{IS}(\hat{f}) + 2\,df$$

  (reduces to C_p for Gaussian likelihood with σ^2 treated as known).

- There is also a *corrected AICc*, which tries to account for estimation error in $\hat{\sigma}^2$:

$$\text{AICc}(\hat{f}) = \text{dev}_{IS}(\hat{f}) + 2\,df\frac{n}{n - df - 1}.$$

- Caution: different software packages (or different routines within the same software package) may make different normalizations when computing AIC, so be careful comparing across different packages/commands.

- AIC is just a substitute for CV with lower computational cost. We use it in the same way: compute $\text{AIC}(\lambda)$ for our regularization path $\lambda_1, \ldots, \lambda_T$, and use the choice of λ that

minimizes $\text{AIC}(\lambda)$.

- Note that the AIC penalty $2\,df$ and the regularization penalty $\lambda \sum_{j=1}^{d} c(\beta_j)$ are both "penalty terms," but they play different roles:
  - Regularization penalizes deviance and defines a new estimator for each λ.
  - AIC adds a penalty to deviance of a given estimator in order to measure its predictive performance and is used to select the optimal λ.

4.3 AIC vs CV

- AIC is basically a computational shortcut to CV.
- The theory of AIC requires some idealized assumptions, and it estimates a slightly different notion of error. But they are essentially trying to do the same thing.

4.4 Other Information Criteria

- **BIC (Bayesian Information Criterion).** Schwarz (1978) proposed

$$\text{BIC}(\hat{f}) = \text{dev}_{IS}(\hat{f}) + \log(n) \cdot df.$$

  BIC approximates -2 times the log marginal likelihood. Since $\log(n) > 2$ for $n \geq 8$, BIC penalizes complexity more heavily than AIC and favors simpler models.

- **Properties of BIC.**
  - *Consistent*: if the true model is among the candidates, BIC selects it with probability $\to 1$ as $n \to \infty$.
  - *Not asymptotically efficient*: BIC can underfit in finite samples, yielding higher prediction risk than AIC.

- **Hannan–Quinn (HQ) Criterion.** Hannan and Quinn (1979) proposed

$$\text{HQ}(\hat{f}) = \text{dev}_{IS}(\hat{f}) + 2\log(\log(n)) \cdot df.$$

  The penalty $2\log(\log(n))$ is the smallest rate that guarantees consistency. Since $2 < 2\log(\log(n)) < \log(n)$ for moderate n, HQ sits between AIC and BIC.

Part II

Regularization

1 Regularization

- Given a set of candidate models, we now know how to choose the best one in terms of predictive performance using K-fold CV. This is the standard way in ML.
- For the binned sample mean, the set of candidate models was manageable: basically the same as the number of observations, which was about 5,000.
- Even with univariate data, we may have many different possible models:
  - Cutting the bins in other ways (rather than same number of observations per bin). We'll see this with *regression trees* later.
  - All sorts of different ideas from approximation theory: polynomials, Fourier series, wavelets.
- Things become complicated quickly if we have many regressors. If we have d covariates, we can form a model with any of the 2^d possible subsets of these variables.
  - For $d = 20$, this is just over a million models to search over.
  - For $d = 100$, the number of models is greater than the number of all possible 16 digit passwords.
  - In typical high-dimensional data, d is in the thousands or higher.
- **High-dimensional** issue — the number of parameters p is large, compared to the sample size n: $p \geq n$ or $p \gg n$.
  - Let X be the covariate matrix, $\mathscr{L}_{\text{null}} \neq \{0\}$, which indicates the least squares solution is not unique. If $\hat{\beta} \in \arg\min \|y - X\beta\|^2$, then so is $\hat{\beta} + v$, where $v \in \mathscr{L}_{\text{null}}(X)$.
  - Even if $p < n$, OLS probably introduces high variance and overfitting problems.
  - MLE may not even exist when classes are perfectly separated for logistic regression.
  - When it is hard to interpret output, it motivates simpler, structured solutions.
- One way to get a manageable set of candidate models (and a fundamentally important idea in its own right) is *regularization*.

- Regularization modifies the deviance function by adding a penalty for model complexity. This gives a *regularized objective function*.
- A typical regularized objective function takes the form:

$$\text{dev}_{IS}(\beta) + \underbrace{\lambda \sum_{j=1}^{d} c(\beta_j)}_{\text{penalty}},$$

  where $c(\beta_j) \geq 0$ is a *cost* associated with the size of β_j and $\lambda \geq 0$ is a *penalty parameter* that controls how much to penalize model complexity.
- We as the researcher pick $c(\cdot)$ and λ and choose $\hat{\beta}$ to minimize the regularized objective instead of $\text{dev}_{IS}(\beta)$ alone.

1.1 The Role of the Regularization Penalty

- Obviously $\lambda = 0$ corresponds to our original estimator. If instead λ is very large, then we're effectively just minimizing $\sum_{j=1}^{d} c(\beta_j)$.
- Typically, we choose $c(\cdot)$ to be increasing in $|\beta_j|$, so that just minimizing the penalty gives $\hat{\beta} = 0$.
- Having $\hat{\beta}$ closer to zero corresponds to having a *less complex model* because in a GLM, your prediction is $\hat{\mathbb{E}}[Y \mid X] = G(X'\hat{\beta})$. This varies less across values of X if $\hat{\beta}$ is near zero.
- Thus, by adding the penalty to the deviance, we're telling the computer to fit the data by minimizing deviance but also to pick a less complex model.
  - Parsimony principle: If multiple models explain data about the same, we favor the simpler one.

Figure 6: Bias-variance tradeoff as a function of the penalty parameter λ.

1.2 Constrained Optimization Formulation

- Another way to see this: minimizing

$$\text{dev}_{IS}(\beta) + \lambda \sum_{j=1}^{d} c(\beta_j)$$

can be reformulated as a constrained minimization problem:

$$\min_{\beta} \text{dev}_{IS}(\beta) \text{ subject to } \sum_{j=1}^{d} c(\beta_j) \leq s$$

- s plays the opposite role of λ. If s is large, the constraint doesn't matter because you're allowing for a very large cost. If $s = 0$, it must pick the least complex model possible.
- Usually, people pick $c(\beta_j)$ such that (1) its minimum value is zero and (2) it is increasing in $|\beta_j|$. This ensures that the lowest cost model is the least complex model possible.
- Perhaps the most popular penalties are *Lasso*($c(\beta_j) = |\beta_j|$), and *ridge*($c(\beta_j) = \beta_j^2$).

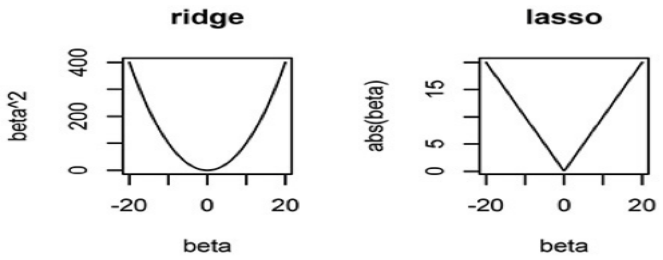


Figure 7: Lasso and ridge penalty functions.

2 Ridge Regression

- The optimization problem is:

$$\min_{\beta \in \mathbb{R}^p} \left\{ \|y - X\beta\|^2 + \lambda \|\beta\|^2 \right\}$$

- **Optimality Conditions:**

$$-2X^T(y - X\hat{\beta}_\lambda) + 2\lambda\hat{\beta}_\lambda = 0$$

gives the closed-form solution of the ridge estimator:

$$\hat{\beta}_\lambda = (X^T X + \lambda I_p)^{-1} X^T y.$$

All eigenvalues of $X^T X + \lambda I_p$ are > 0, so it is always invertible.

- **Ridge-less regression**. Gaillac and L'Hour (2025, Lemma 2.1):

$$\lim_{\lambda \downarrow 0} \hat{\beta}_\lambda = \arg\min\{\|\beta\|_2 \quad \text{s.t. } y = X\beta\} = (X^T X)^+ X^T y = X^+ y.$$

- This problem always has a *unique minimizer regardless of* X.

2.1 Bias–variance tradeoff

- If $y \sim N_n(X\beta, \sigma^2 I_n)$,

$$\mathbb{E}\hat{\beta}_\lambda = (X^T X + \lambda I)^{-1} X^T \mathbb{E} y$$

When we substitute $\mathbb{E}y = X\beta$, the estimator is *biased* for $\lambda > 0$.

$$\mathrm{Cov}(\hat{\beta}_\lambda) = \sigma^2 (X^T X + \lambda I)^{-1} X^T X (X^T X + \lambda I)^{-1}$$

So the covariance is *smaller* than OLS: $\sigma^2 (X^T X)^{-1}$.

$$\begin{aligned} \mathrm{MSE} &= \mathbb{E}\|\hat{\beta}_\lambda - \beta\|^2 \\ &= \mathbb{E}\|\hat{\beta}_\lambda - \mathbb{E}\hat{\beta}_\lambda\|^2 + \|\mathbb{E}\hat{\beta}_\lambda - \beta\|^2 \\ &= \sum_{j=1}^{p} \mathrm{Var}(\hat{\beta}_{\lambda_j}) + \mathrm{bias}^2 \end{aligned}$$

where $\hat{\beta}_\lambda - \mathbb{E}\hat{\beta}_\lambda$ has mean 0, and $\mathbb{E}\hat{\beta}_\lambda - \beta$ is a constant.

- **Simplified Case:** If $X^T X = I_p$ (orthonormal design), then $\hat{\beta}_\lambda = \frac{1}{1+\lambda} X^T y$.

$$\mathbb{E}[\hat{\beta}_\lambda] = \frac{1}{1+\lambda}\beta, \qquad \mathrm{Cov}(\hat{\beta}_\lambda) = \frac{\sigma^2}{(1+\lambda)^2} I_p$$

$$\mathrm{MSE}(\lambda) = \frac{\sigma^2 p + \lambda^2 \|\beta\|^2}{(1+\lambda)^2}$$

The minimum-MSE choice of λ satisfies:

$$\frac{\partial \text{MSE}(\hat{\lambda})}{\partial \lambda} = 0 \quad \Rightarrow \quad \hat{\lambda} = \frac{\sigma^2 p}{\|\beta\|^2} = \frac{1}{\text{SNR}}$$

- High SNR $\Longrightarrow \lambda \approx 0$ (almost OLS)
- Low SNR $\Longrightarrow \lambda$ is large (more shrinkage)

3 Lasso

- Arguably the most successful statistical learning method; Cited more than 70,000 times since Tibshirani (1996)
- The Lasso optimization problem is:

$$\min_{\beta \in \mathbb{R}^p} \left\{ \|y - X\beta\|^2 + \lambda \|\beta\|_1 \right\}$$

- Contours of constant value of $|\beta_1|^q + |\beta_2|^q$ for some values of q:

3.1 Selection Effects

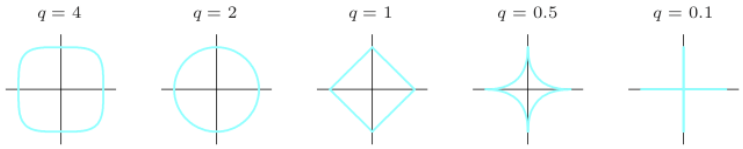


Figure 8: Source: *Elements of Statistical Learning*, Hastie et al. (2009).

- $q < 1$ also "spikey," but nonconvex.
- Lasso is most popular because its "spikiness" at zero means that the solution $\hat{\beta}$ often sets some of the $\hat{\beta}_j$'s to *exactly* zero. This more strongly reduces model complexity by selecting strictly fewer covariates.
- Both lasso and ridge shrink $\hat{\beta}_j$'s towards zero relative to OLS, but ridge solutions usually will not have any $\hat{\beta}_j$'s exactly zero.

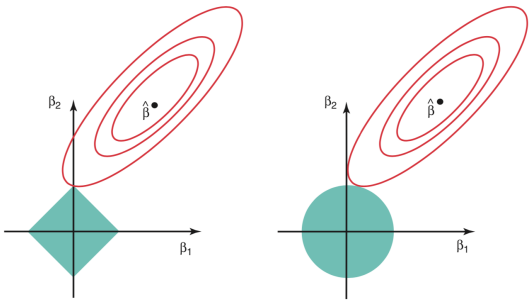


Figure 9: Depiction of lasso and ridge minimization problems (James et al., 2021)

- Blue areas are constraint regions, the set of $\hat{\beta}$'s satisfying $c(\hat{\beta}_1) + c(\hat{\beta}_2) \leq s$.
- Red ellipses are contours of the deviance function. The center of the ellipses is $\hat{\beta}$, the (unconstrained) OLS solution.
- The Lasso and ridge solutions are the intersection of a contour and the constraint region.
- Note the spikiness of the lasso constraint region in contrast to ridge. Contours of the deviance will often end up intersecting the region at one of the spikes; at each spike, the solution zeros out one of the $\hat{\beta}_j$'s, thereby selecting a smaller model.
- By zeroing out some coefficients, lasso produces simpler and potentially more interpretable models.
- When the coefficients of $\hat{\beta} = (\hat{\beta}_1, \ldots, \hat{\beta}_d)'$ are mostly zero, we say that it is *sparse*. A large body of theoretical work shows that lasso performs well when the true coefficient vector is sparse.

3.2 Optimality Conditions

3.2.1 KKT Conditions

- Recall that LASSO solves the following regularized least square problem,

$$\min f(\beta)$$

where

$$f(\beta) = \frac{1}{2}\|y - X\beta\|^2 + \lambda\|\beta\|_1.$$

- Note that f is non-differentiable, we cannot drive the usual first order conditions $\nabla_\beta f(\hat{\beta}) = 0$

and the closed-form solution for $\hat{\beta}$. Instead, define the subdifferential of f at $\tilde{\beta}$ as

$$\partial f(\tilde{\beta}) = \left\{ g \in \mathbb{R}^p : f(\beta) \geq f(\tilde{\beta}) + g'(\tilde{\beta} - \beta) \right\}$$

and immediately from the definition we have $0 \in \partial f(\hat{\beta})$ iff $\hat{\beta} \in \arg\min_\beta f(\beta)$.

- *Subgradients of ℓ_1 penalty*:

$$\partial |\beta| = \begin{cases} \operatorname{sgn}(\beta) & \text{if } \beta_j \neq 0 \\ [-1, 1] & \text{if } \beta_j = 0 \end{cases}$$

which is illustrated in Figure 10. Similarly,

$$s \in \partial \|\beta\|_1 \Leftrightarrow s_j \begin{cases} = \operatorname{sgn}(\beta_j) & \text{if } \beta_j \neq 0 \\ \in [-1, 1] & \text{if } \beta_j = 0 \end{cases}.$$

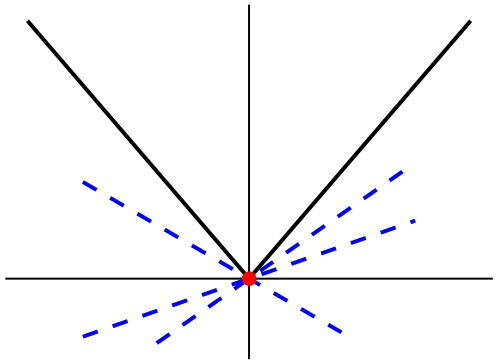


Figure 10: Subdifferential of the absolute value function.

- We can derive the optimality conditions (Karush-Kuhn-Tucker (KKT) conditions) of LASSO.

$$0 \in \partial f(\hat{\beta}_\lambda) = -X^\top (y - X\hat{\beta}_\lambda) + \lambda \partial \|\hat{\beta}_\lambda\|_1 \implies X^\top (y - X\hat{\beta}_\lambda) \in \lambda \partial \|\hat{\beta}_\lambda\|_1,$$

which implies

$$X^\top (y - X\hat{\beta}_\lambda) = \lambda s, \tag{5}$$

where $s \in \partial \|\hat{\beta}_\lambda\|_1$. Then

$$\|X^\top (y - X\hat{\beta}_\lambda)\|_\infty \leq \lambda, \tag{6}$$

and for $\hat{\beta}_{\lambda,j} \neq 0$,

$$X_j^\top (y - X\hat{\beta}_\lambda) = \lambda \operatorname{sgn}(\hat{\beta}_{\lambda,j}). \tag{7}$$

3.2.2 Lasso Regularization Path

- Consider the lasso cost function. Each value of λ yields a different lasso estimator. Which value do we choose?
- For $\lambda_1, \ldots, \lambda_T$, let $\hat{\beta}^1, \ldots, \hat{\beta}^T$ be the associated lasso estimators. We can pick the best $\hat{\beta}^t$ from this set using cross-validation.
- A common method for generating a sequence of candidate λ's is as follows.
  1. Begin with λ_1 such that $\hat{\beta}^1 = 0$.
  2. For $t = 1, \ldots, T$, set $\lambda_t = \delta\lambda_{t-1}$ for some $\delta \in (0, 1)$, and compute $\hat{\beta}^t$ under penalty λ_t.
- The resulting $\hat{\beta}^1, \ldots, \hat{\beta}^T$ is called the *lasso regularization path*. The idea is we start with a very large λ_1 that gives us the least complex model and then iteratively shrink it to get increasingly complex models.
- This is done automatically by `gamlr` and `glmnet`.
- Recall the first order condition of the least square estimation, $X^\top(y - X\hat{\beta}_{\text{ols}}) = 0$, which is referred as the *normal equation*. When $\lambda = 0$, we recover the normal equation from the optimality conditions.
- For $\lambda > 0$, we have
$$\left|X_j^\top(y - X\hat{\beta}_\lambda)\right| < \lambda \implies \hat{\beta}_{\lambda,j} = 0.$$
- From the KKT condition (5), $\hat{\beta}_\lambda$ is **piece-wise linear** in λ.

3.2.3 LARS Algorithm

- If features are standardized, i.e. $\sum_i X_{ij} = 0$ and $\frac{1}{n-1}\sum_i X_{ij}^2 = 1$, then $X_j^\top(y - X\hat{\beta}_\lambda)$ is simply the correlation between X_j and the residual. We can interpret the λ as a tuning parameter to select variables based on the correlation between the variable X_j and the residual.
- Efron et al. (2004)'s LARS algorithm is based on thinking about how these correlations change as λ is varied starting with $\lambda_{\max} = \|X^\top y\|_\infty$, which is the smallest λ for which $\hat{\beta}_\lambda \equiv 0$. As we decrease λ, covariates enter the model from the highest correlation to the least.
- Similar to simple forward selection, LARS algorithm is a greedy algorithm that adds one variable at a time, but it only enters "as much" of a variable as it deserves (in terms of the correlation).

Algorithm: Least Angle Regression (LAR)

1. Standardize the predictors to have mean zero and unit norm. Start with the residual $r = y - \bar{y}$ and $\beta \equiv 0$.

2. Find the predictor X_j most correlated with r.

3. Move β_j in the direction of $X_j^\top r$ until some other predictor X_k has as much correlation with the current residual as does X_j.

4. Move β_j and β_k in the direction defined by their joint least squares coefficient of the current residual on (X_j, X_k), until some other predictor X_l has as much correlation with the current residual.

5. Continue until all predictors have entered after $\min\{N-1, p\}$.

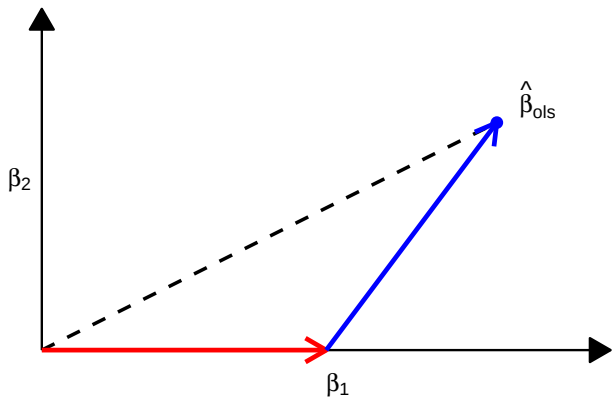


Figure 11: Illustration of the LARS algorithm with two covariates.

- By construction the coefficients in LARS are piecewise linear. The step length can be computed analytically (Ex. 3.25 in Hastie et al. (2009)). The LARS algorithm is efficient and can be used to compute the entire regularization path.

- **Lasso Modification.** LARS algorithm can be modified to solve the Lasso problem. Note by (7), the correlation between the current residual and an active covariate must have the same sign as the coefficient of the covariate. We should remove a covariate from the active set if its coefficient touches 0 along the path to avoid the sign inconsistency to obtain the Lasso solution. Based on this observation, we modify the LARS algorithm by:

- **4a**. If a non-zero coefficient hits zero, drop its variable from the active set of variables and recompute the current joint least squares direction.

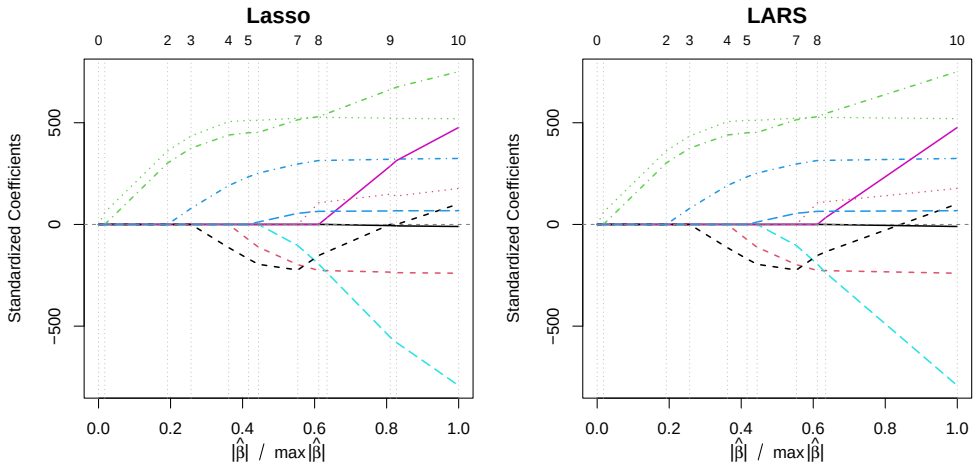


Figure 12: Solution paths on the diabetes data (`lars` package). Left: LARS-Lasso (note the coefficient that re-enters zero around step 10). Right: LARS (coefficients are piecewise linear and never revisit zero).

3.2.4 Uniqueness of the Lasso Estimator

c.f. Tibshirani (2013) for details.

- Lasso solution is *NOT* guaranteed to be unique (e.g. $X_j = X_k$ for some $j \neq k$), however, $X\hat{\beta}_\lambda$ is unique since

$$\min_{\beta,u} \left\{ \frac{1}{2}\|y-u\|^2 + \lambda \|\beta\|_1 \text{ s.t. } u = X\beta \right\}$$

$$= \underbrace{\min_{\beta,u} \left\{ \frac{1}{2}\|y-u\|^2 + \lambda \underbrace{\min_{u} \{\|\beta\|_1 \text{ s.t. } u = X\beta\}}_{\text{convex function in } u} \right\}}_{\text{strongly convex objective function in } u}.$$

- So, $\hat{u} = X\hat{\beta}_\lambda$ is unique even when $\hat{\beta}_\lambda$ is not.
- $\|\hat{\beta}_\lambda\|_1$ is also unique since the minimized objective value $\frac{1}{2}\|y-\hat{u}\|^2 + \lambda\|\hat{\beta}_\lambda\|_1$ and $\hat{u}$ are unique.
- $\hat{s}$ is also unique since $\hat{s}$ is defined in terms of $X^\top(y-\hat{u})$ where $\hat{u}$ is unique.

- If $\hat{\beta}_\lambda$ and $\tilde{\beta}_\lambda$ are both LASSO solutions, then

$$\hat{\beta}_{\lambda,j} > 0 \implies \hat{s}_j = 1 \implies \tilde{s}_j = 1 \implies \tilde{\beta}_{\lambda,j} \geq 0.$$

- The estimated active set $\mathcal{A}_\lambda = \{j : \hat{\beta}_{\lambda,j} \neq 0\}$ is not necessarily unique. We define the *equicorrelation set*,

$$\mathcal{E}_\lambda = \left\{ j : |X_j^\top (y - X\hat{\beta}_\lambda)| = \lambda \right\}.$$

- Any active set is a subset of the equicorrelation set: $\mathcal{A}_\lambda \subseteq \mathcal{E}_\lambda$, where $\mathcal{E}_\lambda$ is the largest active set (and it's produced by LARS).

3.3 Computational Considerations

- The lasso and ridge penalties are *convex*. This turns out to make computation much easier.
- Lasso packages such as `gamlr` and `glmnet` have built-in optimization routines.
- If you're interested in the details of convex optimization, a good resource is Boyd and Vandenberghe's book (free online https://web.stanford.edu/~boyd/cvxbook/).
- Read the lecture notes optimization. In that note, Lasso works as a running example to illustrate various numerical optimization algorithms.
- For further details, see also Gao and Shi (2021) for a `Rmosek` implementation.
- We could impose sparsity directly with the cost function $c(\beta_j) = \mathbb{1}\{\beta_j \neq 0\}$, but this would make computation much more difficult: we'd basically be back to checking 2^d models.
- Not all ML estimators are computationally "nice:" deep neural networks are difficult/impossible to compute, but have become popular recently due to (a) massive parallelization and (b) evidence that they work well with ad hoc optimization methods that don't exactly find the global minimum.

3.4 Choosing λ

- A regularization path gives you a set of possible estimators, one for each penalty parameter.
- To choose the best one, we can use cross-validation, just as we saw previously for choosing bin size with the binned sample mean estimator.
- Algorithm:

1. Get a sequence of candidate λ's by estimating a regularization path $\lambda_1, \ldots, \lambda_T$ as before.
2. Split the data randomly into K folds.
3. For each $k = 1, \ldots, K$, estimate $\hat{\beta}^t$ for each λ_t on the training set (all data except kth fold), and compute the deviance $\text{dev}_{OOS}(\hat{\beta}^t)$ on the test set (kth fold).
4. This results in K estimates of the deviance for each candidate λ_t. Take the average deviance over these K and choose the λ_t with the lowest deviance as your $\hat{\lambda}$.
5. Rerun lasso setting the penalty equal to $\hat{\lambda}$ to get your final $\hat{\beta}$.

- All of this is automated in the function `cv.glmnet()` in the `glmnet` package.
- Theoretical discussion on cross-validated Lasso: Chetverikov et al. (2021); Chetverikov (2024).

3.5 Choosing Number of Folds K

- In `glmnet`, the default is $K = 10$ folds (you can change this with the `nfolds` argument of `cv.glmnet`).
- Larger K is more computationally expensive but also gives you a better estimate of OOS deviance.
- Let $SD_{OOS}(\hat{\beta}^t)$ be the sample standard deviation of the deviances for λ_t across the K folds. The "standard error" for the estimated OOS deviance is then $SD_{OOS}(\hat{\beta}^t)/\sqrt{K}$.
- Let $\hat{\text{dev}}_{OOS}(\hat{\beta}^t)$ be the CV average deviance for λ_t. Then

$$\hat{\text{dev}}_{OOS}(\hat{\beta}^t) \pm \frac{SD_{OOS}(\hat{\beta}^t)}{\sqrt{K}}$$

  is a "confidence interval" for a population OOS deviance.
- Standard errors are also sometimes used to choose λ:
  - Basic idea: any choice of λ that gets us within, say, 1 standard error of the minimum, should also be fine.
  - We might prefer larger λ just because it gives us fewer nonzero coefficients.
  - The "one standard error" choice is the largest λ such that the CV error is within one standard error of the minimum. This is returned as `lambda.1se` in `cv.glmnet`.

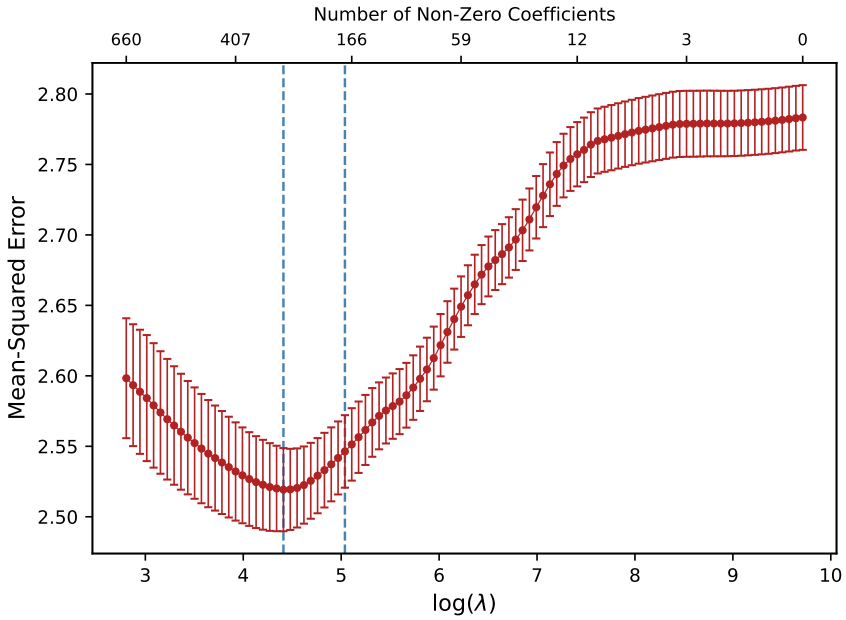


Figure 13: 10-fold cross-validation for λ selection (online spending data). Each point is the CV mean-squared error at a given λ; error bars show ± 1 standard error across folds. The left dashed line marks `lambda.min`; the right marks `lambda.1se`.

3.6 Standardization

- With OLS, the scale of the covariates doesn't matter for estimation.
- But with lasso, all coefficients are penalized by the same λ, so covariates on different scales are penalized differently.
- In order to penalize all covariates the same, we *standardize* the penalty, changing it to $\lambda \sum_{k=1}^{d} \mathrm{sd}(X_k)|\beta_k|$.
- In `glmnet()`, standardization of covariates is done by default (you can turn this off with `standardize=FALSE`).

3.7 Brief Introduction to Lasso Theory

3.7.1 KKT Conditions

Because the ℓ_1 norm is not differentiable at zero, we write first-order optimality using **subgradients**. Consider the lasso problem

$$\min_{\beta \in \mathbb{R}^p} \; f(\beta) := \frac{1}{2n}\|Y - X\beta\|_2^2 + \lambda\|\beta\|_1.$$

Let $\widehat{\beta}$ be a solution and define residuals $\widehat{e} := Y - X\widehat{\beta}$. The KKT condition is

$$0 \in \partial f(\widehat{\beta}) \quad \Longleftrightarrow \quad \frac{1}{n} X'\widehat{e} = \lambda s,$$

where $s \in \partial \|\widehat{\beta}\|_1$. Component-wise,

$$s_j = \begin{cases} \text{sign}(\widehat{\beta}_j) & \text{if } \widehat{\beta}_j \neq 0, \\ u, \quad u \in [-1, 1] & \text{if } \widehat{\beta}_j = 0. \end{cases}$$

Lasso sets $\widehat{\beta}_j = 0$ whenever the (absolute) sample covariance $|\frac{1}{n} x_j'\widehat{e}|$ is not large enough to overcome the threshold λ.

3.7.2 Restricted Eigenvalue Condition

When p is large (possibly $p > n$), $X'X$ is singular so the usual smallest-eigenvalue condition cannot hold. Lasso theory instead uses eigenvalue conditions that only need to hold on **sparse directions**.

Let the (sample) Gram matrix be $\widehat{\Sigma} := \frac{1}{n} X'X$. For an index set $S \subset \{1, \ldots, p\}$ and a constant $c_0 \geq 1$, define the cone

$$\mathcal{C}(S, c_0) := \{\delta \in \mathbb{R}^p : \ \|\delta_{S^c}\|_1 \leq c_0 \|\delta_S\|_1\}.$$

We say X satisfies the **restricted eigenvalue condition** with sparsity level s and constant $\kappa(s, c_0) > 0$ if for all sets S with $|S| \leq s$ and all $\delta \in \mathcal{C}(S, c_0) \setminus \{0\}$,

$$\delta' \widehat{\Sigma}\, \delta \ \geq \ \kappa(s, c_0)^2 \, \|\delta_S\|_2^2.$$

The RE condition prevents strong collinearity *among sparse combinations of regressors*. It is a standard assumption used to derive lasso error bounds.

3.7.3 Consistency of Lasso

Step 1: Basic Inequality

Optimality implies the **basic inequality**

$$\frac{1}{2n} \|Y - X\widehat{\beta}\|_2^2 + \lambda \|\widehat{\beta}\|_1 \leq \frac{1}{2n} \|Y - X\beta_0\|_2^2 + \lambda \|\beta_0\|_1.$$

Substitute $Y = X\beta_0 + e$ and expand to obtain

$$\frac{1}{2n} \|X\Delta\|_2^2 \leq \frac{1}{n} e' X\Delta + \lambda(\|\beta_0\|_1 - \|\widehat{\beta}\|_1),$$

where $\Delta := \widehat{\beta} - \beta_0$. By Hölder's inequality:

$$\frac{1}{n} e' X \Delta \leq \left\| \frac{1}{n} X' e \right\|_\infty \|\Delta\|_1.$$

Choose λ so that $\lambda \geq 2 \left\| \frac{1}{n} X' e \right\|_\infty$. Then

$$\frac{1}{2n} \|X\Delta\|_2^2 \leq \frac{3\lambda}{2} \|\Delta_S\|_1 - \frac{\lambda}{2} \|\Delta_{S^c}\|_1.$$

Since the left-hand side is nonnegative, this implies the **cone condition**: $\|\Delta_{S^c}\|_1 \leq 3\|\Delta_S\|_1$, so $\Delta \in \mathcal{C}(S, 3)$.

Step 2: Apply the RE Condition

Under the RE condition with $c_0 = 3$, we have

$$\frac{1}{n} \|X\Delta\|_2^2 = \Delta' \widehat{\Sigma}\, \Delta \geq \kappa(s, 3)^2 \|\Delta_S\|_2^2.$$

Combine with Step 1 and the Cauchy-Schwarz inequality $\|\Delta_S\|_1 \leq \sqrt{s}\|\Delta_S\|_2$:

$$\kappa(s, 3)^2 \|\Delta_S\|_2^2 \leq 3\lambda\sqrt{s} \|\Delta_S\|_2.$$

Hence $\|\Delta_S\|_2 \leq \frac{3\lambda\sqrt{s}}{\kappa(s,3)^2}$ and

$$\frac{1}{n} \|X\Delta\|_2^2 \leq 9 \frac{\lambda^2 s}{\kappa(s, 3)^2}, \qquad \|\Delta\|_1 \leq 12 \frac{\lambda s}{\kappa(s, 3)^2}.$$

Step 3: Consistency

Under standard tail conditions (e.g., sub-Gaussian errors and normalized columns), one can show

$$\left\| \frac{1}{n} X' e \right\|_\infty = O_p\left(\sqrt{\frac{\log p}{n}} \right),$$

so setting $\lambda = \lambda_n = C\sqrt{\frac{\log p}{n}}$ yields

$$\frac{1}{n} \|X(\widehat{\beta} - \beta_0)\|_2^2 = O_p\left(\frac{s \log p}{n} \right), \qquad \|\widehat{\beta} - \beta_0\|_1 = O_p\left(s\sqrt{\frac{\log p}{n}} \right),$$

and therefore lasso is **prediction consistent** if $s \log p / n \to 0$ and $\kappa(s, 3)$ is bounded away from zero.

4 Elastic Net

When features are highly correlated, we often want their coefficients to be similar:

- Ridge shrinks coefficients of highly correlated features together.
- Standard Lasso can exhibit unstable behavior with correlated predictors.

For example, with standard Lasso: if $X_j = X_k$ and $\hat{\beta}_j > 0$, then: (1) $\hat{\beta}_k \geq 0$ (same sign); (2) a different solution can be obtained by changing $(\hat{\beta}_j, \hat{\beta}_k)$ to $(0, \hat{\beta}_j + \hat{\beta}_k)$.

Unlike Ridge, the Lasso performs variable selection by shrinking some coefficients exactly to zero. It is favorable to compromise between them and leverage on the advantages of both.

Note that Lasso and Ridge can be seen as special cases of the bridge estimator (Frank and Friedman, 1993), which is defined as

$$\hat{\beta}^q = \arg\min_{\beta} \left\{ \frac{1}{2}\|y - X\beta\|^2 + \lambda \sum_{j=1}^{p} |\beta|^q \right\},$$

for some $q > 0$, and the contour regions of $\sum_{j=1}^{p} |\beta|^q \leq 1$ are illustrated in Figure 14.

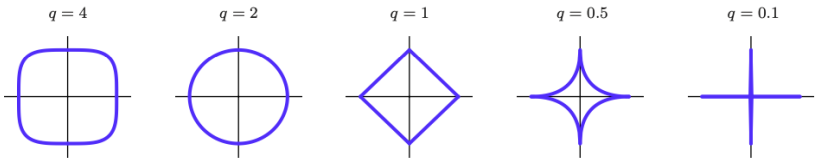


Figure 14: Constraint regions of the penalty (Figure 2.6 of Hastie et al. (2015)).

When $q \in (1, 2)$, $\hat{\beta}^q$ compromises between Ridge and Lasso. However, the computation is not straightforward.

4.1 The Elastic Net Formulation

Zou and Hastie (2005) proposed the Elastic Net, which combines the Lasso and Ridge penalties:

$$\min_{\beta \in \mathbb{R}^p} \left\{ \frac{1}{2}\|y - X\beta\|^2 + \lambda \left[(1-\alpha)\|\beta\|_1 + \alpha \|\beta\|_2^2 \right] \right\}$$

Where $\alpha \in [0, 1]$ controls the mixture of L_1 and L_2 penalties ($\alpha = 0$ gives Lasso, $\alpha = 1$ gives Ridge) and λ controls the overall strength of regularization.

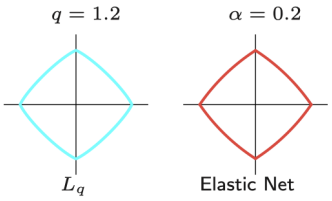


FIGURE 3.13. *Contours of constant value of* $\sum_j |\beta_j|^q$ *for* $q = 1.2$ *(left plot), and the elastic-net penalty* $\sum_j (\alpha\beta_j^2 + (1-\alpha)|\beta_j|)$ *for* $\alpha = 0.2$ *(right plot). Although visually very similar, the elastic-net has sharp (non-differentiable) corners, while the* $q = 1.2$ *penalty does not.*

Figure 15: Figure 3.13 of Hastie et al. (2009).

- **Uniqueness of solution**: When $\alpha > 0$, the solution is guaranteed to be unique due to strong convexity.
- **Variable selection capacity**: We can have more than n nonzeros.
- For highly correlated variables, coefficients tend to be similar. Specifically, if $\hat{\beta}_j \hat{\beta}_k > 0$, then

$$|\hat{\beta}_j - \hat{\beta}_k| \leq \frac{\|X_j - X_k\|_2}{2\lambda\alpha} \|y\|_2. \tag{8}$$

  - If $X_j = X_k$, then $\hat{\beta}_j = \hat{\beta}_k$.
  - The bound of $|\hat{\beta}_j - \hat{\beta}_k|$ gets tighter as X_j and X_k get closer.

Remark 3 (Proof of (8))**.** Denote $\hat{r} = y - X\hat{\beta}$. We derive the KKT conditions,

$$\begin{aligned} -X_j^T \hat{r} + \lambda(1-\alpha)s_j + 2\lambda\alpha\hat{\beta}_j = 0, \\ -X_k^T \hat{r} + \lambda(1-\alpha)s_k + 2\lambda\alpha\hat{\beta}_k = 0, \end{aligned}$$

where $s_j = s_k$ since $\hat{\beta}_j \hat{\beta}_k > 0$. Take the difference between two conditions, we have

$$2\lambda\alpha(\hat{\beta}_j - \hat{\beta}_k) = (X_j - X_k)^\top \hat{r}$$

By Cauchy-Schwarz inequality,

$$|\hat{\beta}_j - \hat{\beta}_k| = \frac{|X_j - X_k|^\top \hat{r}}{2\lambda\alpha} \leq \frac{\|X_j - X_k\|_2 \|\hat{r}\|_2}{2\lambda\alpha} \leq \frac{\|X_j - X_k\|_2 \|y\|_2}{2\lambda\alpha}$$

where the last inequality comes from the fact that $\hat{\beta}$ is the minimizer of the E-net problem by

noting that

$$\frac{1}{2}\|\hat{r}\|^2 + \lambda\left[(1-\alpha)\|\beta\|_1 + \alpha\|\beta\|_2^2\right] \leq \underbrace{\frac{1}{2}\|y\|^2}_{\text{plugging } \beta=0 \text{ in the objective}}.$$

4.2 Implementation of Elastic Net

One can solve the elastic net with a LASSO solver by noting that

$$\frac{1}{2}\|y - X\beta\|^2 + \lambda\left[(1-\alpha)\|\beta\|_1 + \alpha\|\beta\|_2^2\right] = \frac{1}{2}\left\|\underbrace{\begin{pmatrix} y \\ 0_p \end{pmatrix}}_{\text{new } y} - \underbrace{\begin{pmatrix} X \\ \sqrt{2\lambda\alpha} I_p \end{pmatrix}}_{\text{new } X}\beta\right\|^2 + \underbrace{\lambda(1-\alpha)}_{\text{new } \lambda}\|\beta\|_1 .$$

As $\alpha \downarrow 0$, we get the minimum ℓ_2-norm Lasso solution, which is the LARS algorithm solution.

5 Group LASSO

We may have predefined groups of variables and want to set all variables in the same group to 0 together.

E.g. 1: Each categorical variable with K categories corresponds to K dummy variables.

E.g. 2: Generalized additive models (GAMs): $y = f(x) + \varepsilon$ where $f(x) = \sum_{j=1}^{p} f_j(x_j) = \sum_{j=1}^{p}\sum_{\ell=1}^{q_j} \beta_{j\ell}\psi_{j\ell}(x_j)$. We might want to zero out $(\beta_{j1}, \ldots, \beta_{jq_j})$ together.

E.g. 3: Multitask learning (multivariate regression): When predicting $Y \in \mathbb{R}^{n\times m}$ using $X \in \mathbb{R}^{n\times p}$, we solve $\min_{B\in\mathbb{R}^{p\times m}} \frac{1}{2}\|Y - XB\|_F^2$, which decouples into m separate OLS problems. However, maybe there is some feature X_j that is irrelevant to all m responses, then we want to set $B_{j1}, B_{j2}, \cdots, B_{jm}$ to 0 together.

5.1 Group LASSO Formulation

Partition $\{1, \ldots, p\}$ into $g_1, \ldots, g_L$ groups. The Group Lasso (Yuan and Lin, 2006) optimization problem is:

$$\min_{\beta} \frac{1}{2}\|y - X\beta\|^2 + \lambda \sum_{\ell=1}^{L} w_\ell \|\beta_{g_\ell}\|_2$$

where w_ℓ are known weights.

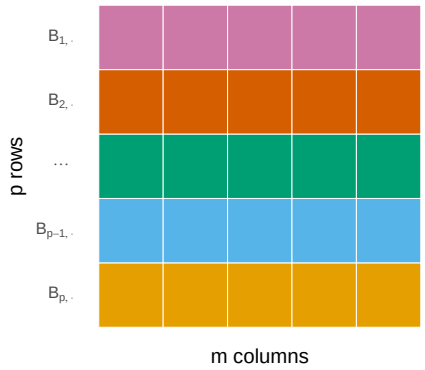


Figure 16: B matrix: Each row represents a group of variables.

Remark 4. The key ingredient of the group Lasso penalty is $\|\beta_{g_\ell}\|_2$ instead of $\|\beta_{g_\ell}\|_2^2$. This distinction enables the group Lasso to select groups of variables. Consider the following special cases.

- When $|g_\ell| = 1$, i.e. $g_\ell = \{\ell\}$, $\|\beta_{g_\ell}\|_2$ is just the absolute value, and Group Lasso reduces to standard Lasso.
- For $|g_\ell| = 2$, $\|\beta_{g_\ell}\|_2 = \sqrt{\beta_1^2 + \beta_2^2}$. The conic constraint has a non-differentiable kink at the origin, which is similar to Lasso.

If we check out the constraint region of the group Lasso penalty, it shares attributes of both the ℓ_1 and ℓ_2 balls.

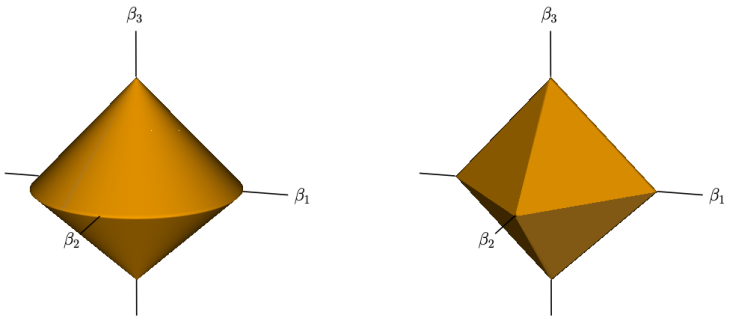


Figure 4.3 *The group lasso ball (left panel) in $\mathbb{R}^3$, compared to the ℓ_1 ball (right panel). In this case, there are two groups with coefficients $\theta_1 = (\beta_1, \beta_2) \in \mathbb{R}^2$ and $\theta_2 = \beta_3 \in \mathbb{R}^1$.*

Figure 17: Constraint regions of the group Lasso penalty (Figure 4.3 of Hastie et al. (2015)).

Take the sub-differential,

$$\partial\|\beta\|_2 = \begin{cases} \left\{\frac{\beta}{\|\beta\|_2}\right\} & \beta \neq 0 \\ \{u : \|u\|_2 \leq 1\} & \beta = 0 \end{cases}$$

and the proximal operator for the group Lasso:

$$\text{Prox}_{\lambda P_{GL}(\cdot)}(y) = \arg\min_{\beta} \left\{ \frac{1}{2}\|y-\beta\|^2 + \lambda \sum_{\ell=1}^{L} w_\ell \|\beta_{g_\ell}\|_2 \right\} = \underbrace{\left(1 - \frac{\lambda w_\ell}{\|y_{g_\ell}\|_2}\right)_+ y_{g_\ell}}_{\text{groupwise soft-thresholding}}$$

where $a_+ = \max\{a, 0\}$ for $a \in \mathbb{R}$.

This observation suggests a common choice of the weight, $w_\ell = |g_\ell|^{1/2}$, which accounts for the group size.

E.g. returning to multitask learning. The group Lasso estimator:

$$\min_{B \in \mathbb{R}^{p \times m}} \left\{ \frac{1}{2}\|Y - XB\|_F^2 + \lambda \sum_{j=1}^{p} \|B_{j\cdot}\|_2 \right\}.$$

Unlike standard regression, this formulation does not decouple — information is shared across all m response variables, encouraging common sparsity patterns.

Remark. In this example, groups are disjoint, which is relatively easy to deal with.

5.2 The Challenge of Overlapping Groups

In many applications, groups may overlap. For example, genes can belong to multiple biological *pathways*.

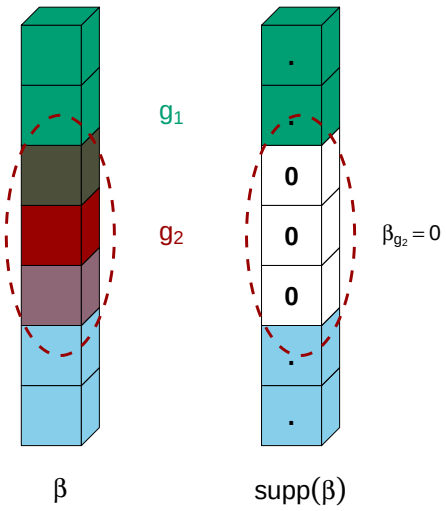


Figure 18: Graph illustration of overlapping groups. When groups overlap, the sparsity pattern might not be what we initially expected: $\text{supp}(\beta)$ is not simply a union of groups; instead, a union of groups is being set to zero.

5.2.1 Solving the Proximal via Dual Formulation

For overlapping Group Lasso, the proximal operator does not have a closed-form solution in general.

Jenatton et al. (2011) proposed solving the proximal problem by considering its dual:

$$(\hat{\eta}^{(1)}, \ldots, \hat{\eta}^{(L)}) \in \underset{\eta^{(1)},\eta^{(2)},\ldots,\eta^{(L)}}{\arg\min} \left\{ \frac{1}{2} \left\| y - \sum_{\ell=1}^{L} \eta^{(\ell)} \right\|^2 \text{ s.t. } \|\eta^{(\ell)}\|_2 \leq \lambda w_\ell,\ \eta^{(\ell)}_{g_\ell^c} = 0 \right\}$$

The primal solution is then retrieved as:

$$\hat{\beta} = \text{Prox}_{\lambda P_{GL}(\cdot)}(y) = y - \sum_{\ell=1}^{L} \hat{\eta}^{(\ell)}$$

Remark 5. When there's no overlap, this dual problem decouples by group, and each $\hat{\eta}^{(\ell)}_{g_\ell}$ is simply the projection of y_{g_ℓ} onto an ℓ_2-ball of radius λw_ℓ.

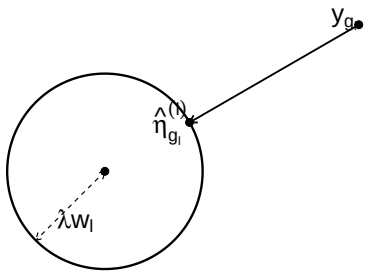


Figure 19: Illustration of the projection onto an ℓ_2-ball.

When groups do overlap, block-wise coordinate descent can be used to solve the dual problem.

5.2.2 The Latent Overlapping Group Lasso

Obozinski et al. (2011) proposed a different approach for handling overlapping groups. The core idea is to decompose β into group-specific components:

$$\beta = \sum_{\ell=1}^{L} v^{(\ell)} \quad \text{where} \quad v^{(\ell)}_{g_\ell^c} = 0,$$

and then penalize the sum of the norms of these components $\sum_\ell \|v^{(\ell)}\|_2$.

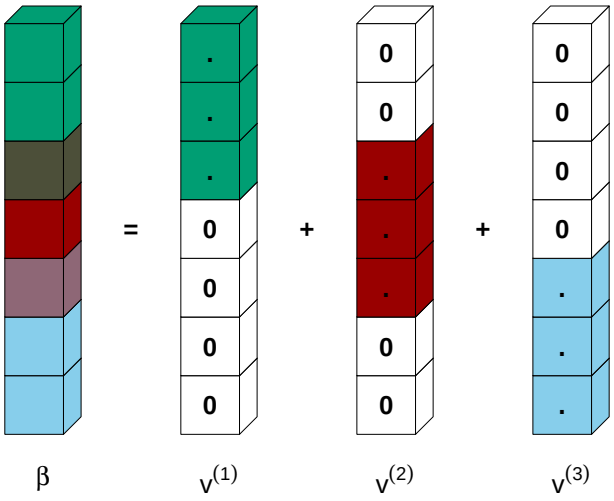


Figure 20: Graph illustration of the latent group Lasso. Notice that $\text{supp}(\hat{\beta})$ is union of some groups.

The *latent overlapping group Lasso* penalty:

$$P_{\text{LOG}}(\beta) = \min_{v^{(1)},\cdots,v^{(L)}\in\mathbb{R}^p} \left\{ \sum_{\ell=1}^{L} \|v^{(\ell)}\|_2 \text{ s.t. } v^{(\ell)}_{g_\ell^c} = 0 \text{ and } \beta = \sum_{\ell=1}^{L} v^{(\ell)} \right\}$$

Note: P_{LOG} is convex because it is the partial minimum of a joint convex function in $(v^{(1)}, \cdots, v^{(L)}, \beta)$:

$$P_{\text{LOG}}(\beta) = \min_{\{v^{(\ell)}\}} f(v^{(1)}, \cdots, v^{(L)}, \beta)$$

where

$$f(v^{(1)}, \cdots, v^{(L)}, \beta) = \sum_{\ell} \|v^{(\ell)}\|_2 + \sum_{\ell} \mathbb{I}_\infty\{v^{(\ell)}_{g_\ell^c} = 0\} + \mathbb{I}_\infty\{\beta = \sum_{\ell} v^{(\ell)}\}.$$

6 Adaptive Lasso

- **Formulation.** The adaptive Lasso (Zou, 2006) solves

$$\min_{\beta} \frac{1}{2}\|y - X\beta\|^2 + \lambda \sum_{j=1}^{p} \hat{w}_j |\beta_j|,$$

  where $\hat{w}_j = 1/|\hat{\beta}_j^{\text{init}}|^\gamma$ for some $\gamma > 0$, and $\hat{\beta}^{\text{init}}$ is an initial consistent estimator (e.g., OLS or Ridge when $p < n$).

- **Oracle property.** An estimator $\hat{\beta}$ has the *oracle property* if:

  1. *Model selection consistency:* $P(\hat{S} = S_0) \to 1$, where $S_0 = \{j : \beta_{0,j} \neq 0\}$.
  2. *Asymptotic normality:* the nonzero coefficient estimates satisfy

$$\sqrt{n}(\hat{\beta}_{S_0} - \beta_{0,S_0}) \xrightarrow{d} N\left(0,\ \sigma^2 \left(\frac{X'_{S_0} X_{S_0}}{n}\right)^{-1}\right),$$

     i.e., the same limiting distribution as OLS fit on the true support S_0.

- **Intuition.** The standard Lasso uses a single λ for all coefficients, creating a bias–selection tradeoff: a large λ zeros out irrelevant variables but over-shrinks the nonzero ones; a small λ reduces bias but fails to exclude noise variables. The adaptive weights resolve this tension:

  – For truly nonzero β_j: $|\hat{\beta}_j^{\text{init}}|$ is large $\implies$ $\hat{w}_j$ is small $\implies$ less penalization, less bias.

  – For truly zero β_j: $|\hat{\beta}_j^{\text{init}}|$ is small $\implies$ $\hat{w}_j$ is large $\implies$ strong penalization, shrunk to zero.

This differential penalization allows the adaptive Lasso to achieve the oracle property, which the standard Lasso in general cannot.

7 Fused Lasso

- Tibshirani et al. (2005): The fused Lasso solves

$$\min_{\beta} \frac{1}{2}\|y - X\beta\|^2 + \lambda_1\|\beta\|_1 + \lambda_2 \sum_{j=2}^{p} |\beta_j - \beta_{j-1}|.$$

  - The λ_1 penalty induces sparsity of coefficients (standard Lasso).
  - The λ_2 penalty encourages *smoothness*: consecutive coefficients are shrunk toward each other.

  Suitable when features have a natural ordering (e.g., time series, spatial location, genomic position).

- **1D signal approximation.** When $X = I_n$, the fused Lasso reduces to

$$\min_{\theta} \frac{1}{2}\|y - \theta\|^2 + \lambda_1\|\theta\|_1 + \lambda_2 \sum_{j=2}^{n} |\theta_j - \theta_{j-1}|,$$

  which estimates piecewise-constant signals with sparsity.

7.1 Variants and Applications of Fused Lasso

Structural Break Detection: Qian and Su (2016a,b)

- **Setting.** Time series or panel data where regression coefficients may change at unknown break dates: $y_t = x_t'\beta_t + \varepsilon_t$, with β_t piecewise constant over regimes.
- **Estimator.** Adaptive group fused Lasso — penalizes $\sum_{t=2}^{T} w_t\|\beta_t - \beta_{t-1}\|$ to shrink consecutive coefficient differences to zero.
- **Intuition.** If no break occurs at time t, then $\beta_t = \beta_{t-1}$ and the fused penalty enforces this. Break dates are identified where $\hat{\beta}_t \neq \hat{\beta}_{t-1}$. The method achieves consistent estimation of both the number and locations of breaks.

7.1.1 Trend Filtering (on Graphs): Wang et al. (2016)

- **Setting.** Signal-plus-noise model on a graph G: observe $y_i = \theta_i + \varepsilon_i$ at nodes of G.
- **Estimator.**

$$\min_{\theta} \frac{1}{2}\|y - \theta\|^2 + \lambda \|D\theta\|_1,$$

  where D is the edge-node incidence matrix of G.
- **Intuition.** Generalizes the 1D fused Lasso to graphs. The ℓ_1 penalty on graph differences induces piecewise-constant solutions with sharp transitions at edges, unlike ℓ_2 Laplacian smoothing which produces smooth decay.

7.1.2 Latent Group Structure in Panel Data: Mehrabani (2023)

- **Setting.** Panel data $y_{it} = x_{it}'\beta_i + \varepsilon_{it}$, where individuals belong to unknown groups and individuals within a group share the same coefficient vector:

$$\beta_i = \alpha_k$$

  for some $k \in \{1, 2, \cdots, K\}$.
- **Estimator (PAGFL).**

$$\min_{\beta_1,\ldots,\beta_N} \sum_{i=1}^{N}\sum_{t=1}^{T}(y_{it} - x_{it}'\beta_i)^2 + \lambda \sum_{i<j} w_{ij}\|\beta_i - \beta_j\|_2$$

  with adaptive weights w_{ij}.
- **Intuition.** The pairwise fused penalty merges individuals whose coefficients are similar into the same group. Group membership and group-specific parameters are estimated simultaneously.
- The Classifier Lasso (Su et al., 2016) uses a similar pairwise fused penalty formulation for identifying latent group structures in panel data, achieving the oracle property.

$$\min_{\beta_1,\ldots,\beta_N,\alpha_1,\ldots\alpha_K} \sum_{i=1}^{N}\sum_{t=1}^{T}(y_{it} - x_{it}'\beta_i)^2 + \lambda \sum_{i=1}^{N}\prod_{k=1}^{K}\|\beta_i - \alpha_k\|_2,$$

- Gao and Shi (2021) develop a MOSEK-based computational approach for solving the pairwise fused penalization problem.

8 Dantzig Selector

- **Formulation.** (Candes and Tao, 2007)

$$\min_{\beta} \|\beta\|_1 \quad \text{subject to} \quad \left\| \frac{1}{n} X'(y - X\beta) \right\|_\infty \leq \lambda_n.$$

- **Connection to Lasso KKT.** The Lasso KKT condition (5) states $\|X'(y - X\hat{\beta})/n\|_\infty \leq \lambda$. The Dantzig selector relaxes the normal equations $X'(y - X\beta)/n = 0$ to approximate satisfaction within an ℓ_∞-tolerance λ_n, then picks the sparsest (ℓ_1-minimizing) solution.

- **Linear program.** Unlike the Lasso (a QP), the Dantzig selector is a linear program, solvable via standard LP solvers.

- **Theory.** Bickel et al. (2009): under the RE condition, the Lasso and Dantzig selector achieve the same convergence rates:

$$\frac{1}{n}\|X(\hat{\beta} - \beta_0)\|_2^2 = \mathcal{O}_p\left(\frac{s \log p}{n}\right), \qquad \|\hat{\beta} - \beta_0\|_1 = \mathcal{O}_p\left(s\sqrt{\frac{\log p}{n}}\right).$$

- Econometric methods inspired by the Dantzig Selector:
  - Shi (2016): Extends the Dantzig idea to moment condition models. Just as the Dantzig selector relaxes the normal equations to an ℓ_∞ constraint, this approach relaxes high-dimensional moment equalities $\mathbb{E}[g(Z, \beta)] = 0$ to approximate satisfaction, enabling inference in high-dimensional GMM settings.
  - Shi et al. (2025): The ℓ_2-relaxation replaces $\min \|\beta\|_1$ with $\min \|\beta\|_2^2$ subject to similar relaxed constraints, producing dense (diversified) solutions suitable for forecast combination and portfolio analysis.

9 Best Subset Selection

- **Formulation.**

$$\min_{\beta} \frac{1}{2}\|y - X\beta\|^2 \quad \text{subject to} \quad \|\beta\|_0 \leq k,$$

where $\|\beta\|_0 = \#\{j : \beta_j \neq 0\}$ counts the number of nonzero coefficients.

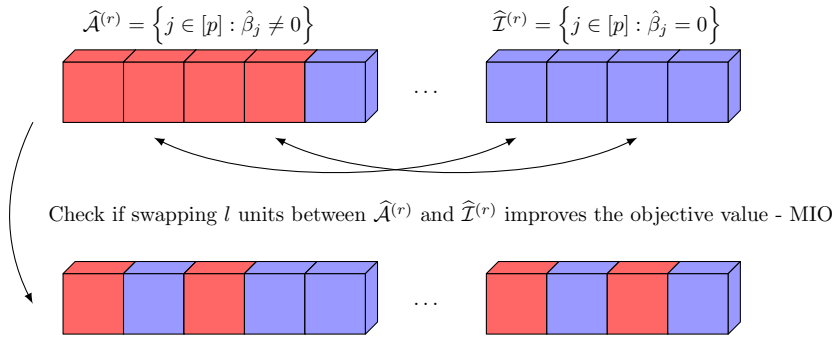
- **Combinatorial challenge.** There are $\binom{p}{k}$ possible subsets. Exhaustive search is NP-hard in general. Even for moderate dimensions (e.g., $p = 40$, $k = 20$), there are over 10^{11} subsets.

- **MIO formulation** (Bertsimas et al., 2016): Introduce binary variables $z_j \in \{0, 1\}$:

$$\min_{\beta, z} \frac{1}{2}\|y - X\beta\|^2 \quad \text{s.t.} \quad |\beta_j| \leq M_j z_j, \ \sum_{j=1}^{p} z_j \leq k, \ z_j \in \{0, 1\},$$

  where M_j are upper bounds on coefficient magnitudes. Modern MIO solvers (e.g., Gurobi) can handle moderate dimensions.

- **Local combinatorial search** (Hazimeh and Mazumder, 2020): Combines coordinate descent with local combinatorial swaps for scalable computation.



Part III

Tree-based Methods

1 Regression Trees

- To motivate regression trees, recall the binned sample mean estimator:
  1. Place the data into bins according to X_i.
  2. Given X_{n+1}, predict Y_{n+1} using the sample mean of Y_i's such that X_i is in the same bin as X_{n+1}.
- We dealt with univariate X_i, and we formed our bins using quantiles of X_i.
  - Does this make sense? What if $\mathbb{E}[Y_i|X_i = x]$ is flat in some regions and wiggly in others? Then bin size should vary with x.
  - How to generalize to multivariate X_i?
    - *Curse of dimensionality*: if we have d variables $X_{i,1}, \ldots, X_{i,d}$ and we cut our observations by each variable, we have $(c+1)^d$ bins where c is the number of cutpoints. This quickly gets to be $\gg n$ so that we have no observations in most bins.
    - Can we use variable selection (similar to lasso) to avoid this?
- A *regression tree* is a particular way of forming these bins which addresses these issues to some extent.
  - We'll use regression trees for classification as well. This whole class of methods is sometimes called "CART" for "classification and regression trees" (Breiman et al., 1984).

1.1 Decision Trees

- The idea of regression trees is to form our bins as *leaves* on a *decision tree*.
- A decision tree is a nested series of if-else statements that take an input and output a conclusion.
- Here the input is weather; output is umbrella usage.

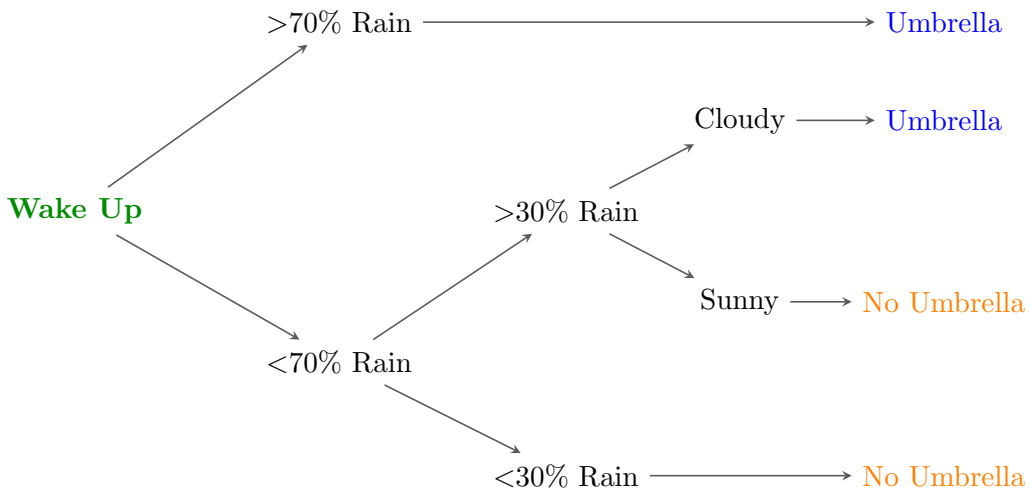


Figure 21: Example of a decision tree.

- This is a series of *nodes* with a *parent/child structure*. *Leaf nodes* at the bottom specify conclusions.
- This type of diagram for visualizing a decision tree is called a *dendrogram*.

1.2 Regression Trees

- In a regression tree, each node is a threshold for splitting on the basis of one of the variables of $X_i = (X_{i,1}, \ldots, X_{i,d})$. E.g. if a node is $X_{ik} = 0$, then the tree filters the input left if $X_k \leq 0$ and otherwise right.

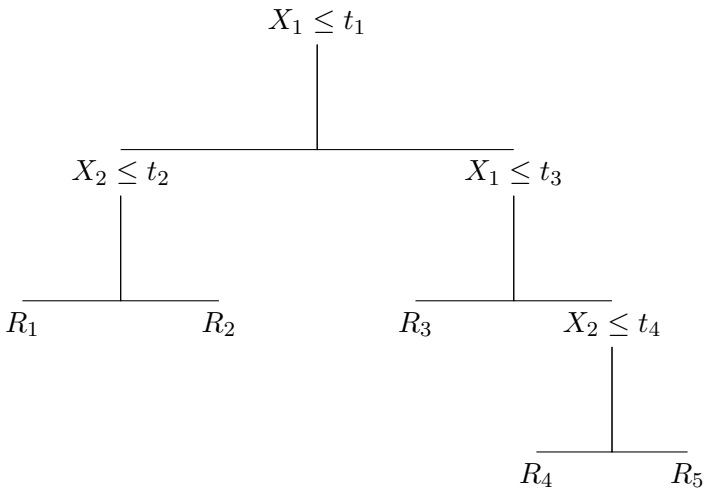


Figure 22: Example regression tree.

- In this case, input X_i belongs in the leftmost leaf node R_1 if $X_{i,1} \leq t_1$ and $X_{i,2} \leq t_2$.
- Given a prespecified tree of this type, we estimate the tree by sending a sample of n observations through the tree.
- When we reach the leaf node, we take the observations meeting its criteria and output a "prediction" based on the type of variable Y.
  - If Y is real-valued, we output the average outcome as the prediction. E.g. in the leftmost leaf node, the output is $\hat{\mathbb{E}}[Y \mid X_{i,1} \leq t_1, X_{i,2} \leq t_2]$.
  - If Y is categorical, we output the fraction of observations that belong to each category, e.g. $\hat{\Pr}(Y = m \mid X_{i,1} \leq t_1, X_{i,2} \leq t_2)$ for each category m.
- Thus, our regression tree estimator is just the binned sample mean, with the bins given by the leaf nodes of the tree.
- Given a new observation X_{n+1}, if we want a prediction based on our past data, we input X_{n+1} into the tree and follow the logic to figure out the leaf node for X_{n+1}.
- It's common to use tree diagrams to visualize the results of estimation/classification with regression trees. If X_i only has one or two variables, we can also draw a graph illustrating how the leaves partition X_i.
- We can then visualize the regression function $\mathbb{E}[Y_i|X_i = x]$ as a step function, which is constant over the leaves.
- The partition created by the tree is called a *recursive binary partition*: we partition a subset of the feature space into two further subsets at each node.

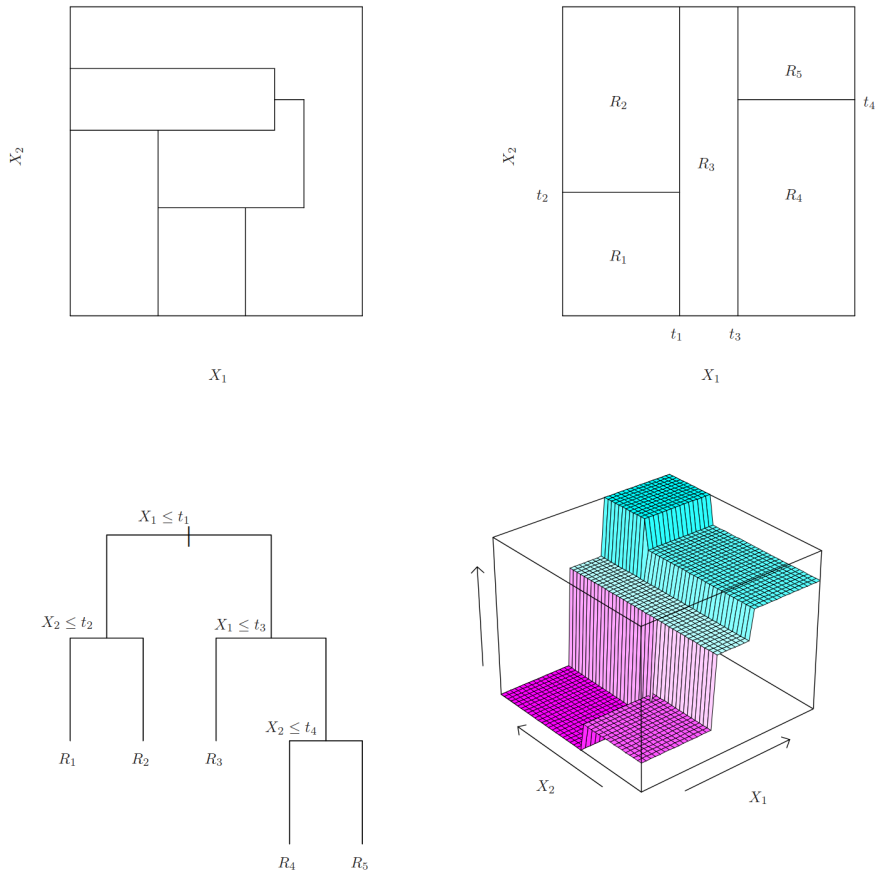


FIGURE 8.3. Top Left: *A partition of two-dimensional feature space that could not result from recursive binary splitting.* Top Right: *The output of recursive binary splitting on a two-dimensional example.* Bottom Left: *A tree corresponding to the partition in the top right panel.* Bottom Right: *A perspective plot of the prediction surface corresponding to that tree.*

Figure 23: Regression Tree Illustration

- Recursive binary partitioning rules out certain types of partitions, such as this one:

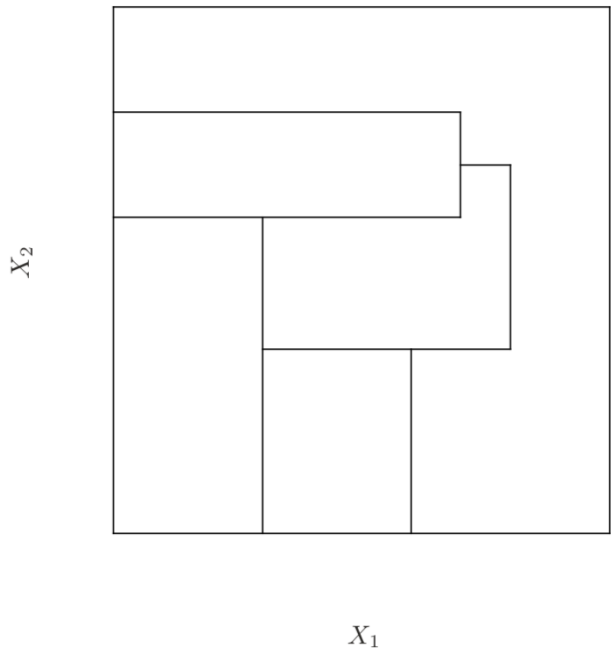


Figure 24: A non-tree partition of the feature space.

1.3 Growing Trees

- How do we pick the tree (set of splitting rules)? We will choose a rule to minimize some *deviance* or *loss function*.

- If Y is real-valued, we can use as the deviance the sum of squares

$$\sum_{i=1}^{n}(Y_i - \hat{\mathbb{E}}[Y \mid X_i])^2.$$

  Compare with OLS where we use $X_i'\hat{\beta}$ in place of $\hat{\mathbb{E}}[Y \mid X_i]$.

- If Y is categorical, we can use the multinomial deviance

$$-\sum_{i=1}^{n}\sum_{k=1}^{K}\mathbb{1}\{Y_i = k\}\log\hat{\Pr}(Y_i = k \mid X_i).$$

- Actually, most software implementations use a different deviance for the categorical case, the

"Gini impurity", which estimates the variance of $\mathbb{1}\{Y_i = k\}$:

$$\sum_{i=1}^{n}\sum_{k=1}^{K}\hat{\Pr}(Y_i = k \mid X_i)(1 - \hat{\Pr}(Y_i = k \mid X_i)).$$

- Both choices of deviance tend to perform the same, so either is fine.
- Ideally, we'd like to search over all possible trees to choose the one that minimizes the deviance, but this is computationally infeasible. The set of possible trees is exponentially large.
  - The regression tree that minimizes all possible trees is called *optimal regression tree* (ORT). See Bertsimas and Dunn (2019, Section 7 - 11) for a systematic discussion on how to compute ORT via a mixed-integer programming approach.
- Instead, we search through the space of trees using a "top-down", "greedy" algorithm called *recursive binary splitting*.

1.4 Splitting Procedure

- Given data $\{(Y_i, X_i)\}_{i=1}^n$, the first split we make is to choose an observation i and dimension j, with associated covariate X_{ij}, and divide the data into left and right halves.
  - Left half: all observations ℓ such that $X_{\ell j} \leq X_{ij}$.
  - Right half: all observations ℓ such that $X_{\ell j} > X_{ij}$.
- For each half, we form a temporary "prediction" based on whether Y is real or categorical.
  - Real Y: prediction is average Y in the half.
  - Categorical Y: predictions are fraction of observations in the half that are of each category.
- Based on that prediction, we compute the deviance for all observations.
- E.g. for real Y, the deviance is

$$\sum_{i=1}^{n}\left[(Y_i - \bar{Y}_{\text{left}})^2\mathbb{1}\{i \in \text{left half}\} + (Y_i - \bar{Y}_{\text{right}})^2\mathbb{1}\{i \in \text{right half}\}\right],$$

where $\bar{Y}_{\text{left}}$ is the average Y in the left half.

- The deviance measures how badly this choice of i and j ends up predicting. Finally, we pick the **optimal split** by minimizing this deviance across all i and j.
- This completes the first split. Now we repeat this procedure at each child node. E.g. in the left child node, we repeat this only using the subset of observations $\{(Y_i, X_i) \colon i \in \text{left half}\}$.

1.5 CART Algorithm

To summarize, we use the following algorithm:

1. Determine the optimal split X_{ij} across observations i and variables j that minimizes the deviance.
2. Split this parent node into left and right child nodes and filter the observations into their appropriate nodes.
3. Repeat the previous steps for each child node.
4. Continue recursively until a **stopping criterion** is met at a node, at which point the node becomes a leaf. Two standard criteria govern this:
   - `mincut`: the minimum number of observations that must be present in a node for it to be eligible for splitting. A node with fewer than `mincut` observations is declared a leaf. For example, `mincut` $= 5$ prevents any node with fewer than 5 observations from being split further.
   - `mindev`: the minimum *relative* reduction in deviance required to justify a split. Specifically, a split is only performed if

     $$\frac{\text{deviance reduction at this node}}{\text{deviance at the root node}} \geq \texttt{mindev}.$$

     A split that yields a proportional deviance reduction smaller than `mindev` is rejected and the node becomes a leaf.

   Splitting stops at a node when *either* criterion is violated.

1.6 Pruning Trees

- The trees we estimate are generally overfit because they fit the data too flexibly.
- In the extreme case where `mincut` $= 1$ and `mindev` $= 0$, you'll split all the way down to one-observation leaves, which are maximally overfit.

- Again we face the *bias-variance tradeoff*.
- What can we do? The most obvious idea would be to treat `mincut` and `mindev` as complexity parameters and use cross-validation to choose them.
- However, this can lead the tree to stop too early. The issue is that the algorithm is "greedy" and lacks "foresight."
- Consider applying regression trees to the (fake) dataset in Figure 25, which looks like the flag of the Dominican Republic.
  - The first split will give little improvement in predictive power, but the second split will give perfect predictive power!

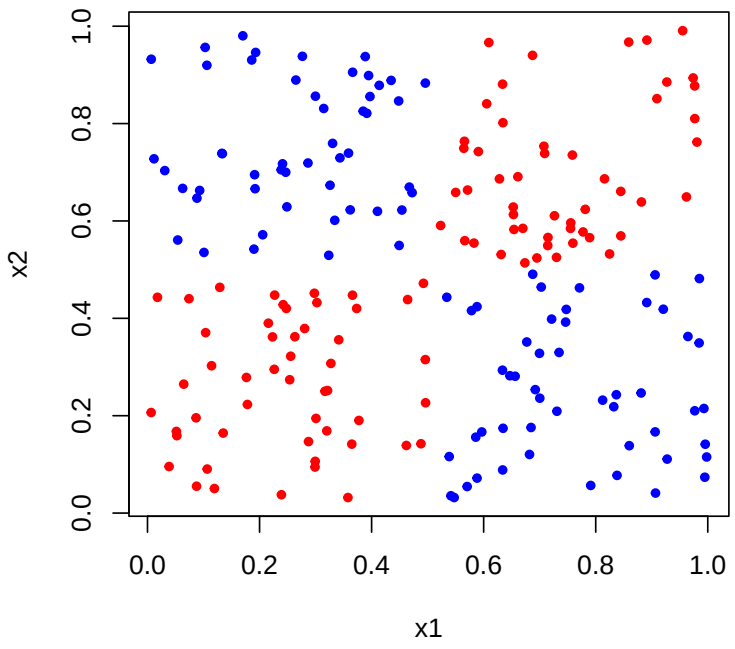


Figure 25: A dataset where the first split gives little improvement but the second split gives perfect predictive power.

- So this rules out using `mindev` as a tuning parameter: we should probably set `mindev` $= 0$ or at least set it very small so that we get to the second cut in the above example.

- What about using `mincut` (i.e. the minimum number of observations in a node)? If this is the only tuning parameter and `mindev` $= 0$, we'll end up with approximately the same number of observations in each bin, which basically just gets us back to the binned sample mean.
- To avoid these issues, the typical technique is to first grow a deliberately overfit ("overgrown") tree and then "prune" it back by removing the leaf that *least increases the deviance*. We can then iterate this process, gradually reversing the initial tree-growing process.
- We make the "overgrown" tree by choosing `mindev` $= 0$ (or, at least, very small) and `mincut` also very small.
- We then "prune" the tree by iteratively eliminating the cut points that give the least improvement in deviance.
  - This is sometimes called "weakest link pruning," since we eliminate the node that gives the least improvement in deviance.
  - It turns out that pruning in this way does not lead to the problems with "lack of foresight" that can occur with growing the tree: it's guaranteed to find a sequence of trees such that each tree minimizes the deviance among *all* subtrees of each given size (see references on p. 308 of Hastie et al. (2009)).
- Pruning gives us a sequence of models that we can select from using penalization and cross-validation.

See the companion notebook `example_tree.qmd` for illustrations of pruning with the TV genre tree data.

1.7 Cost-Complexity Pruning

- The pruning algorithm produces a sequence of trees of increasing size (measured by the number of leaves).
- To avoid overfitting, we choose the size of the tree by adding a penalty term $\lambda \cdot \{\text{number of leaves}\}$ that penalizes the number of leaves. This is called *cost-complexity pruning*.
- We then choose the penalty weight λ using cross validation.
- To summarize, we use the following algorithm (adapted from the description in James et al. (2021), p. 333):

1. Grow a large tree, stopping only when each terminal node ("leaf") has fewer than some minimum number of observations.

2. Apply cost complexity pruning to the large tree: for each λ, minimize

$$\sum_{i=1}^{n}(Y_i - \hat{\mathbb{E}}_{\text{subtree } T}(Y|X_i))^2 + \lambda \cdot \{\text{number of leaves in subtree } T\}$$

   using cost-complexity pruning.

3. Use K-fold cross-validation to choose λ: divide the training observations into K folds and, for each fold $k = 1, \ldots, K$:

   (a) Repeat Steps 1 and 2 on all but the kth fold of the training data.

   (b) Evaluate the mean squared prediction error on the data in the left-out kth fold, as a function of λ.

   Average the results for each value of λ, and pick λ to minimize the average error.

4. Return the subtree from Step 2 that corresponds to the chosen value of λ.

2 Random Forests

2.1 Bagging

- Random forests are a class of estimators that apply a technique called *bagging* to regression trees.

- "Bagging" refers to "Bootstrap AGgregation."

- Recall that we originally used the bootstrap to assess sampling uncertainty:

  1. For $b = 1, \ldots, B$, draw a new sample of observations, with replacement, from the original data.

  2. Compute a bootstrapped estimator $\hat{\beta}_b^*$ from each of the bootstrap replications $b = 1, \ldots, B$.

  3. Form a CI by adding and subtracting the 0.95 quantile of the distribution of $|\hat{\beta}_b^* - \hat{\beta}|$ across the bootstrap replications $b = 1, \ldots, B$.

- In Bootstrap AGgregation, we instead use the bootstrap estimators $\hat{\beta}_b^*$ to form an estimate $\hat{\beta}_{\text{BAG}} = \frac{1}{B}\sum_{b=1}^{B}\hat{\beta}_b^*$.

- A *random forest* gets a regression tree estimate $\hat{f}_b^*(x)$ from each bootstrap sample, and then performs bagging to get the estimate $\frac{1}{B}\sum_{b=1}^{B}\hat{f}_b^*(x)$.
- There are also a few modifications to this algorithm that are typically done with random forests.
- Regression trees are a good candidate for bagging because the cutpoints have "too much randomness:" they depend too much on small changes in the data. Bagging averages out this randomness, thereby lowering variance.

2.2 Random Forests

- When applied to regression trees, bagging produces a *random forest*.
- Random forests actually involve a slight variation. In each bootstrap sample, when choosing the best covariate to split on, the algorithm only uses a random subsample of m out of the d covariates rather than all covariates.
  - So, we get the standard version of bagging applied to regression trees when $m = d$.
- Why does this help? It forces the greedy splitting algorithm to avoid always splitting on variables in a similar order.
  - Recall the example where the data looked like the flag of the Dominican Republic: it can help to choose a split that is suboptimal from a "greedy" standpoint if it helps later on.
  - The cost is additional randomness (since we randomly choose variables in the split), but this is less of a problem now since we're averaging it out through bagging.
- Disadvantage of random forest relative to tree is you lose interpretability. There's no tree plot since it's an average of many trees.
  - Breiman (2001) remarks:

    > *On interpretability, trees rate an A+ ... So forests are A+ predictors. BUt their mechanism for producing a prediction is difficult to understand. Trying to delve into the tangled web that generated a plurality vote from 100 tree is a Herculean task. So on interpretability, they rate an F.*

  - Instead, we can make a *variable importance plot*, which averages the improvement in deviance from splitting along a given variable over the B trees.

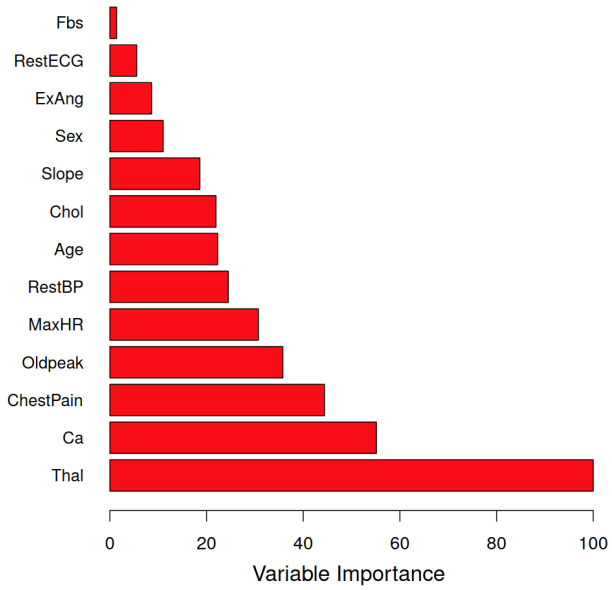


Figure 26: Variable Importance Plot from James et al. (2021)

- An appealing feature of random forests is that they seem to do well without much tuning: we don't use cross-validation to choose tree-depth like we did before.
  - The main tuning parameter is m (the number of covariates used for each split), and this is typically just chosen as $m = \sqrt{d}$ where d is the total number of covariates (although we could also use cross-validation to choose it).
  - We also have to choose the minimum number of observations in a leaf in each tree, but we typically just take it to be very small. The reasoning for this is that it's fine to overfit each individual tree, since bagging will average out the randomness.

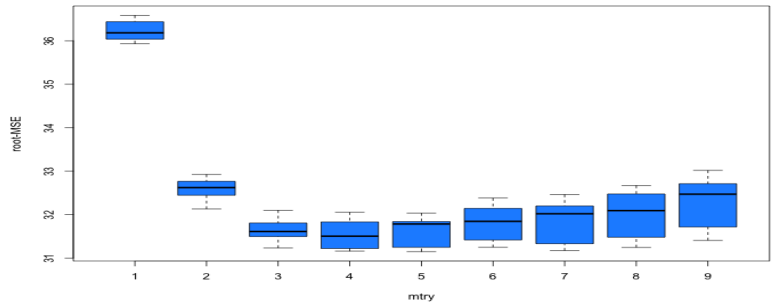


Figure 27: Sensitivity of random forest performance to the m (number of variables considered at each split). Boxplots show root-MSE across several repetitions as m varies.

2.2.1 Random Forest Algorithm

To summarize, we form a random forest as follows (adapted from p. 588 in Hastie et al. (2009)):

1. For $b = 1$ to B:
   (a) Draw a bootstrap sample from the data (same sample size, with replacement).
   (b) Grow a tree T_b to the bootstrapped data, by repeating the following steps for each terminal node of the tree, until the minimum node size is reached:
      - Select m variables at random from the d variables.
      - Pick the best variable/split-point among these m variables.
      - Split the node into two daughter nodes.
2. Output the ensemble of trees $\{T_b\}_{b=1}^{B}$.

- For regression, we output the average of the tree predictions: $\hat{f}_{\text{RF}}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}_b^*(x)$.
- For classification, it is common to instead use majority vote: the predicted class of Y_{n+1} for $X_{n+1} = x$ is the class such that the largest number of the B trees predict this class for $X_{n+1} = x$.

3 Boosted Trees

- Boosting is another approach for improving the predictions resulting from decision trees. Like bagging, it is a general technique that can be applied to many statistical learning methods, but here we focus on decision trees.
- Recall that bagging involves creating multiple copies of the training data via the bootstrap, fitting a separate tree to each copy independently, and then averaging the predictions. Each tree is built on a bootstrap sample, independent of the other trees.
- Boosting works differently: the trees are grown *sequentially*, with each tree grown using information from previously grown trees. Boosting does not involve bootstrap sampling; instead each tree is fit on a modified version of the original data.
- The key idea is that, unlike fitting a single large tree to the data, the boosting approach instead *learns slowly*.

3.1 Boosting Algorithm for Regression Trees

Consider the regression setting with data $\{(Y_i, X_i)\}_{i=1}^n$. The boosting algorithm builds up the model by sequentially fitting small trees to the residuals of the current fit (adapted from p. 345 in James et al. (2021)):

1. Set $\hat{f}(x) = 0$ and $r_i = Y_i$ for all i in the training set.

2. For $b = 1, 2, \ldots, B$, repeat:

   (a) Fit a regression tree $\hat{f}_b$ with J terminal nodes to the training data (X, r), where $r = (r_1, \ldots, r_n)'$ are the current residuals.

   (b) Update the model by adding in a shrunken version of the new tree:

   $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}_b(x).$$

   (c) Update the residuals:

   $$r_i \leftarrow r_i - \lambda \hat{f}_b(X_i).$$

3. Output the boosted model:

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}_b(x).$$

3.2 Intuition

- What is the idea behind this procedure? Given the current model $\hat{f}$, we fit a decision tree to the *residuals* from the model. That is, we fit a tree using the current residuals r_i, rather than the outcome Y_i, as the response. We then add this new tree into the fitted function in order to update the residuals.

- Each of these trees can be rather small, with just a few terminal nodes (determined by J). By fitting small trees to the residuals, we slowly improve $\hat{f}$ in areas where it does not perform well.

- The shrinkage parameter λ slows the process down even further, allowing more and differently shaped trees to attack the residuals.

- In general, statistical learning approaches that *learn slowly* tend to perform well. This is related to the idea that regularization (controlling the complexity of the model) helps prevent overfitting.

- Note that in boosting, unlike in bagging, the construction of each tree depends strongly on the trees that have already been grown. This is fundamentally different from random forests, where each tree is built independently.

3.3 Tuning Parameters

Boosting has three tuning parameters:

- **The number of trees B.** Unlike bagging and random forests, boosting can overfit if B is too large, although this overfitting tends to occur slowly if at all. We use cross-validation to select B.
- **The shrinkage parameter λ.** This is a small positive number that controls the rate at which boosting learns. Typical values are 0.01 or 0.001, and the right choice can depend on the problem. Very small λ can require using a very large value of B in order to achieve good performance. There is a tradeoff between λ and B: smaller λ requires larger B.
- **The number of terminal nodes J.** This controls the complexity of each individual tree in the boosted ensemble.
  - Often $J = 2$ works well, in which case each tree is a *stump* consisting of a single split. In this case, the boosted ensemble is fitting an *additive model*, since each term involves only a single variable.
  - More generally, $J - 1$ controls the *interaction depth* of the boosted model, since a tree with $J-1$ splits can involve at most $J-1$ variables. For example, $J = 3$ allows pairwise interactions.

3.4 Discussion

- In boosting, because the growth of a particular tree takes into account the other trees that have already been grown, smaller trees are typically sufficient. Using smaller trees can aid in interpretability as well; for instance, using stumps ($J = 2$) leads to an additive model.
- A key difference from random forests is the potential for overfitting. Random forests will not overfit as B increases (adding more trees just averages out the noise further), but boosting can overfit if B is too large. In practice, however, this overfitting tends to occur slowly, especially when λ is small.

- Boosting classification trees proceeds in a similar but slightly more complex way. The basic idea is the same—sequentially fitting weak learners to improve the overall model—but the loss function and update rules differ. A prominent example is AdaBoost (Freund and Schapire, 1997), which reweights observations at each step, placing more weight on misclassified observations.

- Gradient boosting generalizes the algorithm above by replacing the squared-error residuals with the negative gradient of an arbitrary differentiable loss function. The algorithm presented above is gradient boosting with squared error loss. See Hastie et al. (2009, Section 10.10) for details.

- A widely used implementation of gradient boosting is *XGBoost* (eXtreme Gradient Boosting) (Chen and Guestrin, 2016). XGBoost extends the basic gradient boosting framework in several ways:

  - It uses a second-order Taylor expansion of the loss function, incorporating both the gradient and the Hessian (second derivative). This amounts to performing Newton steps in function space rather than gradient descent steps, leading to faster convergence.
  - It adds explicit regularization to the objective: in addition to the loss, the objective includes a penalty on the number of leaves and the ℓ_2 norm of the leaf predictions, which helps control overfitting.
  - It includes a sparsity-aware algorithm that efficiently handles missing values and sparse features.
  - It supports column subsampling (as in random forests), which further reduces overfitting and speeds up computation.

  XGBoost has been highly successful in practice, notably dominating many Kaggle machine learning competitions. Other popular implementations of gradient boosting include LightGBM (Ke et al., 2017) and CatBoost (Prokhorenkova et al., 2018).

4 Bayesian Additive Regression Trees

- *Bayesian additive regression trees* (BART) is another ensemble method that uses decision trees as its building blocks. For simplicity, we present BART for regression.

- Recall that bagging and random forests make predictions from an average of regression trees, each built using a random sample of data and/or predictors. Each tree is built separately

from the others. By contrast, boosting uses a weighted sum of trees, each constructed by fitting a tree to the residual of the current fit.

- BART is related to both approaches: each tree is constructed in a random manner as in bagging and random forests, and each tree tries to capture signal not yet accounted for by the current model, as in boosting. The main novelty in BART is the way in which new trees are generated.

4.1 Notation and Setup

- Let K denote the number of regression trees, and B the number of iterations for which the BART algorithm will be run.
- The notation $\hat{f}_k^b(x)$ represents the prediction at x for the kth regression tree used in the bth iteration.
- At the end of each iteration, the K trees from that iteration are summed:

$$\hat{f}^b(x) = \sum_{k=1}^{K} \hat{f}_k^b(x), \quad \text{for } b = 1, \ldots, B.$$

4.2 BART Algorithm

The BART algorithm proceeds as follows (adapted from p. 348 in James et al. (2021)):

1. Initialize all K trees to have a single root node:

$$\hat{f}_1^1(x) = \hat{f}_2^1(x) = \cdots = \hat{f}_K^1(x) = \frac{1}{nK} \sum_{i=1}^{n} Y_i.$$

   Thus $\hat{f}^1(x) = \sum_{k=1}^{K} \hat{f}_k^1(x) = \frac{1}{n} \sum_{i=1}^{n} Y_i$, the sample mean.

2. For $b = 2, \ldots, B$:

   (a) For $k = 1, 2, \ldots, K$:

      i. For $i = 1, \ldots, n$, compute the current *partial residual*:

$$r_i = Y_i - \sum_{k'<k} \hat{f}_{k'}^b(X_i) - \sum_{k'>k} \hat{f}_{k'}^{b-1}(X_i).$$

This removes from Y_i the predictions of all trees except the kth, using the already-updated trees (indices $k' < k$) from the current iteration and the not-yet-updated trees (indices $k' > k$) from the previous iteration.

ii. Fit a new tree $\hat{f}_k^b(x)$ to the partial residuals r_i, by *randomly perturbing* the kth tree from the previous iteration, $\hat{f}_k^{b-1}(x)$. Perturbations that improve the fit to the partial residuals are favored.

(b) Compute $\hat{f}^b(x) = \sum_{k=1}^{K} \hat{f}_k^b(x)$.

3. Compute the final prediction by averaging over iterations after a *burn-in* period of L iterations:

$$\hat{f}(x) = \frac{1}{B-L} \sum_{b=L+1}^{B} \hat{f}^b(x).$$

Key Ideas:

- A key element of the BART approach is that in Step 2(a)ii., we do *not* fit a fresh tree to the current partial residual. Instead, we try to improve the fit by slightly modifying the tree obtained in the previous iteration. This guards against overfitting since it limits how "hard" we fit the data in each iteration.

- There are two components to the perturbation of each tree:

  1. We may change the *structure* of the tree by adding or pruning branches.
  2. We may change the *prediction* in each terminal node of the tree.

- The individual trees are typically quite small. We limit the tree size in order to avoid overfitting, which would be more likely to occur if we grew very large trees.

- We throw away the first L iterations (the *burn-in* period) because models obtained in the earlier iterations tend not to provide good results. After the burn-in, the ensemble predictions stabilize.

- Beyond the mean, the collection $\hat{f}^{L+1}(x), \ldots, \hat{f}^B(x)$ can be used to quantify uncertainty: for instance, the percentiles of these predictions provide a measure of uncertainty in the final prediction, analogous to a credible interval.

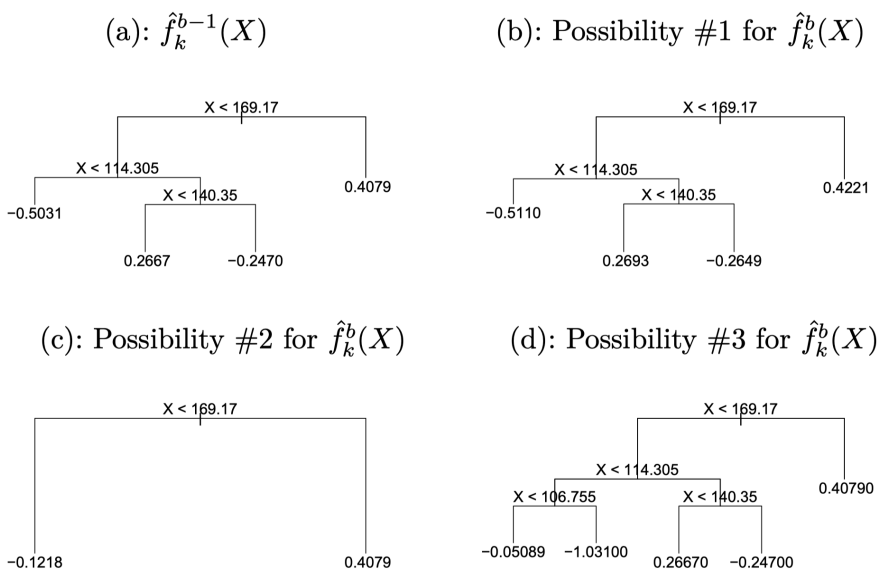


FIGURE 8.12. *A schematic of perturbed trees from the BART algorithm.* (a): *The kth tree at the* $(b-1)$*st iteration,* $\hat{f}_k^{b-1}(X)$*, is displayed. Panels (b)–(d) display three of many possibilities for* $\hat{f}_k^b(X)$*, given the form of* $\hat{f}_k^{b-1}(X)$*.* (b): *One possibility is that* $\hat{f}_k^b(X)$ *has the same structure as* $\hat{f}_k^{b-1}(X)$*, but with different predictions at the terminal nodes.* (c): *Another possibility is that* $\hat{f}_k^b(X)$ *results from pruning* $\hat{f}_k^{b-1}(X)$*.* (d): *Alternatively,* $\hat{f}_k^b(X)$ *may have more terminal nodes than* $\hat{f}_k^{b-1}(X)$*.*

Figure 28: Illustration of BART (Bayesian Additive Regression Trees). The ensemble model aggregates predictions from many small trees, combining flexibility with regularization.

4.3 Tuning Parameters and Bayesian Interpretation

- BART has three tuning parameters: the number of trees K, the number of iterations B, and the number of burn-in iterations L. We typically choose large values for B and K, and a moderate value for L: for instance, $K = 200$, $B = 1{,}000$, and $L = 100$ is a reasonable choice.

- BART has been shown to have very impressive out-of-the-box performance—that is, it performs well with minimal tuning.

- The BART method can be viewed as a Bayesian approach to fitting an ensemble of trees: each time we randomly perturb a tree in order to fit the residuals, we are in fact drawing a new tree from a *posterior distribution*. The algorithm can be viewed as a *Markov chain Monte Carlo* (MCMC) algorithm for sampling from this posterior, which is why we discard the initial burn-in samples and average over the remaining draws.

5 Summary of Tree Ensemble Methods

Trees are an attractive choice of weak learner for ensemble methods for a number of reasons, including their flexibility and ability to handle predictors of mixed types (qualitative as well as quantitative). We have now seen four approaches for fitting an ensemble of trees: bagging, random forests, boosting, and BART.

- In **bagging**, the trees are grown independently on random samples of the observations. Consequently, the trees tend to be quite similar to each other. Bagging can get caught in local optima and can fail to thoroughly explore the model space.
- In **random forests**, the trees are once again grown independently on random samples of the observations. However, each split on each tree is performed using a random subset of the features, thereby decorrelating the trees and leading to a more thorough exploration of model space relative to bagging.
- In **boosting**, we only use the original data and do not draw any random samples. The trees are grown successively, using a "slow" learning approach: each new tree is fit to the signal that is left over from the earlier trees, and shrunken down before it is used.
- In **BART**, we once again only make use of the original data, and we grow the trees successively. However, each tree is perturbed in order to avoid local minima and achieve a more thorough exploration of the model space.

Part IV

Support Vector Machines

1 Separating Hyperplanes

1.1 Binary Classification

- Consider binary outcomes $y_i \in \{-1, 1\}$. For SVMs it is more convenient to use the convention $y_i \in \{-1, 1\}$ rather than $y_i \in \{1, 2\}$.
- For logistic regression and linear discriminant analysis, we already discussed classification rules based on a hyperplane:
$$f(x) = \text{sgn}(\beta_0 + \beta' x).$$
- For SVMs we initially use such linear classification rules, but afterwards generalize to the nonlinear case via kernels.

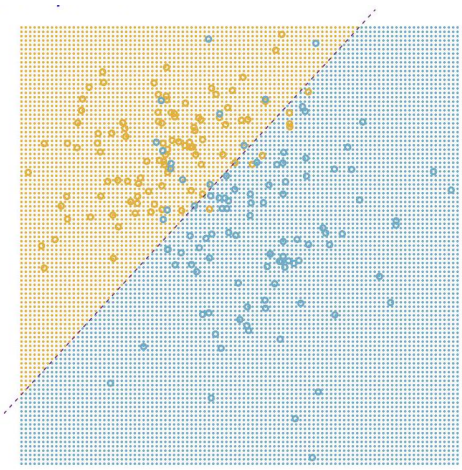


Figure 29: Example of a classification rule based on a separating hyperplane for $p = 2$.

1.2 Basic Linear Geometry

Define $h(x) := \beta_0 + \beta' x$. A hyperplane in $\mathbb{R}^p$ is the set

$$\mathcal{H} = \{x \in \mathbb{R}^p : h(x) = 0\}.$$

Without loss of generality we can impose the normalization $\|\beta\| = 1$.

- The vector β is normal (orthogonal) to $\mathcal{H}$.
- For any point $x \in \mathbb{R}^p$, the distance to $\mathcal{H}$ is given by $|h(x)|$.

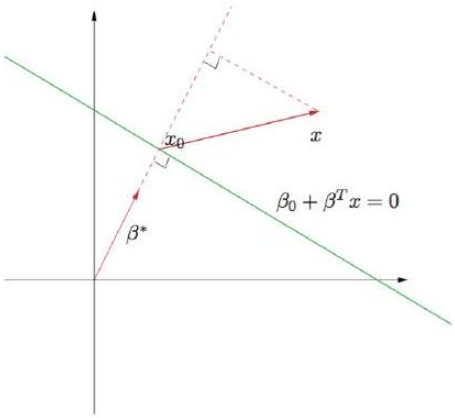


Figure 30: The linear algebra of a hyperplane (affine set). For $x_1, x_2 \in L$, $\beta'(x_1 - x_2) = 0$, so $\beta^* := \beta/\|\beta\|$ is the unit normal to L. For any $x_0 \in L$, $\beta' x_0 = -\beta_0$. The signed distance of any x to L is $\beta^{*\prime}(x - x_0) = (\beta' x + \beta_0)/\|\beta\| = h(x)/\|h'(x)\|$. Source: Hastie et al. (2009).

1.3 Separating Hyperplanes

- Consider the classification based on a hyperplane $\mathcal{H}$:

$$f(x) = \text{sgn}(\beta_0 + \beta' x).$$

- A correct classification is one such that $h(x_i)y_i > 0$ (equivalently, $f(x_i)y_i > 0$), for all $i = 1, \ldots, n$.
- Under classical separability, we can find a function such that $y_i h(x_i) > 0$ for all $i = 1, \ldots, n$.
- The larger the quantity $\min_i[y_i h(x_i)]$, the better the separation between the two classes.

1.4 Optimal Separating Hyperplane

This idea can be encoded in the following convex program:

$$\max_{\beta_0, \beta} M \quad \text{subject to} \quad y_i h(x_i) \geq M \ \text{ for each } i, \quad \|\beta\| = 1.$$

- We know that $y_i h(x_i) > 0 \Rightarrow f(x_i) = y_i$. Hence, larger $y_i h(x_i)$ indicates a "more confident" correct classification.

- For M to have meaning as a distance (the margin), we impose $\|\beta\| = 1$.

This program can equivalently be rewritten as

$$\min_{\beta_0,\beta} \frac{1}{2}\|\beta\|^2 \quad \text{subject to} \quad y_i h(x_i) \geq 1 \;\text{ for each } i.$$

If $\|\beta\| = 1$, then $y_i h(x_i) \geq \|\beta\| M$. For any β and β_0 satisfying these inequalities, any positively scaled multiples satisfy them too, so we can set $\|\beta\| = 1/M$. This is a convex optimization program: quadratic criterion with linear inequality constraints.

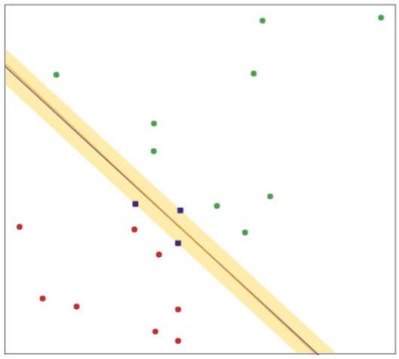


Figure 31: The shaded region delineates the maximum margin separating the two classes. Three support points lie on the boundary of the margin, and the optimal separating hyperplane (blue line) bisects the slab. The boundary found using logistic regression (red line) is very close. Source: Hastie et al. (2009).

Lagrangian formulation. We convert the constrained optimization problem into Lagrangian form:

$$\min_{\beta_0,\beta} \frac{1}{2}\|\beta\|^2 - \sum_{i=1}^{n} \alpha_i \left[y_i(x_i'\beta + \beta_0) - 1\right],$$

where $\alpha_i \geq 0$ are Lagrange multipliers chosen such that the constraints $y_i h(x_i) \geq 1$ are satisfied for the optimal β_0, β.

Taking derivatives with respect to β and β_0:

- $\beta = \sum_{i=1}^{n} \alpha_i y_i x_i$,
- $0 = \sum_{i=1}^{n} \alpha_i y_i$.

Substituting back into the Lagrangian gives the *dual objective*:

$$\sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{k=1}^{n} \alpha_i \alpha_k y_i y_k x_i' x_k,$$

subject to $\alpha_i \geq 0$. We maximize this dual objective over α_i.

Complementary slackness and support vectors. The Karush–Kuhn–Tucker (KKT) conditions require

$$\alpha_i \left[1 - y_i h(x_i)\right] = 0 \quad \text{for all } i.$$

This implies either:

- $\alpha_i = 0$, which happens when the constraint $y_i h(x_i) > 1$ is nonbinding, or
- $\alpha_i > 0$, which happens when the constraint $y_i h(x_i) = 1$ is binding.

The points (x_i, y_i) with $\hat{\alpha}_i > 0$ are called *support vectors*. All other points are irrelevant for the classification boundary. The resulting classifier is

$$\hat{f}(x) = \text{sgn}(x'\hat{\beta} + \hat{\beta}_0), \quad \text{where } \hat{\beta} = \sum_{i=1}^{n} \hat{\alpha}_i y_i x_i.$$

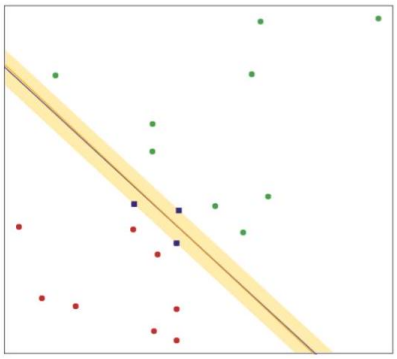


Figure 32: Support vectors are the points on the boundary of the margin (blue squares). The estimate $\hat{\beta}$ depends only on these support vectors. Source: Hastie et al. (2009).

2 Support Vector Classifier

2.1 Motivation

- In practice we cannot assume that the data are linearly separable. Thus the optimal separating hyperplane problem usually has no feasible solution.
- Even when the data are separable, a small change in the dataset can result in a large change in the optimal separating hyperplane:

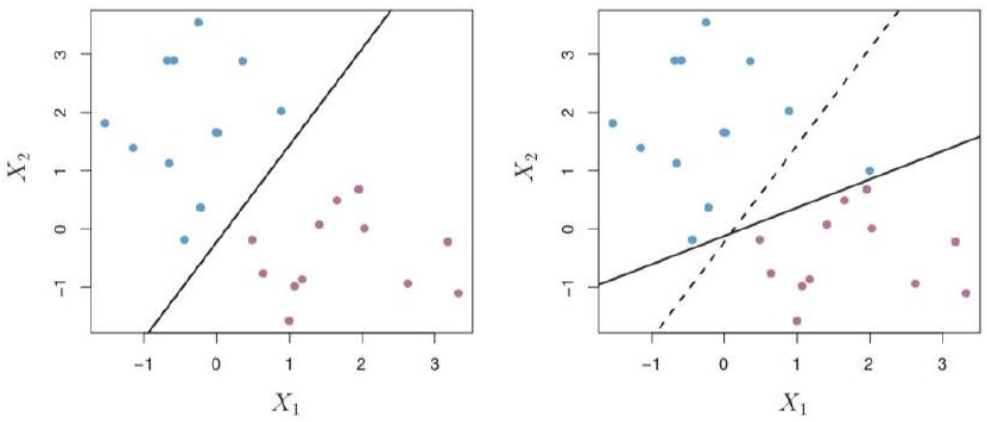


Figure 33: *Left:* Two classes shown in blue and purple, along with the maximal margin hyperplane. *Right:* An additional blue observation leads to a dramatic shift in the maximal margin hyperplane (solid line). The dashed line is the hyperplane from the left panel. Source: James et al. (2021).

2.2 Formulation with Slack Variables

We introduce slack variables $\boldsymbol{\xi} = (\xi_1, \ldots, \xi_n)$ that allow for overlap among the classes:

$$\max_{\beta_0, \beta, \xi_1, \ldots, \xi_n} M \quad \text{subject to} \quad y_i h(x_i) \geq M(1 - \xi_i), \quad \xi_i \geq 0, \quad \sum_{i=1}^{n} \xi_i \leq t, \quad \|\beta\| = 1.$$

- t is a tuning parameter that controls the total amount of slackness. The literature often denotes t by C.
- The separable case (optimal separating hyperplane) corresponds to $t = 0$.

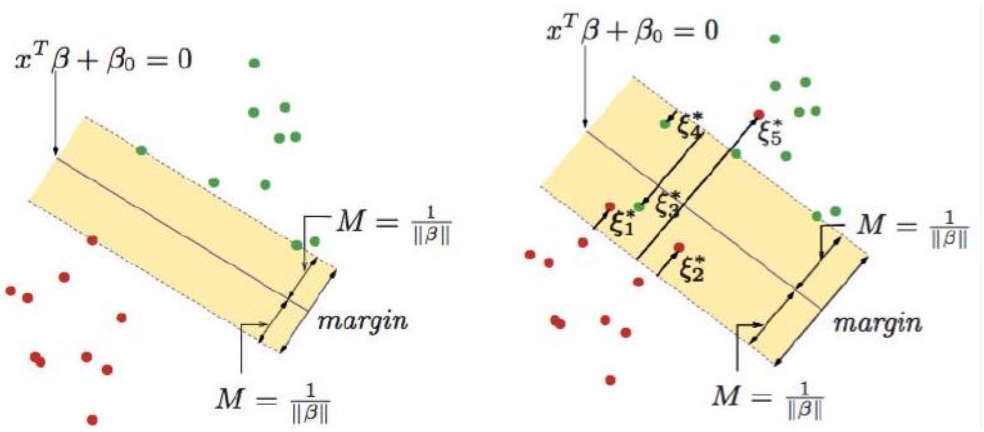
Equivalently:

$$\min_{\beta_0, \beta, \boldsymbol{\xi}} \frac{1}{2}\|\beta\|^2 \quad \text{subject to} \quad y_i h(x_i) \geq 1 - \xi_i, \quad \xi_i \geq 0, \quad \sum_{i=1}^{n} \xi_i \leq t.$$

2.3 Slack Variables

The slack variables provide insight into each observation's relationship to the margin:

- If $\xi_i = 0$: the observation is on the correct side of the margin.
- If $\xi_i \in (0, 1]$: the observation is on the incorrect side of the margin but still correctly classified.
- If $\xi_i > 1$: the observation is misclassified.



(a) Separable case (b) Nonseparable (overlap) case

Figure 34: *Left:* The decision boundary (solid line) with margin of width $2M = 2/\|\beta\|$. *Right:* Points labeled ξ_j^* are on the wrong side of their margin by amount $\xi_j^* = M\xi_j$. The margin is maximized subject to a total budget $\sum \xi_i \leq t$. Source: Hastie et al. (2009).

2.4 Lagrangian and Dual Formulation

Converting $\sum \xi_i \leq t$ to the Lagrangian:

$$\min_{\beta_0,\beta,\boldsymbol{\xi}} \frac{1}{2}\|\beta\|^2 + \lambda \sum_{i=1}^{n} \xi_i \quad \text{subject to} \quad y_i h(x_i) \geq 1 - \xi_i, \quad \xi_i \geq 0.$$

The full Lagrangian is

$$\frac{1}{2}\|\beta\|^2 + \lambda \sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \alpha_i \left[y_i(x_i'\beta + \beta_0) - (1 - \xi_i)\right] - \sum_{i=1}^{n} \gamma_i \xi_i.$$

The necessary conditions (taking derivatives) are:

- $\beta = \sum_{i=1}^{n} \alpha_i y_i x_i$ (unchanged from the separable case),
- $0 = \sum_{i=1}^{n} \alpha_i y_i$ (unchanged),
- $\alpha_i = \lambda - \gamma_i$ (more $\alpha_i > 0$ implies more support vectors).

Substituting, we again obtain the dual objective:

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{i'=1}^{n} \alpha_i \alpha_{i'} y_i y_{i'} x_i' x_{i'},$$

subject to the KKT conditions. The necessary condition $\beta = \sum_{i=1}^{n} \alpha_i y_i x_i$ implies estimators of the form

$$\hat{\beta} = \sum_{i=1}^{n} \hat{\alpha}_i y_i x_i, \qquad \hat{\beta}' x = \sum_{i=1}^{n} \hat{\alpha}_i y_i \, x_i' x.$$

2.5 Tuning Parameter

- If $t = 0$, we tolerate no margin violations. If $t > 0$, then no more than t observations can be misclassified.
- A large t allows many violations, so many observations enter into $\hat{\beta}$.
- Thus t calibrates a bias–variance trade-off. In practice, t is selected via cross-validation.

2.6 Summary

- A support vector machine solves the classification problem by constructing a hyperplane in the (high-dimensional) space of predictors.
- Intuitively, a good separation is achieved by the hyperplane with the largest distance to the nearest training-data point of any class, since the larger the margin, the lower the generalization error.
- Perfect classification by the hyperplane may not be possible (or desirable), which is why SVMs introduce slackness parameters that allow some observations to be misclassified. This increases the robustness of the prediction.

3 Kernel SVMs

3.1 Nonlinear Classifiers

- Sometimes linear decision boundaries are insufficient. We may need to include polynomial effects or other transformations.
- We can account for such nonlinearities by enlarging the feature space via basis expansions. Choose a vector of (nonlinear) transformations of $x_i \in \mathbb{R}^p$:

$$g(x_i) = (g_1(x_i), g_2(x_i), \ldots, g_M(x_i))' \in \mathbb{R}^M.$$

For example, with $p = 2$:

$$g(x_i) = (x_{i1}, x_{i2}, x_{i1}^2, x_{i2}^2, x_{i1}x_{i2})' \in \mathbb{R}^5.$$

- Then consider classifiers of the form $f(x) = \beta_0 + \beta' g(x)$. Everything from the linear case remains valid; simply replace x by $g(x)$ everywhere.

3.2 The Kernel Trick

- What if we want to choose $g(x)$ to be very high-dimensional (or even infinite-dimensional)?
- As a simple example, consider the ridge regression prediction:

$$\hat{y} = X(X'X + \lambda I_p)^{-1} X'y.$$

If p is large, computations with $X'X + \lambda I_p$ ($p \times p$ matrix) may be daunting. Instead, rewrite as[1]

$$\hat{y} = (XX' + \lambda I_n)^{-1} XX'y.$$

The matrix to invert is now only $n \times n$, which can be much simpler computationally.

- We only need to compute inner products of the observations (XX'). This argument also applies when x represents transformations of the original features: we only need to define inner products in the transformed space, without actually computing $g(x)$.

3.3 Kernel Support Vector Machines

Recall the dual objective for the support vector classifier:

$$\sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{k=1}^{n} \alpha_i \alpha_k y_i y_k \, x_i' x_k.$$

After replacing x with $g(x)$, define the *kernel function* $k(x_i, x_k) := g(x_i)'g(x_k)$. The dual objective becomes

$$\sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{k=1}^{n} \alpha_i \alpha_k y_i y_k \, k(x_i, x_k).$$

The solution function is

$$h(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i y_i \, k(x_i, x).$$

[1] Here we use the push-through identity: https://en.wikipedia.org/wiki/Woodbury_matrix_identity.

We need to choose a kernel that is symmetric and positive definite. Common choices include:

- Polynomial: $k(x, \tilde{x}) = (1 + x'\tilde{x})^d$.
- Gaussian (radial basis function): $k(x, \tilde{x}) = \exp\left(-\frac{\|x - \tilde{x}\|^2}{2\sigma^2}\right)$.

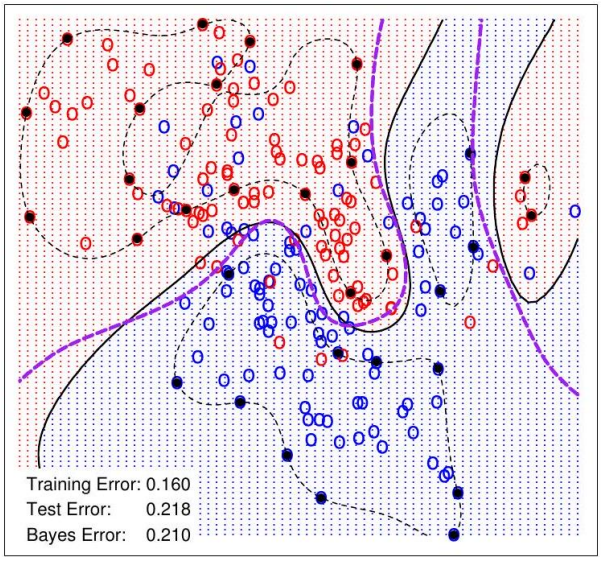


Figure 35: Kernel SVM on a two-class mixture example. The nonlinear decision boundary captures complex class structure. Source: Hastie et al. (2009).

4 SVMs via Penalization

SVMs can also be derived from penalized loss methods. Recall the support vector classifier:

$$\min_{\beta_0, \beta, \boldsymbol{\xi}} \frac{1}{2}\|\beta\|^2 + \lambda \sum_{i=1}^{n} \xi_i \quad \text{subject to} \quad y_i h(x_i) \geq 1 - \xi_i, \quad \xi_i \geq 0.$$

Defining $[z]_+ := \max(z, 0)$, this is equivalent to:

$$\min_{\beta, \beta_0} \sum_{i=1}^{n} [1 - y_i h(x_i)]_+ + \tau \|\beta\|^2,$$

where $2\lambda = 1/\tau$.

4.1 Hinge Loss

The loss component is the *hinge loss*:

$$\ell(x, y) = [1 - y\, h(x)]_+.$$

The hinge loss approximates the zero-one loss function underlying classification, with one major advantage: convexity.

4.2 Surrogate Losses

It is tempting to directly minimize the misclassification loss $\sum_{i=1}^{n} \mathbb{1}(y_i \neq \hat{f}(x_i)) + \tau\|\beta\|^2$, but this is nonconvex in $u = h(x)y$. A common strategy is *convex relaxation* with a surrogate loss function:

- Hinge loss: $[1 - y\, h(x)]_+$.
- Logistic loss: $\log(1 + e^{-y\, h(x)})$.

For a deeper discussion: See Bartlett et al. (2006) and Lugosi and Vayatis (2004).

Part V

Deep Learning

1 Neural Networks

Neural networks rose to prominence in the late 1980s, motivated by the analogy to biological neural computation and popularized through the annual NeurIPS (formerly NIPS) conferences. After a period of relative decline—overshadowed by SVMs, boosting, and random forests—they resurfaced around 2010 under the banner of *deep learning*, fueled by new architectures, larger datasets, and more powerful hardware (James et al., 2021). What distinguishes neural networks from other nonlinear methods is their particular *structural* approach to building the prediction function $f(X)$.

1.1 Single-Layer Neural Networks

- A neural network takes an input vector $X = (X_1, X_2, \ldots, X_p)'$ and constructs a nonlinear function $f(X)$ to predict the response Y. The simplest architecture is a *feed-forward neural network* (also called a *fully connected* network) with a single hidden layer.

- The model is built in two stages. First, K *activations* in the hidden layer are computed as nonlinear transformations of linear combinations of the inputs:

$$A_k = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j\right), \quad k = 1, \ldots, K, \tag{9}$$

  where $g(\cdot)$ is a nonlinear *activation function* specified in advance, and the w_{kj} are unknown *weights* (parameters). The constant w_{k0} is called the *bias* in the deep learning literature (not to be confused with statistical bias). Each A_k can be viewed as a learned basis function $h_k(X)$.

- Second, the output layer computes a linear combination of the activations:

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k. \tag{10}$$

  All parameters $\beta_0, \ldots, \beta_K$ and $w_{10}, \ldots, w_{Kp}$ are estimated from data.

- The architecture has three layers: the *input layer* $(X_1, \ldots, X_p)$, the *hidden layer* $(A_1, \ldots, A_K)$,

and the *output layer* ($f(X)$). The hidden layer is so called because the activations A_k are not directly observed—they are intermediate quantities derived from the data during training.

- A single neuron with the sigmoid activation and cross-entropy loss recovers *logistic regression*: the sigmoid neuron computes $\hat{y} = g(w_0 + \sum_j w_j X_j)$ where $g(z) = 1/(1+e^{-z})$, which is exactly the logistic regression model.

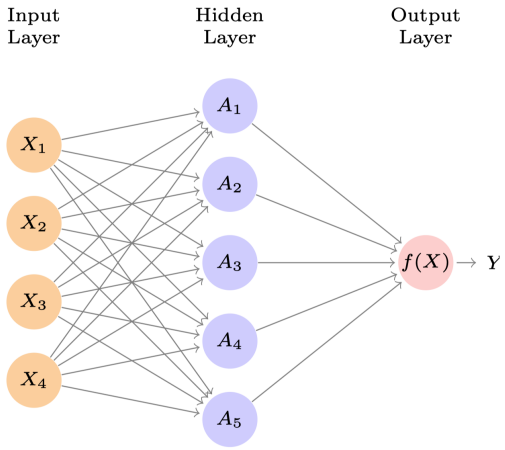


FIGURE 10.1. *Neural network with a single hidden layer. The hidden layer computes activations $A_k = h_k(X)$ that are nonlinear transformations of linear combinations of the inputs $X_1, X_2, \ldots, X_p$. Hence these A_k are not directly observed. The functions $h_k(\cdot)$ are not fixed in advance, but are learned during the training of the network. The output layer is a linear model that uses these activations A_k as inputs, resulting in a function $f(X)$.*

Figure 36: Neural network with a single hidden layer. The hidden layer computes activations $A_k = h_k(X)$ that are nonlinear transformations of linear combinations of the inputs $X_1, X_2, \ldots, X_p$. The output layer is a linear model that uses these activations A_k as inputs, resulting in a function $f(X)$. Source: James et al. (2021), Figure 10.1.

Remark 6. Combining (9) and (10), the model $f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k \, g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j)$ can be viewed as a *nonlinear regression* model, or equivalently as a *projection pursuit* model with learned basis functions. The activations $A_k = h_k(X)$ play the role of basis functions, but unlike splines or polynomial bases, they are not fixed in advance—they are learned from the data.

1.2 Activation Functions

The activation function $g(\cdot)$ injects nonlinearity into the network. Without it, the composition of linear functions would collapse into a single linear model. Common choices include:

- **Sigmoid:** $g(z) = \dfrac{e^z}{1+e^z} = \dfrac{1}{1+e^{-z}}$, which maps $\mathbb{R}$ to $(0,1)$. This is the same function used in logistic regression. Its gradient is $g'(z) = g(z)(1-g(z))$. The output can be interpreted as a neuron "firing" (close to 1) or remaining "silent" (close to 0).

- **Tanh:** $g(z) = \tanh(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}} = 2\,\sigma(2z) - 1$, where σ is the sigmoid function. It maps $\mathbb{R}$ to $(-1,1)$ and is centered at zero, which often leads to faster convergence than the sigmoid. Its gradient is $g'(z) = 1 - \tanh^2(z)$.

- **ReLU** (Rectified Linear Unit): $g(z) = (z)_+ = \max(z, 0)$. The preferred choice in modern deep learning. ReLU is computationally efficient and does not saturate for large positive inputs. Its piecewise linearity makes gradient computation straightforward: $g'(z) = \mathbb{1}(z > 0)$.

- **Leaky ReLU:** $g(z) = \max(z, \alpha z)$ for a small $\alpha > 0$ (e.g., $\alpha = 0.01$). Unlike ReLU, it allows a small gradient for negative inputs, which can help mitigate the "dying ReLU" problem where neurons become permanently inactive.

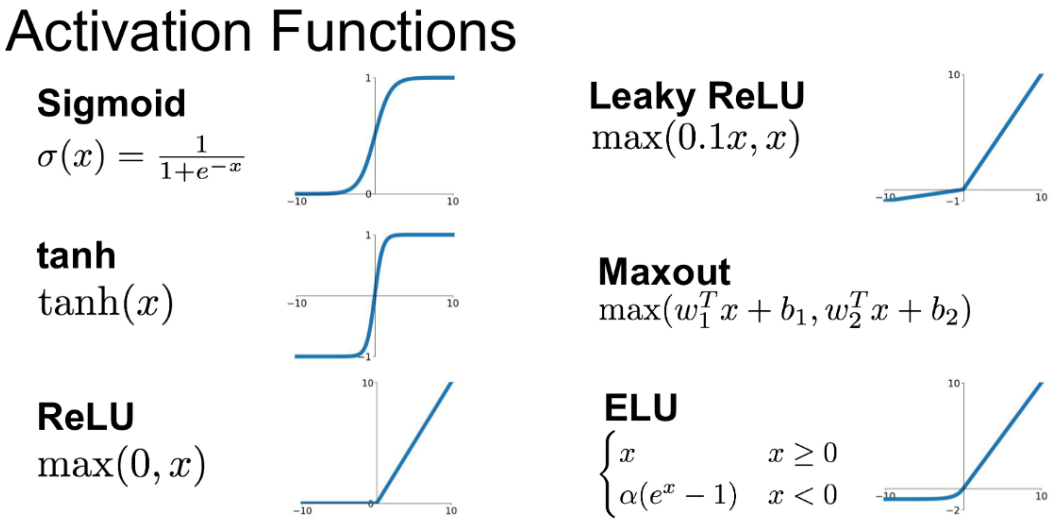


Figure 37: Common activation functions used in neural networks: sigmoid, tanh, ReLU, Leaky ReLU, Maxout, and ELU.

Why nonlinearity matters. Consider $p = 2$ inputs, $K = 2$ hidden units with $g(z) = z^2$, and specific parameter values $\beta_0 = 0$, $\beta_1 = 1/4$, $\beta_2 = -1/4$, $w_{10} = 0$, $w_{11} = w_{12} = 1$, $w_{20} = 0$, $w_{21} = 1$, $w_{22} = -1$. Then

$$h_1(X) = (X_1 + X_2)^2, \quad h_2(X) = (X_1 - X_2)^2,$$

and

$$f(X) = \tfrac{1}{4}(X_1 + X_2)^2 - \tfrac{1}{4}(X_1 - X_2)^2 = X_1 X_2.$$

The sum of two nonlinear transformations of linear functions recovers an *interaction effect*. Sigmoid or ReLU activations provide more flexibility than any fixed polynomial basis (James et al., 2021).

Theorem 1 (Universal Approximation (Cybenko, 1989; Hornik et al., 1989)). *A feed-forward network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of $\mathbb{R}^p$ to arbitrary accuracy, provided the activation function is non-constant, bounded, and continuous (e.g., the sigmoid).*

Remark 7. The Universal Approximation Theorem guarantees *existence* of a good approximation, but says nothing about how to *find* it or how many neurons are needed. In practice, deeper networks with multiple hidden layers of moderate width are often more effective than a single very wide hidden layer.

1.3 Multilayer Neural Networks

Modern neural networks typically have more than one hidden layer. While a single wide layer can theoretically approximate any continuous function, the learning task is often made much easier with multiple layers of moderate size, each building progressively more abstract features.

- The first hidden layer computes activations as in (9):

$$A_k^{(1)} = g\left(w_{k0}^{(1)} + \sum_{j=1}^{p} w_{kj}^{(1)} X_j\right), \quad k = 1, \ldots, K_1.$$

- The second hidden layer uses the first-layer activations as inputs:

$$A_\ell^{(2)} = g\left(w_{\ell 0}^{(2)} + \sum_{k=1}^{K_1} w_{\ell k}^{(2)} A_k^{(1)}\right), \quad \ell = 1, \ldots, K_2.$$

  Each $A_\ell^{(2)} = h_\ell^{(2)}(X)$ is a function of X through the chain of transformations. Through this cascade, the network builds up complex representations of the input.

- In matrix notation, let $\mathbf{W}^{(1)}$ denote the weight matrix from the input to the first hidden layer, and $\mathbf{W}^{(2)}$ from the first to the second hidden layer. Then

$$\boldsymbol{A}^{(1)} = g(\mathbf{W}^{(1)} X + \boldsymbol{b}^{(1)}), \qquad \boldsymbol{A}^{(2)} = g(\mathbf{W}^{(2)} \boldsymbol{A}^{(1)} + \boldsymbol{b}^{(2)}),$$

  where $\boldsymbol{b}^{(\ell)}$ are bias vectors and $g(\cdot)$ is applied element-wise.

- For a *regression* problem, the output layer is simply a linear combination:

$$f(X) = \beta_0 + \sum_{\ell=1}^{K_2} \beta_\ell A_\ell^{(2)}.$$

  The model is typically fit by minimizing squared-error loss: $\sum_{i=1}^{n}(y_i - f(x_i))^2$.

- For a *classification* problem with M classes, we compute M linear models

$$Z_m = \beta_{m0} + \sum_{\ell=1}^{K_2} \beta_{m\ell} A_\ell^{(2)}, \quad m = 1, \ldots, M,$$

  and apply the *softmax* function to obtain class probabilities:

$$f_m(X) = \Pr(Y = m \mid X) = \frac{e^{Z_m}}{\sum_{m'=1}^{M} e^{Z_{m'}}}. \tag{11}$$

- The model is then trained by minimizing the *cross-entropy* (negative log-likelihood):

$$-\sum_{i=1}^{n}\sum_{m=1}^{M} y_{im} \log f_m(x_i), \tag{12}$$

  where $y_{im} = \mathbb{1}(y_i = m)$ is a one-hot encoding of the class labels. For binary classification ($M = 2$), this reduces to the familiar logistic regression loss $-\sum_{i=1}^{n}[y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)]$.

1.4 Fitting Neural Networks

Let θ denote the vector of all parameters in the network (weights and biases across all layers). We seek to minimize an objective of the form

$$Q(\theta) = \sum_{i=1}^{n} \ell(y_i, f_\theta(x_i)),$$

where ℓ is a loss function—squared error for regression, cross-entropy (12) for classification. This objective is *non-convex* in θ due to the nested nonlinear structure of the network, so there are in general multiple local minima. Two key strategies are employed: (i) *slow learning* via gradient descent, stopped when overfitting is detected; and (ii) *regularization* via penalties on the parameters.

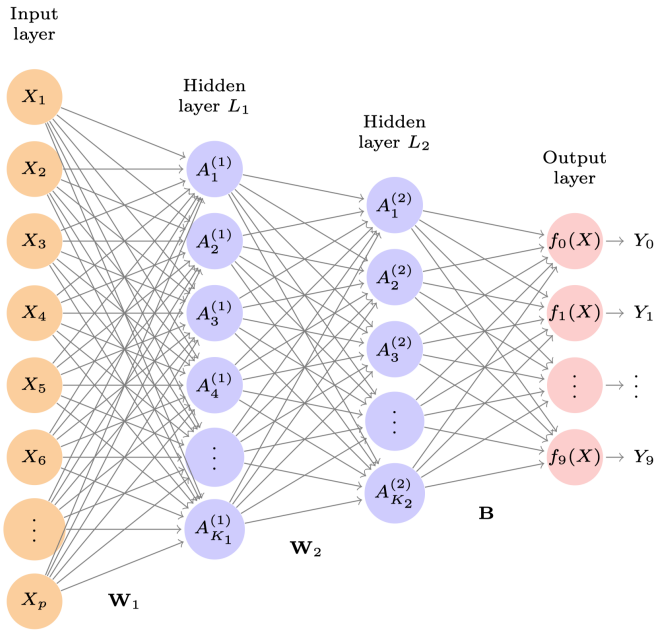


FIGURE 10.4. *Neural network diagram with two hidden layers and multiple outputs, suitable for the* MNIST *handwritten-digit problem. The input layer has* $p = 784$ *units, the two hidden layers* $K_1 = 256$ *and* $K_2 = 128$ *units respectively, and the output layer* 10 *units. Along with intercepts (referred to as* biases *in the deep-learning community) this network has 235,146 parameters (referred to as weights).*

Figure 38: Neural network with two hidden layers and multiple outputs, suitable for the MNIST handwritten-digit problem. The input layer has $p = 784$ units, the two hidden layers $K_1 = 256$ and $K_2 = 128$ units, and the output layer 10 units. Source: James et al. (2021), Figure 10.4.

Gradient descent. The basic algorithm is:

1. Initialize θ^0 (typically randomly; initialization matters—see below).
2. For $t = 0, 1, 2, \ldots$, update:
$$\theta^{t+1} = \theta^t - \rho \nabla Q(\theta^t), \tag{13}$$
where $\rho > 0$ is the *learning rate* and $\nabla Q(\theta^t)$ is the gradient of Q evaluated at the current parameters.
3. Stop when the objective ceases to decrease or when a validation metric deteriorates.

For a small enough learning rate, each step is guaranteed to decrease the objective. A first-order Taylor expansion gives

$$Q(\theta^{t+1}) \approx Q(\theta^t) + \nabla Q(\theta^t)' (\theta^{t+1} - \theta^t) = Q(\theta^t) - \rho \|\nabla Q(\theta^t)\|^2,$$

so Q decreases as long as the gradient is nonzero. For a convex objective with Lipschitz-continuous gradients, achieving $Q(\theta^t) - Q(\theta^*) < \varepsilon$ requires $O(1/\varepsilon)$ iterations (a linear convergence rate). Stronger assumptions (e.g., strong convexity) or second-order methods (e.g., Newton's method) can improve this rate. The objective in neural networks is non-convex, so we can generally only hope to reach a good local minimum.

Backpropagation. Computing $\nabla Q(\theta)$ efficiently is central to training neural networks. *Backpropagation* (Rumelhart et al., 1986) is a systematic application of the chain rule of differentiation that computes the gradient of the loss with respect to all parameters in one forward pass and one backward pass through the network.

To illustrate, consider a single-neuron logistic regression model with cross-entropy loss:

$$z = w_0 + \sum_{j=1}^{p} w_j x_j, \qquad \hat{y} = g(z) = \frac{e^z}{1 + e^z}, \qquad \ell(y, \hat{y}) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})]\,.$$

In the *forward pass*, we compute $z \to \hat{y} \to \ell$ sequentially. In the *backward pass*, we apply the chain rule to obtain the gradient with respect to each weight:

$$\frac{\partial \ell}{\partial w_j} = \underbrace{\frac{\partial \ell}{\partial \hat{y}}}_{\text{output layer}} \cdot \underbrace{\frac{\partial \hat{y}}{\partial z}}_{g'(z)} \cdot \underbrace{\frac{\partial z}{\partial w_j}}_{x_j},$$

where

$$\frac{\partial \ell}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}}, \qquad g'(z) = g(z)(1 - g(z)), \qquad \frac{\partial z}{\partial w_j} = x_j.$$

Each derivative decomposes the residual across the network's layers—a process of attributing "credit" for the prediction error to each parameter.

In a general deep network, the same principle applies recursively. For the single hidden-layer model (9)–(10) with squared-error loss, writing $z_{ik} = w_{k0} + \sum_j w_{kj} x_{ij}$ and $R_i = \frac{1}{2}(y_i - f_\theta(x_i))^2$, the gradients are:

$$\frac{\partial R_i}{\partial \beta_k} = -(y_i - f_\theta(x_i)) \cdot g(z_{ik}), \tag{14}$$

$$\frac{\partial R_i}{\partial w_{kj}} = -(y_i - f_\theta(x_i)) \cdot \beta_k \cdot g'(z_{ik}) \cdot x_{ij}. \tag{15}$$

Both expressions contain the residual $y_i - f_\theta(x_i)$, and through the chain rule a fraction of that residual is attributed to each parameter. In vectorized form, the error signal propagates backwards through the layers according to

$$\boldsymbol{\delta}^{(\ell)} = g'(\boldsymbol{z}^{(\ell)}) \odot \left((\mathbf{W}^{(\ell)})' \, \boldsymbol{\delta}^{(\ell+1)} \right),$$

where $\odot$ denotes the Hadamard (element-wise) product, and the gradient of the weight matrix at layer ℓ is the outer product $\nabla_{\mathbf{W}^{(\ell)}} = \boldsymbol{\delta}^{(\ell+1)} (\boldsymbol{A}^{(\ell)})'$. The key insight is that $\boldsymbol{\delta}^{(\ell)}$ depends on $\boldsymbol{\delta}^{(\ell+1)}$, so the gradients are computed recursively from the output layer back to the input—hence the name "backpropagation."

Stochastic gradient descent. When n is large, computing the full gradient

$$\nabla Q(\theta) = \sum_{i=1}^{n} \nabla_\theta \ell(y_i, f_\theta(x_i))$$

at each step is expensive. *Stochastic gradient descent* (SGD) approximates the gradient using a randomly selected *mini-batch* $\mathcal{B} \subset \{1, \ldots, n\}$ of size B:

$$\theta^{t+1} = \theta^t - \rho \sum_{i \in \mathcal{B}_t} \nabla_\theta \ell(y_i, f_\theta(x_i)). \tag{16}$$

In practice, the training data are randomly shuffled and partitioned into mini-batches of size B (typically 32–256). One complete pass through all mini-batches—i.e., through the entire training set—is called an *epoch*. SGD introduces noise into the gradient estimate, which can help escape shallow local minima and has an implicit regularization effect.

Several refinements of SGD are widely used:

- **Momentum:** maintains a running average of past gradients to accelerate convergence and

dampen oscillations:

$$v^{t+1} = \mu\, v^t - \rho\, \nabla_{\mathcal{B}} Q(\theta^t), \qquad \theta^{t+1} = \theta^t + v^{t+1},$$

where $\mu \in [0, 1)$ is the momentum coefficient (e.g., $\mu = 0.9$).

- **Adam** (Kingma and Ba, 2015): an adaptive method that maintains per-parameter learning rates based on running estimates of the first and second moments of the gradient. Adam is the default optimizer in most modern deep learning frameworks.

1.5 Regularization

Neural networks are highly over-parameterized. For instance, a network with two hidden layers of sizes $K_1 = 256$ and $K_2 = 128$ applied to MNIST data ($p = 784$, $M = 10$) has over 235,000 parameters—roughly four times the number of training observations ($n = 60{,}000$). Without regularization, severe overfitting is inevitable.

Ridge and lasso penalties. Analogous to ridge and lasso regression, we can augment the objective with a penalty on the weights:

$$Q_\lambda(\theta) = Q(\theta) + \lambda \sum_j \theta_j^2 \qquad \text{(ridge)}, \qquad \text{or} \qquad Q_\lambda(\theta) = Q(\theta) + \lambda \sum_j |\theta_j| \qquad \text{(lasso)}.$$

Typically, different regularization strengths λ are used for different layers, and bias terms are not penalized.

Initialization. The choice of initial weights matters significantly. If weights are initialized too large, activations saturate (for sigmoid/tanh) and gradients vanish; too small, and the signal dies out. *Xavier initialization* (Glorot and Bengio, 2010) sets the weights at layer ℓ as

$$W^{(\ell)} \sim U\left[-\sqrt{\frac{6}{n_\ell + n_{\ell+1}}},\ \sqrt{\frac{6}{n_\ell + n_{\ell+1}}}\right],$$

where n_ℓ and $n_{\ell+1}$ are the input and output dimensions of the layer. This scheme preserves the variance of activations and backpropagated gradients across layers, promoting stable training.

Dropout. A powerful regularization technique introduced by Srivastava et al. (2014). During each training step, each neuron is randomly "dropped" (its activation set to zero) with probability ε, independently across neurons and training iterations. The surviving activations are rescaled

by $1/(1-\varepsilon)$ to maintain their expected magnitude. Formally, the effective activation function becomes

$$\tilde{g}(z) = \begin{cases} 0 & \text{with probability } \varepsilon, \\ g(z)/(1-\varepsilon) & \text{with probability } 1-\varepsilon. \end{cases}$$

During testing, the full network is used without dropout. Dropout prevents neurons from becoming overly specialized and can be viewed as an approximate ensemble method—training exponentially many sub-networks simultaneously. The idea has roots in earlier work by Bishop (1995) on training with noise.

Early stopping. Even without explicit penalties, SGD can be stopped before convergence as a form of regularization. As training progresses, the training loss typically decreases monotonically, but the validation loss eventually begins to rise—a signal of overfitting. Stopping at the point of minimum validation loss is called *early stopping*, and it is one of the most common regularization strategies in practice.

1.6 Double Descent

The classical bias–variance tradeoff predicts that test error follows a U-shape as model complexity increases: too simple and we underfit, too complex and we overfit. However, a more nuanced picture emerges in the highly over-parameterized regime common in deep learning.

- At the *interpolation threshold*—where the number of parameters equals the number of training observations—the model can fit the training data exactly, and test error spikes dramatically.
- Beyond the interpolation threshold, as the number of parameters continues to increase, test error can *decrease again*, producing a "double descent" curve. This occurs because among the many interpolating solutions, optimization algorithms (particularly SGD with implicit regularization) tend to select "smooth" solutions with small parameter norms.
- The double descent phenomenon does not contradict the bias–variance tradeoff. Rather, the notion of model "flexibility" changes beyond the interpolation threshold: the minimum-norm interpolating solution with $d \gg n$ parameters can have *lower* variance than the unique interpolating solution with exactly $d = n$ parameters.
- In practice, we typically do not rely on double descent. Regularization techniques (ridge, dropout, early stopping) generally achieve better test performance than unregularized interpolation. Double descent is most pronounced in settings with high signal-to-noise ratios, such as image recognition.

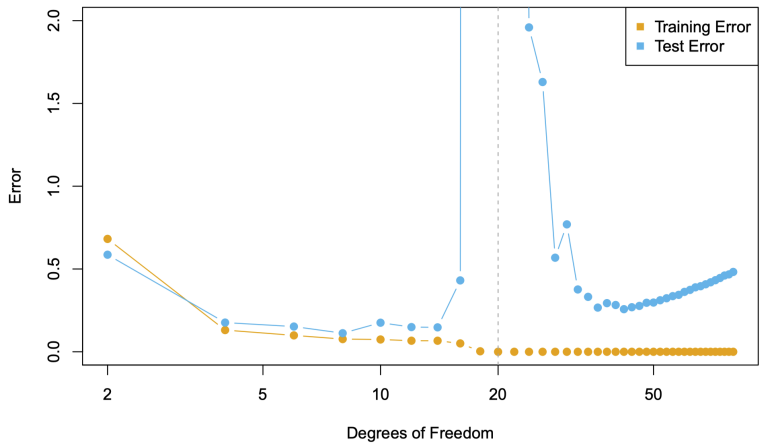


FIGURE 10.20. *Double descent phenomenon, illustrated using error plots for a one-dimensional natural spline example. The horizontal axis refers to the number of spline basis functions on the log scale. The training error hits zero when the degrees of freedom coincides with the sample size $n = 20$, the "interpolation threshold", and remains zero thereafter. The test error increases dramatically at this threshold, but then descends again to a reasonable value before finally increasing again.*

Figure 39: Double descent phenomenon, illustrated using error plots for a one-dimensional natural spline example. The training error hits zero when the degrees of freedom coincides with the sample size $n = 20$, the "interpolation threshold," and remains zero thereafter. The test error increases dramatically at this threshold, but then descends again to a reasonable value before finally increasing again. Source: James et al. (2021), Figure 10.20.

1.7 When to Use Deep Learning

- **Occam's Razor:** When faced with several methods that give roughly equivalent performance, prefer the simplest. On tabular datasets with moderate sample sizes, well-tuned linear models, lasso, or gradient-boosted trees often match or exceed neural networks, with far less effort and greater interpretability.
- Deep learning is most attractive when:
  - the training sample size is *very large*,
  - the input has rich structure (images, text, audio, time series), and
  - *interpretability* of the model is not a primary concern.
- For data like images and sequences, specialized architectures (CNNs, RNNs, Transformers) exploit domain-specific inductive biases and dramatically outperform generic methods. These architectures are discussed in the following sections.

2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a specialized family of neural networks designed for data with *spatial structure*, most notably images. They have produced spectacular results in image classification and are the architecture behind many computer vision successes. CNNs mimic to some degree how humans classify images: by recognizing specific local features or patterns *anywhere* in the image that distinguish each particular object class (James et al., 2021).

The key idea is hierarchical feature extraction. The network first identifies *low-level features* such as edges and patches of color. These are then combined into *higher-level features* such as eyes, ears, or textures. Eventually, the presence or absence of these higher-level features determines the class probabilities. Two specialized layer types—*convolution layers* and *pooling layers*—make this possible.

2.1 Convolution Layers

A convolution layer is made up of a collection of *convolution filters*, each of which is a small template that detects whether a particular local feature is present in the image.

- Consider a simple example: a 4×3 image and a 2×2 filter. The *convolution* operation slides the filter across every 2×2 sub-region of the image, computes the element-wise product, and sums the results. For instance, with image entries $a, b, \ldots, l$ and filter weights $\alpha, \beta, \gamma, \delta$:

$$\text{Convolved image} = \begin{pmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{pmatrix}.$$

  If a 2×2 patch of the image resembles the filter, the corresponding entry in the convolved image will be large; otherwise it will be small. The convolved image thus *highlights regions* of the original image that match the filter pattern.

- In practice, convolution filters are small $\ell_1 \times \ell_2$ arrays (e.g., 3×3), and the filter weights are *learned* during training—they are not fixed in advance. A single convolution layer uses a *bank* of K different filters to detect a variety of edges, textures, and shapes.

- For a color image with three channels (red, green, blue), each convolution filter also has three channels. The results of convolving across all three channels are summed to form a single two-dimensional output *feature map*. Using K different filters produces K output feature maps, which together form a three-dimensional *feature map* with K channels.

- The convolution filters can be viewed as the weights connecting the input to a hidden layer, but with two important structural constraints:
  - *Locality:* each output unit depends only on a small patch of the input (the filter's receptive field), so most weights are structurally zero.
  - *Weight sharing:* the same filter weights are reused across all spatial positions, which drastically reduces the number of parameters.
- After convolution, a ReLU activation function is typically applied element-wise to the output feature maps.

2.2 Pooling Layers

A *pooling layer* downsamples the feature maps to create a more compact representation. The most common form is *max pooling*: each non-overlapping 2×2 block of pixels is replaced by its maximum value. For example:

$$\text{Max pool} \begin{pmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 3 & 5 \\ 2 & 4 \end{pmatrix}.$$

- Max pooling reduces the spatial dimensions by a factor of two in each direction, cutting the number of pixels by a factor of four.
- It also provides a degree of *location invariance*: as long as a large activation appears somewhere in the 2×2 block, the pooled output registers it, regardless of the exact pixel position.

2.3 CNN Architecture

A deep CNN alternates convolution layers and pooling layers, progressively extracting more abstract features while reducing spatial resolution:

1. **Input.** A color image is represented as a three-dimensional array (feature map) of size $H \times W \times 3$, where the third axis indexes the RGB channels.
2. **Convolution + ReLU.** A bank of K learned filters is convolved with the input to produce K output feature maps (with padding to preserve spatial dimensions if desired). ReLU is applied element-wise.

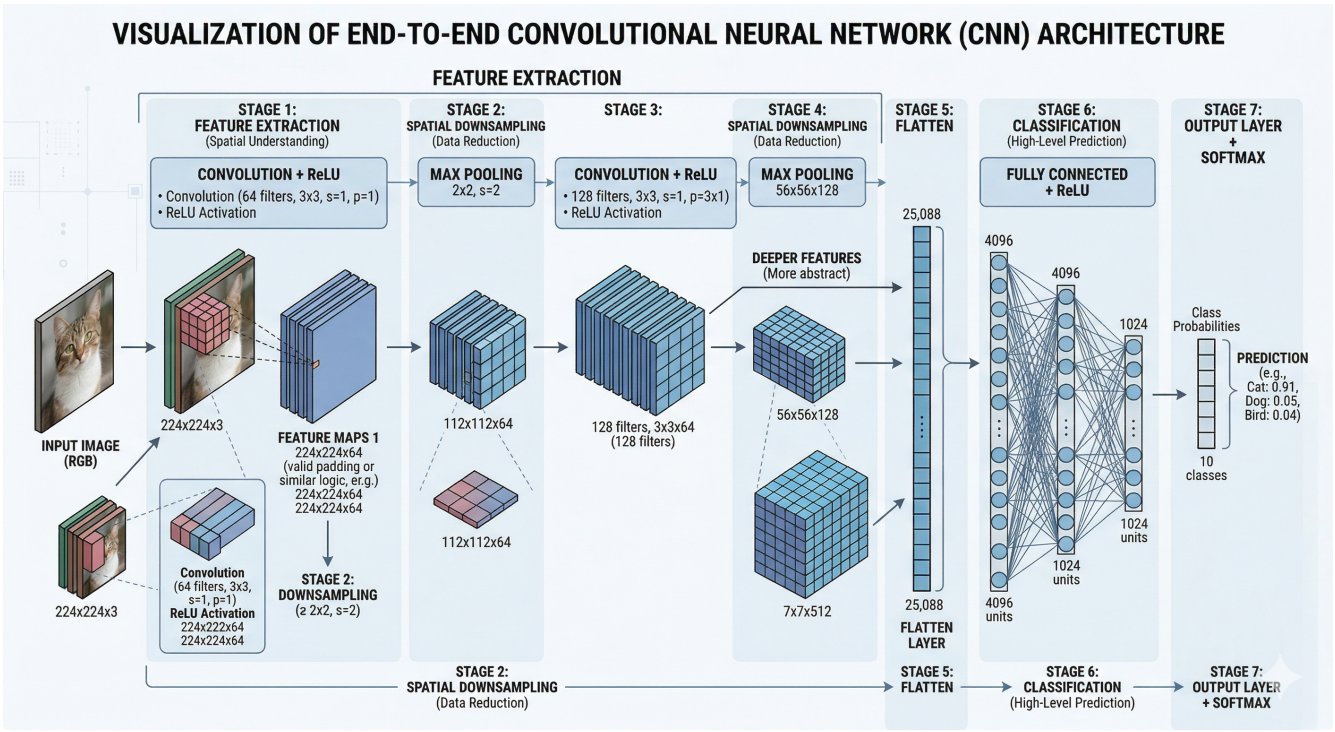


Figure 40: Visualization of an end-to-end CNN architecture for image classification. The network alternates convolution + ReLU stages with max-pooling stages to extract progressively more abstract features, then flattens the final feature maps and passes them through fully connected layers before producing class probabilities via softmax. Source: Gemini.

3. **Pooling.** A max-pool layer halves each spatial dimension, reducing the feature map from $H \times W$ to $H/2 \times W/2$ while keeping the K channels.

4. **Repeat.** Steps 2 and 3 are repeated multiple times. Subsequent convolution layers treat the multi-channel output of the previous layer as their input. Since the spatial dimensions shrink after each pool layer, the number of filters is usually *increased* to compensate, so the feature maps become spatially smaller but deeper.

5. **Flatten.** After the final pooling layer, the three-dimensional feature map is *flattened* into a one-dimensional vector.

6. **Fully connected layers.** The flattened vector feeds into one or more fully connected hidden layers (as in a standard neural network) to combine the abstract features for classification.

7. **Output.** A softmax activation produces class probabilities over the M classes.

2.4 Data Augmentation

An important technique for image classification is *data augmentation*. Each training image is replicated many times, with each copy randomly distorted in ways that do not affect human recognition: small rotations, horizontal/vertical shifts, zooming, shearing, and horizontal flips.

- At face value, this increases the effective training set size. More fundamentally, it acts as a form of *regularization*: by building a cloud of images around each original image—all with the same label—the model is discouraged from memorizing pixel-level details.

- Augmentation integrates naturally with SGD: distortions are applied on the fly to each mini-batch, so no extra storage is required.

2.5 Transfer Learning and Pretrained Models

Training a deep CNN from scratch requires massive datasets and computation. A powerful alternative is *transfer learning*: reuse the convolutional layers of a network that was pretrained on a large corpus (e.g., ImageNet) and retrain only the final fully connected layers on the new task.

- The intuition is that the early convolutional layers learn general-purpose visual features (edges, textures, shapes) that transfer well across different image classification problems.

- The pretrained convolutional weights are *frozen* (not updated during training), and only the task-specific output layers are learned. This process, called *weight freezing* or *fine-tuning*, requires far less data and computation than training from scratch.

- Well-known pretrained architectures include ResNet, VGG, and EfficientNet, all trained on the ImageNet corpus of millions of labeled images.

3 Recurrent Neural Networks

Many data sources are *sequential* in nature and call for specialized architectures. Examples include: documents (where word order conveys meaning for tasks like sentiment analysis and translation), time series (financial, weather, climate), recorded speech and audio, and handwriting. In all of these, the ordering and relative positions of the input elements carry essential information that a bag-of-features approach would discard.

A natural first attempt is a fixed-window approach. For example, in *language modeling*—the task of predicting the next word given its predecessors—a *q-gram model* approximates the conditional probability of word w_ℓ by conditioning only on the preceding $q-1$ words rather than the full history:

$$P(w_1, \ldots, w_L) = \prod_{\ell=1}^{L} P(w_\ell \mid w_1, \ldots, w_{\ell-1}) \approx \prod_{\ell=1}^{L} P(w_\ell \mid w_{\ell-q+1}, \ldots, w_{\ell-1}).$$

Enlarging the window q leads to severe *sparsity* (many q-grams never appear in the training corpus) and *storage* problems (the number of distinct q-grams grows exponentially in q). A window-based neural network (e.g., James et al. 2021, Section 10.4) replaces the count table with a learned function, but still conditions on a fixed number of preceding inputs.

A *recurrent neural network* (RNN) removes this limitation by conditioning on the *entire* history through a hidden state that is updated at each step (James et al., 2021). The input X is a sequence $X = \{X_1, X_2, \ldots, X_L\}$ of L vectors. For instance, in document classification each X_ℓ represents a word. The output Y can also be a sequence (as in language translation), but often is a scalar—e.g., the binary sentiment label of a movie review.

3.1 RNN Architecture

- The network processes the input sequence one element at a time, from X_1 to X_L. At each step ℓ, the hidden layer receives two inputs: the current input vector X_ℓ and the activation vector $A_{\ell-1}$ from the previous step. It produces a new activation vector A_ℓ, which in turn feeds into the output layer to produce a prediction O_ℓ.

- Suppose each input vector X_ℓ has p components $X_\ell' = (X_{\ell 1}, X_{\ell 2}, \ldots, X_{\ell p})$, and the hidden layer has K units $A_\ell' = (A_{\ell 1}, A_{\ell 2}, \ldots, A_{\ell K})$. Let $\mathbf{W}$ denote the $K \times (p+1)$ weight matrix

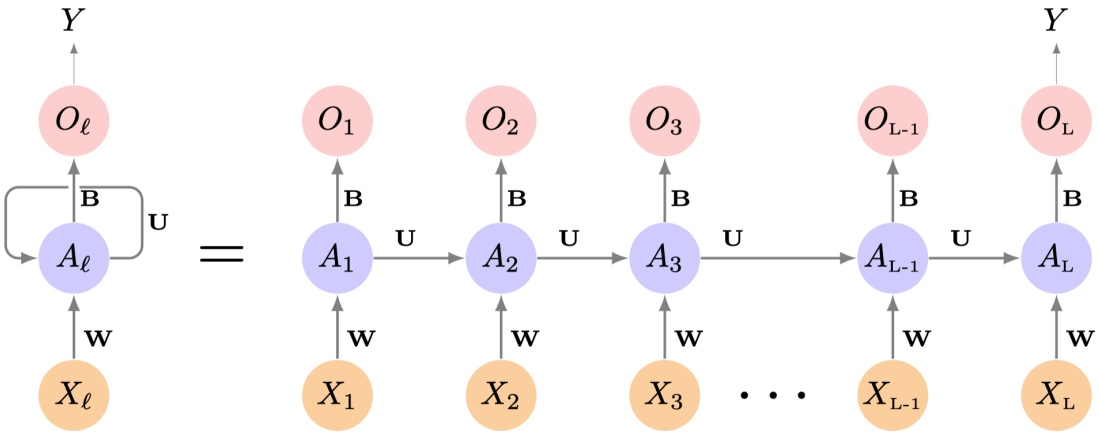


FIGURE 10.12. *Schematic of a simple recurrent neural network. The input is a sequence of vectors* $\{X_\ell\}_1^L$*, and here the target is a single response. The network processes the input sequence* X *sequentially; each* X_ℓ *feeds into the hidden layer, which also has as input the activation vector* $A_{\ell-1}$ *from the previous element in the sequence, and produces the current activation vector* A_ℓ*. The same collections of weights* $\mathbf{W}$*,* $\mathbf{U}$ *and* $\mathbf{B}$ *are used as each element of the sequence is processed. The output layer produces a sequence of predictions* O_ℓ *from the current activation* A_ℓ*, but typically only the last of these,* O_L*, is of relevance. To the left of the equal sign is a concise representation of the network, which is* unrolled *into a more explicit version on the right.*

Figure 41: Schematic of a simple recurrent neural network. The input is a sequence of vectors $\{X_\ell\}_1^L$, and here the target is a single response. The network processes the input sequence X sequentially; each X_ℓ feeds into the hidden layer, which also receives the activation vector $A_{\ell-1}$ from the previous element. The same weight collections $\mathbf{W}$, $\mathbf{U}$ and $\boldsymbol{\beta}$ are used at every step. Left: compact representation. Right: unrolled version. Source: James et al. (2021), Figure 10.12.

from the input to the hidden layer, $\mathbf{U}$ the $K \times K$ weight matrix for the hidden-to-hidden connections, and $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_K)'$ the output-layer weights. The hidden activations are

$$A_{\ell k} = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_{\ell j} + \sum_{s=1}^{K} u_{ks} A_{\ell-1,s}\right), \quad k = 1, \ldots, K, \tag{17}$$

and the output at step ℓ is

$$O_\ell = \beta_0 + \sum_{k=1}^{K} \beta_k A_{\ell k} \tag{18}$$

for a quantitative response (with an additional sigmoid or softmax activation for classification). Here $g(\cdot)$ is an activation function such as ReLU.

- A crucial feature is *weight sharing*: the same parameters $\mathbf{W}$, $\mathbf{U}$, and $\boldsymbol{\beta}$ are used at every step $\ell = 1, \ldots, L$. This is analogous to the reuse of convolution filters across spatial positions in a CNN. As the network proceeds from beginning to end, the activations A_ℓ accumulate a summary of the history seen so far, so that learned context can inform prediction.

- For regression, the loss for a single observation (X, Y) is

$$(Y - O_L)^2,$$

  which depends only on the *final* output O_L. The intermediate outputs $O_1, \ldots, O_{L-1}$ are not used directly, but every element X_ℓ of the input sequence contributes to O_L through the recurrent chain (17), and hence influences the learned parameters via backpropagation. For some tasks (e.g., language translation), the full output sequence $\{O_1, \ldots, O_L\}$ is used.

3.2 Word Embeddings

Applying an RNN to text requires representing words as numerical vectors. The simplest representation is *one-hot encoding*: given a dictionary of D words, each word is encoded as a binary vector of length D with a single 1. This creates an enormous dimensionality problem—D can easily be 10,000 or more.

- A popular solution is to represent each word in a much lower-dimensional *embedding space* of dimension m, where m is typically in the low hundreds. This requires an *embedding matrix* $\mathbf{E}$ of dimension $m \times D$: each column of $\mathbf{E}$ gives the m-dimensional coordinates for one word.

- The embedding matrix can be learned as part of the network training (as an *embedding layer*), or it can be a *pretrained* embedding that is frozen during training. Two widely used

pretrained embeddings are *word2vec* and *GloVe*, both built from very large text corpora. The key property is that words with similar meanings are mapped to nearby points in the embedding space—e.g., synonyms cluster together.

- After embedding, each document is represented as a sequence of m-vectors $X = \{X_1, X_2, \ldots, X_L\}$, where L is a fixed sequence length (shorter documents are padded with zeros, longer ones are truncated to the last L words).

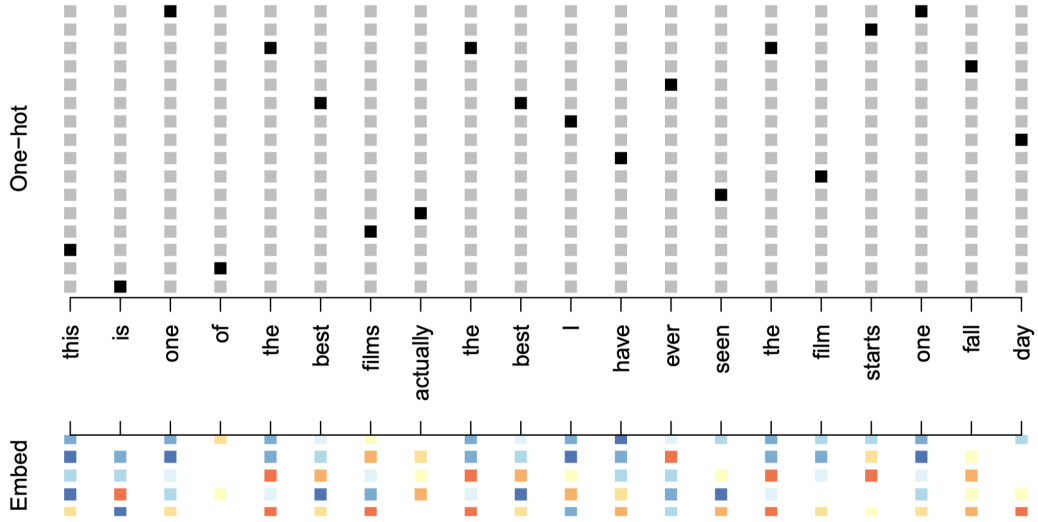


FIGURE 10.13. *Depiction of a sequence of* 20 *words representing a single document: one-hot encoded using a dictionary of* 16 *words (top panel) and embedded in an* m*-dimensional space with* $m = 5$ *(bottom panel).*

Figure 42: Depiction of a sequence of 20 words representing a single document: one-hot encoded using a dictionary of 16 words (top panel) and embedded in an m-dimensional space with $m = 5$ (bottom panel). Source: James et al. (2021), Figure 10.13.

3.3 Sequential Document Classification

As a concrete application, consider classifying IMDb movie reviews as positive or negative. The bag-of-words approach (Section 10.4 of James et al. 2021) scores each document for the presence or absence of 10,000 dictionary words, discarding word order entirely. An RNN instead processes the words sequentially, exploiting the fact that phrases like "blissfully long" and "blissfully short" have very different sentiments.

- Each review is embedded into a sequence of m-vectors via the embedding matrix $\mathbf{E}$ and then processed by the RNN of (17)–(18). The final output O_L (passed through a sigmoid) predicts the probability of a positive review.

- The model has relatively few recurrent parameters. With K hidden units, the input-to-hidden weight matrix $\mathbf{W}$ has $K(m+1)$ parameters, the hidden-to-hidden matrix $\mathbf{U}$ has K^2 parameters, and the output layer $\boldsymbol{\beta}$ has $2(K+1)$ parameters for two-class logistic regression. These are reused at every step in the sequence. If the embedding layer $\mathbf{E}$ is learned, it contributes an additional $m \times D$ parameters—by far the largest cost.

- A simple RNN with learned embeddings ($m = 32$) and $K = 32$ hidden units achieves about 76% test accuracy on the IMDb data—rather disappointing compared to the 88% of the bag-of-words model. The main difficulty is that with long sequences, early signals are progressively "washed out" as they propagate through the recurrent chain. This is the *vanishing gradient* problem: backpropagation through many time steps causes gradients to shrink exponentially, making it hard for the network to learn long-range dependencies.

- More elaborate architectures that maintain both long-term and short-term memory—the *LSTM* (Long Short-Term Memory) discussed in the next section—overcome this problem. An LSTM RNN achieves 87% accuracy on the IMDb data, comparable to the bag-of-words baseline.

3.4 RNN Training and Backpropagation Through Time

When the RNN is used for language modeling or sequence prediction, the output at each step ℓ is passed through a softmax to produce a distribution over a vocabulary of D words, and the loss at step ℓ is the cross-entropy

$$J_\ell(\boldsymbol{\theta}) = -\sum_{j=1}^{D} y_{\ell,j} \log \hat{y}_{\ell,j}, \tag{19}$$

where y_ℓ is the one-hot encoding of the true word and $\hat{y}_\ell$ the predicted distribution. The total loss over a sequence of length L is $J(\boldsymbol{\theta}) = \frac{1}{L}\sum_{\ell=1}^{L} J_\ell(\boldsymbol{\theta})$. A related evaluation metric is *perplexity*, defined as 2^J; lower perplexity indicates that the model assigns higher probability to the observed sequence.

- Because the parameters $\mathbf{W}$, $\mathbf{U}$, and $\boldsymbol{\beta}$ are *shared* across all time steps, the gradient of J_ℓ with respect to, say, $\mathbf{U}$ must account for the dependence of A_ℓ on $A_{\ell-1}$, which in turn depends on $A_{\ell-2}$, and so on. Computing these gradients is called *backpropagation through time* (BPTT): the recurrent network is "unrolled" into an L-layer feedforward network, and standard backpropagation is applied to the unrolled graph.

- Concretely, applying the chain rule yields

$$\frac{\partial J_\ell}{\partial \mathbf{U}} = \sum_{k=1}^{\ell} \frac{\partial J_\ell}{\partial O_\ell} \frac{\partial O_\ell}{\partial A_\ell} \frac{\partial A_\ell}{\partial A_k} \frac{\partial^+ A_k}{\partial \mathbf{U}}, \tag{20}$$

  where $\partial^+ A_k / \partial \mathbf{U}$ denotes the *immediate* (non-recursive) derivative of A_k with respect to $\mathbf{U}$ at step k. The inner sum over k arises because $\mathbf{U}$ participates in computing every hidden state from step 1 to step ℓ. The total gradient is $\partial J / \partial \mathbf{U} = \sum_{\ell=1}^{L} \partial J_\ell / \partial \mathbf{U}$.

- The key factor governing long-range credit assignment is the Jacobian product $\partial A_\ell / \partial A_k$, which we analyze next.

3.5 Vanishing and Exploding Gradients

The informal discussion of vanishing gradients above can be made precise. The Jacobian product in (20) expands as (Bengio et al., 1994; Pascanu et al., 2013)

$$\frac{\partial A_\ell}{\partial A_k} = \prod_{j=k+1}^{\ell} \frac{\partial A_j}{\partial A_{j-1}} = \prod_{j=k+1}^{\ell} \mathbf{U}^\top \operatorname{diag}\big[g'(A_{j-1})\big], \tag{21}$$

where g' denotes the element-wise derivative of the activation function evaluated at the pre-activation of each hidden unit.

- Taking norms and letting $\beta_U = \|\mathbf{U}\|$ and $\beta_g = \sup_z |g'(z)|$:

$$\left\| \frac{\partial A_\ell}{\partial A_k} \right\| \le \prod_{j=k+1}^{\ell} \|\mathbf{U}^\top\| \cdot \|\operatorname{diag}[g'(A_{j-1})]\| \le (\beta_U \, \beta_g)^{\ell - k}. \tag{22}$$

  For the sigmoid activation, $\beta_g \le 1/4$; for tanh, $\beta_g \le 1$.

- When $\beta_U \beta_g < 1$, the bound in (22) shrinks *exponentially* as the gap $\ell - k$ grows: contributions from distant time steps vanish, and the network cannot learn long-range dependencies. This is the **vanishing gradient** problem.

- When $\beta_U \beta_g > 1$, the bound grows exponentially, causing gradients to become extremely large. This is the **exploding gradient** problem, which typically manifests as numerical overflow (`NaN`) during training.

- To build intuition, consider trying to predict a blank in "Jane walked into the room. John walked in too. _____ said hi to _____." The correct answer ("Jane... John") depends on

context many steps back. With vanishing gradients, the error signal from the prediction at the blank cannot propagate far enough to reinforce the association with the distant names.

Solutions.

- **Gradient clipping** addresses exploding gradients (Pascanu et al., 2013). Whenever the gradient norm $\|\hat{g}\|$ exceeds a threshold τ, the gradient is rescaled:

$$\hat{g} \leftarrow \frac{\tau}{\|\hat{g}\|}\,\hat{g} \qquad \text{if } \|\hat{g}\| \geq \tau.$$

  This keeps the step size bounded without changing the gradient direction.

- **Activation functions.** Replacing the sigmoid with ReLU helps, because $g'(z) = 1$ for $z > 0$, so gradients flow without attenuation through active units.

- **Weight initialization.** Initializing $\mathbf{U}$ as the identity matrix (rather than randomly) ensures that gradients neither grow nor shrink at initialization (James et al., 2021).

- **Gated architectures.** The most effective solution is to redesign the recurrence itself so that the network can learn *when* to remember and *when* to forget. This is the idea behind GRUs and LSTMs, discussed in Section 4.

3.6 RNN Variants

The basic RNN of (17)–(18) admits several important extensions.

Bidirectional RNNs. A standard RNN processes the sequence left to right, so the hidden state A_ℓ summarizes only past inputs $X_1, \ldots, X_\ell$. In many tasks (e.g., named-entity recognition, machine translation), future context is equally informative. A *bidirectional RNN* maintains two hidden layers: a forward layer $\overrightarrow{A}_\ell$ that reads the sequence from X_1 to X_L, and a backward layer $\overleftarrow{A}_\ell$ that reads from X_L to X_1:

$$\overrightarrow{A}_\ell = g\big(\overrightarrow{\mathbf{W}} X_\ell + \overrightarrow{\mathbf{U}}\,\overrightarrow{A}_{\ell-1} + \overrightarrow{\boldsymbol{b}}\big), \tag{23}$$

$$\overleftarrow{A}_\ell = g\big(\overleftarrow{\mathbf{W}} X_\ell + \overleftarrow{\mathbf{U}}\,\overleftarrow{A}_{\ell+1} + \overleftarrow{\boldsymbol{b}}\big). \tag{24}$$

The output at step ℓ is then computed from the concatenation $[\overrightarrow{A}_\ell;\, \overleftarrow{A}_\ell]$, giving the network access to both past and future context.

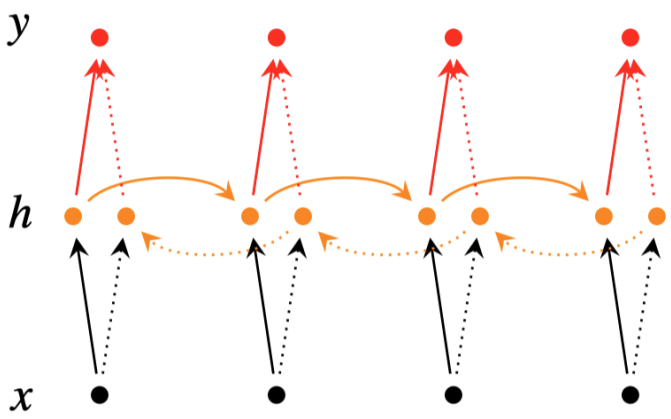


Figure 8: A bi-directional RNN model

Figure 43: A bidirectional RNN model. At each time step, the hidden layer consists of a forward component (solid orange arrows, propagating left to right) and a backward component (dotted orange arrows, propagating right to left). The output $\hat{y}$ at each step is computed from the concatenation of both hidden states. Source: CS224n lecture notes, Figure 8.

Deep (multi-layer) RNNs. RNNs can be stacked: the hidden-state sequence of layer $i-1$ serves as the input sequence for layer i. For a network with M layers, the hidden state of layer i at step ℓ is

$$A_\ell^{(i)} = g\big(\mathbf{W}^{(i)} A_\ell^{(i-1)} + \mathbf{U}^{(i)} A_{\ell-1}^{(i)} + \boldsymbol{b}^{(i)}\big), \qquad i = 1, \dots, M,$$

where $A_\ell^{(0)} = X_\ell$. Deeper architectures can learn more abstract sequential features, at the cost of increased computation and data requirements.

Sequence-to-sequence (Seq2Seq) models. In tasks like machine translation, both the input and the output are variable-length sequences. The *encoder–decoder* framework handles this with two RNNs: an *encoder* that reads the input sequence and compresses it into a fixed-length context vector $\boldsymbol{c} = A_L^{\text{enc}}$, and a *decoder* that generates the output sequence one token at a time, conditioned on $\boldsymbol{c}$ and the previously generated tokens. The decoder hidden state is

$$A_\ell^{\text{dec}} = g\big(\mathbf{W}^{\text{dec}} \hat{Y}_{\ell-1} + \mathbf{U}^{\text{dec}} A_{\ell-1}^{\text{dec}} + \boldsymbol{b}^{\text{dec}}\big),$$

where $\hat{Y}_{\ell-1}$ is the embedding of the previous output token. A common extension feeds $\boldsymbol{c}$ as an additional input to every decoder step, rather than only as the initial hidden state.

3.7 Summary

- RNNs provide a flexible framework for modeling sequential data. Their weight-sharing structure is analogous to convolutional filters in CNNs: where CNNs share filters across *spatial* positions, RNNs share weights across *temporal* positions.
- The central computational challenge is the vanishing gradient problem (Section 3.5), which limits the ability of simple RNNs to learn long-range dependencies.
- The most effective remedy is to replace the standard hidden unit with a *gated* unit that learns when to retain and when to discard information. The two most widely used gated architectures—the GRU and the LSTM—are discussed in the next section.

4 Gated Recurrent Architectures

The vanishing gradient problem (Section 3.5) makes it difficult for a standard RNN to retain information over many time steps. Gated architectures address this by introducing learnable *gates*—sigmoid-activated vectors that control the flow of information through the recurrence. Because each gate output lies in $[0, 1]$, the network can learn to selectively *remember* relevant signals and *forget* irrelevant ones, creating shortcut paths through which gradients can propagate without exponential decay.

We discuss two prominent gated architectures: the *Gated Recurrent Unit* (GRU) of Cho et al. (2014) and the *Long Short-Term Memory* (LSTM) of Hochreiter and Schmidhuber (1997). In what follows we index time steps by t and write x_t for the input vector, h_t for the hidden state, and $\sigma(\cdot)$ for the element-wise sigmoid function.

4.1 Gated Recurrent Units (GRU)

The GRU modifies the standard recurrence by introducing two gates that regulate how much of the past hidden state is retained and how much new information is incorporated. At each step t, the computation proceeds in four stages:

1. **Update gate.** The update gate z_t determines how much of the previous hidden state h_{t-1} is carried forward:

$$z_t = \sigma\big(\mathbf{W}^{(z)} x_t + \mathbf{U}^{(z)} h_{t-1} + \boldsymbol{b}^{(z)}\big). \tag{25}$$

   When $z_t \approx 1$, the hidden state is copied almost unchanged; when $z_t \approx 0$, it is replaced by new content.

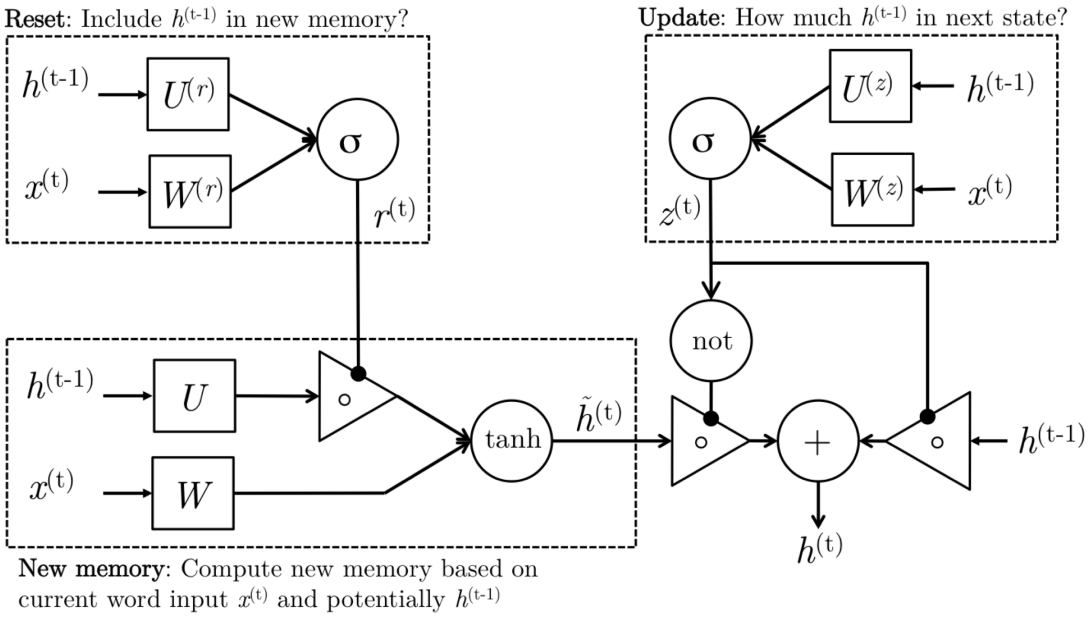


Figure 44: Internal architecture of a Gated Recurrent Unit (GRU). Left: the reset gate r_t controls how much of h_{t-1} enters the new memory computation. Bottom: the new memory $\tilde{h}_t$ is formed from x_t and the reset-gated h_{t-1}. Right: the update gate z_t interpolates between h_{t-1} and $\tilde{h}_t$ to produce h_t. Source: CS224n lecture notes, Figure 12.

2. **Reset gate.** The reset gate r_t controls how much of h_{t-1} is used when computing the candidate hidden state:

$$r_t = \sigma\big(\mathbf{W}^{(r)} x_t + \mathbf{U}^{(r)} h_{t-1} + \boldsymbol{b}^{(r)}\big). \tag{26}$$

   Setting $r_t \approx 0$ effectively ignores the previous hidden state, allowing the unit to act as if it is reading the first symbol of a new subsequence.

3. **Candidate hidden state.** A new candidate $\tilde{h}_t$ is formed by combining the current input with a selectively reset version of h_{t-1}:

$$\tilde{h}_t = \tanh\big(\mathbf{W} x_t + \mathbf{U}(r_t \odot h_{t-1}) + \boldsymbol{b}\big), \tag{27}$$

   where $\odot$ denotes the Hadamard (element-wise) product.

4. **Hidden state.** The final hidden state is a convex combination of h_{t-1} and $\tilde{h}_t$, controlled by the update gate:

$$h_t = (1 - z_t) \odot \tilde{h}_t + z_t \odot h_{t-1}. \tag{28}$$

- The GRU has three sets of weight matrices $(\mathbf{W}^{(z)}, \mathbf{U}^{(z)})$, $(\mathbf{W}^{(r)}, \mathbf{U}^{(r)})$, and $(\mathbf{W}, \mathbf{U})$, each with associated biases. All parameters are learned via BPTT, exactly as for a standard RNN.

- The update gate is the key mechanism for mitigating vanishing gradients: when $z_t \approx 1$, the recurrence $h_t \approx h_{t-1}$ creates a nearly identity mapping, allowing gradients to flow unattenuated across many steps. The network learns *which* dimensions of the hidden state to preserve and which to refresh, based on the input.

4.2 Long Short-Term Memory (LSTM)

The LSTM predates the GRU and introduces a separate *cell state* C_t that runs alongside the hidden state h_t. The cell state acts as a conveyor belt: information can be added to or removed from it through three gates, but otherwise it is transmitted with minimal transformation. Figure 45 illustrates the architecture.

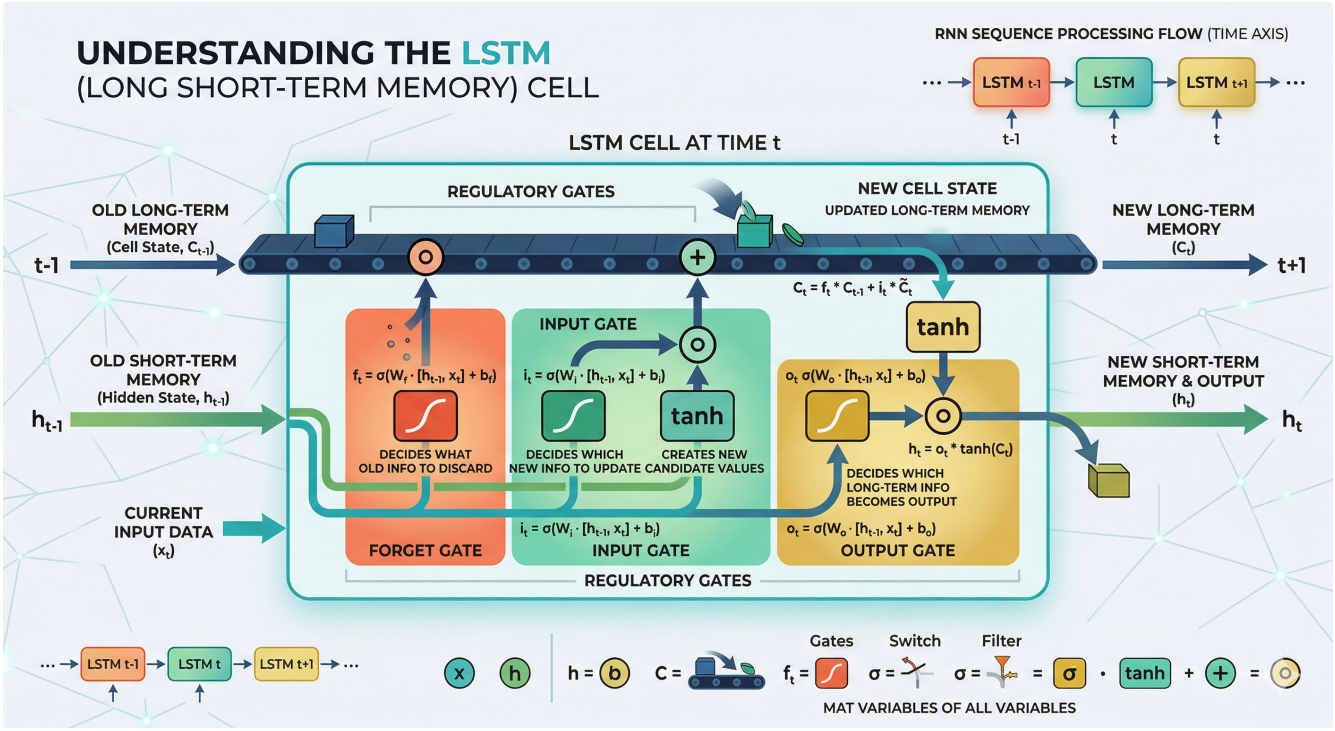


Figure 45: Architecture of a single LSTM cell at time t. The cell state C_t flows along the top, modified only by element-wise operations controlled by the forget, input, and output gates. The hidden state h_t is a gated, tanh-squashed copy of the cell state. Source: Gemini.

At each time step t, the LSTM receives the input x_t and the previous hidden state h_{t-1}, and performs the following computations:

1. **Forget gate.** The forget gate f_t decides which components of the *old* cell state C_{t-1} to retain:

$$f_t = \sigma\big(\mathbf{W}^{(f)} x_t + \mathbf{U}^{(f)} h_{t-1} + \boldsymbol{b}^{(f)}\big). \tag{29}$$

When $f_t \approx 0$ for a particular component, that component of the old memory is erased; when $f_t \approx 1$, it is preserved.

2. **Input gate.** The input gate i_t decides which components of the *new* candidate memory to write:
$$i_t = \sigma\big(\mathbf{W}^{(i)} x_t + \mathbf{U}^{(i)} h_{t-1} + \boldsymbol{b}^{(i)}\big). \tag{30}$$

3. **Candidate cell state.** A new candidate memory $\tilde{C}_t$ is generated from the current input and previous hidden state:
$$\tilde{C}_t = \tanh\big(\mathbf{W}^{(c)} x_t + \mathbf{U}^{(c)} h_{t-1} + \boldsymbol{b}^{(c)}\big). \tag{31}$$

4. **Cell state update.** The new cell state C_t combines selectively forgotten old memories with selectively written new memories:
$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t. \tag{32}$$
This additive update is the central design feature of the LSTM: because C_t depends *linearly* on C_{t-1} (modulated by f_t), gradients can flow through the cell state without the multiplicative shrinkage that plagues the standard RNN.

5. **Output gate.** The output gate o_t controls which components of the cell state are exposed as the hidden state:
$$o_t = \sigma\big(\mathbf{W}^{(o)} x_t + \mathbf{U}^{(o)} h_{t-1} + \boldsymbol{b}^{(o)}\big). \tag{33}$$

6. **Hidden state.** The hidden state h_t is a gated, tanh-squashed version of the cell state:
$$h_t = o_t \odot \tanh(C_t). \tag{34}$$
The cell state C_t may contain information that is useful for future predictions but not needed at the current step; the output gate allows the network to keep such information hidden from the rest of the network.

- The LSTM has four sets of parameters: $(\mathbf{W}^{(f)}, \mathbf{U}^{(f)}, \boldsymbol{b}^{(f)})$ for the forget gate, $(\mathbf{W}^{(i)}, \mathbf{U}^{(i)}, \boldsymbol{b}^{(i)})$ for the input gate, $(\mathbf{W}^{(c)}, \mathbf{U}^{(c)}, \boldsymbol{b}^{(c)})$ for the candidate cell, and $(\mathbf{W}^{(o)}, \mathbf{U}^{(o)}, \boldsymbol{b}^{(o)})$ for the output gate. If the hidden state has dimension K and the input has dimension p, the total parameter count is $4\big[K(K+p)+K\big] = 4K(K+p+1)$—roughly four times that of a simple RNN with the same hidden size.

- The distinction between cell state C_t and hidden state h_t is a key difference from the GRU, which maintains only a single state vector. The cell state provides a dedicated, slowly changing memory channel, while the hidden state can vary more freely across steps.

4.3 GRU vs. LSTM

- Both architectures mitigate the vanishing gradient problem through gating mechanisms that create adaptive shortcut connections across time. The LSTM uses three gates and a separate cell state; the GRU uses two gates and folds everything into a single hidden state.

- In practice, neither architecture uniformly dominates the other. LSTMs tend to perform well on tasks requiring very long memory (e.g., language modeling over long documents), while GRUs, having fewer parameters, can be faster to train and may generalize better on smaller datasets.

- Both GRU and LSTM cells can be used as drop-in replacements for the standard RNN hidden layer in any of the architectures discussed earlier—bidirectional, deep, and encoder–decoder models. For instance, the encoder–decoder model for machine translation typically uses LSTM or GRU cells in both the encoder and the decoder.

5 Attention is All You Need

The recurrent architectures of the preceding sections suffer from two fundamental limitations. First, the sequential nature of the recurrence—h_t depends on h_{t-1}—prevents parallelization across time steps, making training on long sequences slow even on modern GPUs. Second, the *linear interaction distance* means that the number of computation steps separating two tokens grows linearly with their distance in the sequence; despite the gating mechanisms of LSTMs and GRUs, learning long-range dependencies remains difficult.

The *Transformer* architecture (Vaswani et al., 2017) addresses both limitations by dispensing with recurrence entirely. Its core mechanism is *self-attention*, which allows every token in a sequence to attend directly to every other token in a single computation step. This reduces the maximum interaction distance to $O(1)$ and enables full parallelization across sequence positions. Since its introduction, the Transformer has become the dominant architecture not only in natural language processing but increasingly across many domains of machine learning.

5.1 Self-Attention

The key idea of self-attention is to build a *contextual representation* of each token by allowing it to "look at" and aggregate information from all other tokens in the sequence. In contrast to the non-contextual embedding $x_i = \mathbf{E}w_i$, which depends only on the identity of the word, a self-attention layer produces a representation h_i that depends on the entire sequence $x_1, \ldots, x_n$.

The mechanism is based on a *key-query-value* analogy. Given input representations $x_1, \ldots, x_n \in \mathbb{R}^d$, we define three linear projections parameterized by matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{d \times d}$:

$$q_i = \mathbf{Q}^\top x_i, \qquad k_j = \mathbf{K}^\top x_j, \qquad v_j = \mathbf{V}^\top x_j, \tag{35}$$

where $q_i \in \mathbb{R}^d$ is the *query* for token i ("what am I looking for?"), $k_j \in \mathbb{R}^d$ is the *key* for token j ("what do I contain?"), and $v_j \in \mathbb{R}^d$ is the *value* for token j ("what information do I provide?"). Using distinct matrices $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ allows the network to use different "views" of each token for the three roles.

- The contextual representation of token i is a weighted sum of all values:

$$h_i = \sum_{j=1}^{n} \alpha_{ij}\, v_j, \tag{36}$$

  where the *attention weights* α_{ij} determine how much token i attends to token j. These weights are computed via *scaled dot-product attention*:

$$\alpha_{ij} = \frac{\exp\!\left(q_i^\top k_j \,/\, \sqrt{d}\right)}{\sum_{j'=1}^{n} \exp\!\left(q_i^\top k_{j'} \,/\, \sqrt{d}\right)}. \tag{37}$$

- The scaling factor $1/\sqrt{d}$ is important: when the dimension d is large, the dot product $q_i^\top k_j$ grows in magnitude roughly as $\sqrt{d}$ (assuming entries are independent with unit variance), pushing the softmax into saturated regions with very small gradients. Dividing by $\sqrt{d}$ counteracts this effect (Vaswani et al., 2017).

- The dot product $q_i^\top k_j$ measures the relevance of token j to token i; the softmax converts these relevance scores into a probability distribution over the sequence. The resulting attention weight α_{ij} "softly selects" the information from v_j to include in the representation h_i.

Matrix form. In practice, self-attention is computed for all positions simultaneously. Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ denote the matrix whose rows are $x_1^\top, \ldots, x_n^\top$. Then the entire self-attention operation can

be written as

$$\text{Attention}(\mathbf{X}) = \text{softmax}\left(\frac{\mathbf{X}\mathbf{Q}\,\mathbf{K}^\top\mathbf{X}^\top}{\sqrt{d}}\right)\mathbf{X}\mathbf{V}, \tag{38}$$

where the softmax is applied row-wise. This formulation involves only matrix multiplications and can be computed efficiently on GPUs.

5.2 Position Representations

Self-attention is *permutation-equivariant*: if we shuffle the input sequence, the outputs are shuffled in exactly the same way. Unlike RNNs, where the sequential rollout inherently encodes position, self-attention has no built-in notion of token order. Without additional information, the model would treat "the dog bit the man" identically to "the man bit the dog."

To inject positional information, a *positional encoding* $p_i \in \mathbb{R}^d$ is added to each input embedding before self-attention:

$$\tilde{x}_i = x_i + p_i.$$

Sinusoidal positional encoding. Vaswani et al. (2017) propose fixed, deterministic encodings based on sinusoidal functions of varying frequencies. For position i and dimension index j:

$$p_{i,2j} = \sin\left(\frac{i}{10000^{\,2j/d}}\right), \qquad p_{i,2j+1} = \cos\left(\frac{i}{10000^{\,2j/d}}\right), \tag{39}$$

for $j = 0, 1, \ldots, d/2 - 1$.

- Each dimension oscillates at a different frequency, ranging from 2π to $10000 \cdot 2\pi$. A useful property is that $p_{i+\Delta}$ can be expressed as a linear function of p_i for any fixed offset Δ, enabling the model to learn to attend to relative positions.
- An alternative is to use *learned positional embeddings*: a trainable matrix $\mathbf{P} \in \mathbb{R}^{N\times d}$, where N is the maximum sequence length, with p_i as its i-th row. This approach is used in BERT (Devlin et al., 2019). In practice, both approaches perform comparably.

5.3 Multi-Head Self-Attention

A single self-attention head computes one weighted average over the values, which can limit its ability to simultaneously capture different types of relationships. For example, syntactic structure may require attention to nearby words, while semantic meaning may require attention to distant words. *Multi-head attention* addresses this by running H self-attention operations in parallel, each with its own parameters.

For each head $\ell \in \{1, \ldots, H\}$, we define separate projection matrices $\mathbf{Q}^{(\ell)}, \mathbf{K}^{(\ell)}, \mathbf{V}^{(\ell)} \in \mathbb{R}^{d \times d_k}$, where $d_k = d/H$. Each head computes its own attention output:

$$h_i^{(\ell)} = \sum_{j=1}^{n} \alpha_{ij}^{(\ell)} \, v_j^{(\ell)}, \qquad \alpha_{ij}^{(\ell)} = \frac{\exp\big(q_i^{(\ell)\top} k_j^{(\ell)} / \sqrt{d_k}\big)}{\sum_{j'=1}^{n} \exp\big(q_i^{(\ell)\top} k_{j'}^{(\ell)} / \sqrt{d_k}\big)}. \tag{40}$$

- The outputs of all heads are concatenated and projected through a linear transformation $\mathbf{O} \in \mathbb{R}^{d \times d}$:

$$h_i = \mathbf{O}^\top \begin{bmatrix} h_i^{(1)} \\ \vdots \\ h_i^{(H)} \end{bmatrix}. \tag{41}$$

  Since each head output has dimension $d_k = d/H$, the concatenation has dimension d, and the final output $h_i \in \mathbb{R}^d$ preserves the model dimension.

- Multi-head attention is *no more expensive* than single-head attention with the same model dimension. Each head operates on a (d/H)-dimensional subspace, and the H heads together cover the full d-dimensional space. The total computation remains dominated by $O(n^2 d)$.

5.4 The Transformer Block

The Transformer architecture is built by stacking identical *Transformer blocks*. Each block combines multi-head self-attention with a position-wise feed-forward network, connected via *residual connections* and *layer normalization*.

Feed-forward network. After the self-attention sub-layer, each token representation is passed independently through a two-layer feed-forward network (FFN):

$$\mathrm{FFN}(h_i) = \mathbf{W}_2 \, \mathrm{ReLU}(\mathbf{W}_1 h_i + \boldsymbol{b}_1) + \boldsymbol{b}_2, \tag{42}$$

where $\mathbf{W}_1 \in \mathbb{R}^{d_{ff} \times d}$, $\mathbf{W}_2 \in \mathbb{R}^{d \times d_{ff}}$, and typically $d_{ff} = 4d$. The larger inner dimension provides additional model capacity, and because the operation is applied independently to each position and involves only matrix multiplications, it is highly parallelizable.

Layer normalization. Layer normalization (Ba et al., 2016) stabilizes training by normalizing the activations across the feature dimension. For a vector $h_i \in \mathbb{R}^d$:

$$\mathrm{LN}(h_i) = \frac{h_i - \hat{\mu}_i}{\hat{\sigma}_i}, \tag{43}$$

where $\hat{\mu}_i = \frac{1}{d}\sum_{j=1}^{d} h_{ij}$ and $\hat{\sigma}_i = \left(\frac{1}{d}\sum_{j=1}^{d}(h_{ij} - \hat{\mu}_i)^2\right)^{1/2}$ are the mean and standard deviation computed across the d components of h_i. Importantly, the statistics are computed independently for each position i and each example in the batch.

Residual connections. Following He et al. (2016), residual (skip) connections add the input of a sub-layer to its output:

$$f_{\text{res}}(h) = f(h) + h, \tag{44}$$

where f is either the self-attention or feed-forward sub-layer. The identity shortcut provides a direct path for gradient flow (the local gradient of the identity is 1 everywhere), enabling the training of much deeper networks.

Putting it together. Each Transformer block consists of two sub-layers, each wrapped with a residual connection and layer normalization. In the *pre-normalization* variant (Xiong et al., 2020), which provides more stable gradients at initialization:

$$h^{\text{mid}} = \text{MultiHeadAttn}\big(\text{LN}(h^{\text{in}})\big) + h^{\text{in}}, \tag{45}$$

$$h^{\text{out}} = \text{FFN}\big(\text{LN}(h^{\text{mid}})\big) + h^{\text{mid}}. \tag{46}$$

In diagrams, the combination of a residual connection and layer normalization is commonly labeled "Add & Norm."

5.5 Transformer Encoder, Decoder, and Encoder–Decoder

The Transformer block is the fundamental unit from which three main architectures are constructed.

Transformer Encoder. A Transformer Encoder takes a single input sequence $w_{1:n}$, embeds it via $\tilde{x}_i = \mathbf{E}w_i + p_i$ (word embedding plus positional encoding), and passes the result through a stack of L identical Transformer blocks. Attention is *bidirectional*: each token can attend to all other tokens in the sequence, including those at later positions. This produces rich contextual representations useful for tasks such as classification and named-entity recognition. BERT (Devlin et al., 2019) is a prominent example.

Transformer Decoder. For *autoregressive* tasks such as language modeling—predicting the next token given all previous tokens—the model must not attend to future positions. A Transformer Decoder enforces this via *future masking* (also called *causal masking*): the attention weights are

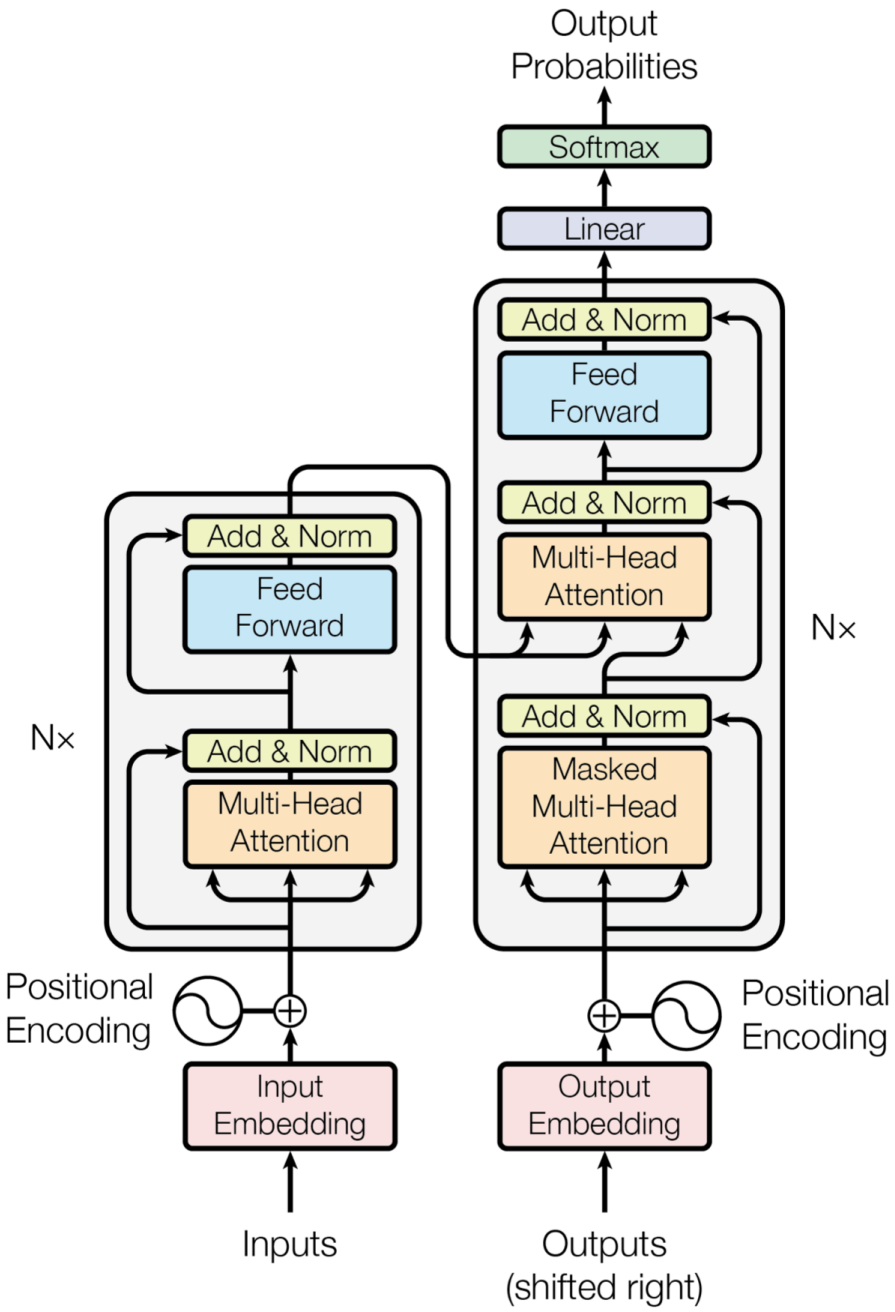


Figure 1: The Transformer - model architecture.

Figure 46: The Transformer architecture. Source: Vaswani et al. (2017), Figure 1; see also CS224n lecture notes.

modified so that position i can only attend to positions $j \le i$:

$$\alpha_{ij}^{\text{masked}} = \begin{cases} \alpha_{ij} & \text{if } j \le i, \\ 0 & \text{otherwise.} \end{cases} \tag{47}$$

In practice, this is implemented by adding a large negative constant (e.g., -10^5) to the attention logits $q_i^\top k_j / \sqrt{d_k}$ for $j > i$ before applying the softmax. The GPT family of models (Radford et al., 2019; Brown et al., 2020) are prominent examples of Transformer Decoders.

Transformer Encoder–Decoder. For sequence-to-sequence tasks such as machine translation, the Transformer uses both an encoder and a decoder. The encoder processes the source sequence $x_{1:n}$ with bidirectional (unmasked) attention. The decoder generates the target sequence $y_{1:m}$ autoregressively, using *cross-attention* to incorporate information from the encoder output.

Cross-attention is identical to self-attention except that the queries come from the decoder while the keys and values come from the encoder. Let $h_{1:n}^{(\text{enc})}$ denote the encoder output. At each decoder layer, the cross-attention sub-layer computes:

$$q_i = \mathbf{Q}^\top h_i^{(\text{dec})}, \qquad k_j = \mathbf{K}^\top h_j^{(\text{enc})}, \qquad v_j = \mathbf{V}^\top h_j^{(\text{enc})}, \tag{48}$$

for $i \in \{1, \dots, m\}$ and $j \in \{1, \dots, n\}$, and the attention output is computed as in (36)–(37). Each decoder block thus contains three sub-layers: (1) masked self-attention over the decoder sequence, (2) cross-attention from the decoder to the encoder, and (3) a position-wise feed-forward network—each with its own residual connection and layer normalization.

- While the encoder–decoder architecture provides strong performance by allowing bidirectional encoding of the source (Raffel et al., 2020), most of the largest language models today (e.g., GPT-3 and its successors) use a *decoder-only* architecture, which simplifies the design and scales well.

- The total number of parameters in a Transformer is dominated by the weight matrices in the attention and feed-forward layers. For a model with L blocks, hidden dimension d, H heads, and feed-forward dimension $d_{ff} = 4d$, each block contributes approximately $4d^2$ parameters for attention (from $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{O}$) and $8d^2$ parameters for the feed-forward network, yielding roughly $12Ld^2$ parameters in total (excluding embeddings and biases).

- Self-attention computes n^2 pairwise interactions per layer, resulting in $O(n^2 d)$ time and $O(n^2)$ memory complexity per layer. This quadratic dependence on sequence length is the primary

computational bottleneck for very long sequences and has motivated a large body of work on efficient attention mechanisms.

Part VI

Unsupervised Learning

- Everything we have covered so far concerns regression—estimating $\mathbb{E}[Y \mid X]$—or classification. These are *supervised learning* problems: we observe both features X_i and an outcome Y_i, and the goal is to learn the mapping from X to Y.
- In *unsupervised learning*, we only observe features $X_1, \ldots, X_n$ (each $X_i \in \mathbb{R}^d$) without an associated response Y. The goal is to discover interesting structure in the data: lower-dimensional representations, clusters, or latent variables.
- We cover two foundational unsupervised learning methods—*principal components analysis* (PCA) and *K-means clustering*—and then discuss their applications in panel data econometrics: *interactive fixed effects* models (which use PCA) and *grouped fixed effects* models (which use K-means).

1 Principal Components Analysis

1.1 What Are Principal Components?

- Suppose we have n observations on d features $X_1, \ldots, X_d$. When d is large, visualizing or summarizing the data is difficult. PCA finds a *low-dimensional representation* that captures as much of the variation in the data as possible.
- Assume the data have been centered so that each column of the $n \times d$ data matrix $\mathbf{X}$ has mean zero.
- The *first principal component* is the normalized linear combination of the features

$$Z_{i1} = \phi_{11} X_{i1} + \phi_{21} X_{i2} + \cdots + \phi_{d1} X_{id} = \boldsymbol{\phi}_1' X_i \tag{49}$$

that has the largest sample variance across the n observations. The vector $\boldsymbol{\phi}_1 = (\phi_{11}, \ldots, \phi_{d1})'$ is the *first loading vector*; the values $z_{11}, \ldots, z_{n1}$ are the *scores* of the first principal component.

- Formally, $\boldsymbol{\phi}_1$ solves

$$\max_{\boldsymbol{\phi}} \left\{ \frac{1}{n} \sum_{i=1}^{n} \left(\sum_{j=1}^{d} \phi_j X_{ij} \right)^2 \right\} \quad \text{subject to} \quad \|\boldsymbol{\phi}\|^2 = \sum_{j=1}^{d} \phi_j^2 = 1. \tag{50}$$

  The objective equals $\frac{1}{n}\sum_{i=1}^{n} z_{i1}^2$, the sample variance of the scores (since the data are centered). The normalization constraint prevents inflating the variance by scaling up $\boldsymbol{\phi}$.

- The solution to (50) is the eigenvector of the $d \times d$ sample covariance matrix $\mathbf{X}'\mathbf{X}/n$ corresponding to its largest eigenvalue.

- The *second principal component* $Z_2 = \boldsymbol{\phi}_2' X$ is the linear combination with maximal variance among all directions *uncorrelated* with Z_1. Constraining Z_2 to be uncorrelated with Z_1 is equivalent to requiring $\boldsymbol{\phi}_2 \perp \boldsymbol{\phi}_1$. In general, the kth principal component loading $\boldsymbol{\phi}_k$ is the kth eigenvector of $\mathbf{X}'\mathbf{X}/n$.

1.2 Low-Rank Approximation and SVD

- PCA has an equivalent interpretation as the *best low-rank approximation* to the data matrix. The first M principal components provide the closest M-dimensional linear subspace to the n data points, in the sense of minimizing the sum of squared Euclidean distances.

- Specifically, the first M score vectors and loading vectors solve the matrix factorization problem

$$\min_{\mathbf{A} \in \mathbb{R}^{n \times M},\, \mathbf{B} \in \mathbb{R}^{d \times M}} \sum_{j=1}^{d} \sum_{i=1}^{n} \left(X_{ij} - \sum_{m=1}^{M} A_{im} B_{jm} \right)^2. \tag{51}$$

  The solution satisfies $\hat{A}_{im} = z_{im}$ and $\hat{B}_{jm} = \phi_{jm}$, so we obtain the approximation

$$X_{ij} \approx \sum_{m=1}^{M} z_{im} \phi_{jm}. \tag{52}$$

  When $M = \min(n-1, d)$, the representation is exact.

- The solution can be computed via the *singular value decomposition* (SVD) of $\mathbf{X}$:

$$\underbrace{\mathbf{X}}_{n \times d} = \underbrace{\mathbf{U}}_{n \times n}\, \underbrace{\mathbf{D}}_{n \times d}\, \underbrace{\mathbf{V}'}_{d \times d},$$

  where $\mathbf{U}'\mathbf{U} = \mathbf{I}_n$, $\mathbf{V}'\mathbf{V} = \mathbf{I}_d$, and $\mathbf{D}$ has non-negative diagonal entries $d_1 \geq d_2 \geq \cdots \geq 0$. The

kth loading vector $\boldsymbol{\phi}_k$ is the kth column of $\mathbf{V}$, and the kth score vector is the kth column of $\mathbf{UD}$.

- An important consequence: the first k principal components do not depend on the choice of M (as long as $M \geq k$). We can therefore speak unambiguously of "the kth principal component."

1.3 Proportion of Variance Explained

- How much information is captured by the first M principal components? The *total variance* of the data is

$$\sum_{j=1}^{d} \mathrm{Var}(X_j) = \sum_{j=1}^{d} \frac{1}{n} \sum_{i=1}^{n} X_{ij}^2. \tag{53}$$

- The variance explained by the mth principal component is

$$\frac{1}{n} \sum_{i=1}^{n} z_{im}^2. \tag{54}$$

- The *proportion of variance explained* (PVE) by the mth component is

$$\mathrm{PVE}_m = \frac{\sum_{i=1}^{n} z_{im}^2}{\sum_{j=1}^{d} \sum_{i=1}^{n} X_{ij}^2}. \tag{55}$$

  All PVEs are non-negative and sum to one. The cumulative PVE of the first M components is obtained by summing $\mathrm{PVE}_1 + \cdots + \mathrm{PVE}_M$.

- The variance decomposition identity links the two interpretations of PCA:

$$\underbrace{\sum_{j=1}^{d} \frac{1}{n} \sum_{i=1}^{n} X_{ij}^2}_{\text{Total variance}} = \underbrace{\sum_{m=1}^{M} \frac{1}{n} \sum_{i=1}^{n} z_{im}^2}_{\text{Variance of first } M \text{ PCs}} + \underbrace{\frac{1}{n} \sum_{j=1}^{d} \sum_{i=1}^{n} \left(X_{ij} - \sum_{m=1}^{M} z_{im} \phi_{jm} \right)^2}_{\text{MSE of } M\text{-dim. approximation}}. \tag{56}$$

  Maximizing the variance of the PCs is equivalent to minimizing the approximation error. Thus, PVE can be interpreted as the R^2 of the low-rank approximation.

- A *scree plot* displays PVE (or eigenvalues) against m. One looks for an "elbow"—a point where the marginal PVE drops sharply—to choose the number of components. This is inherently subjective; in supervised settings (e.g., principal components regression), cross-validation provides a more objective criterion.

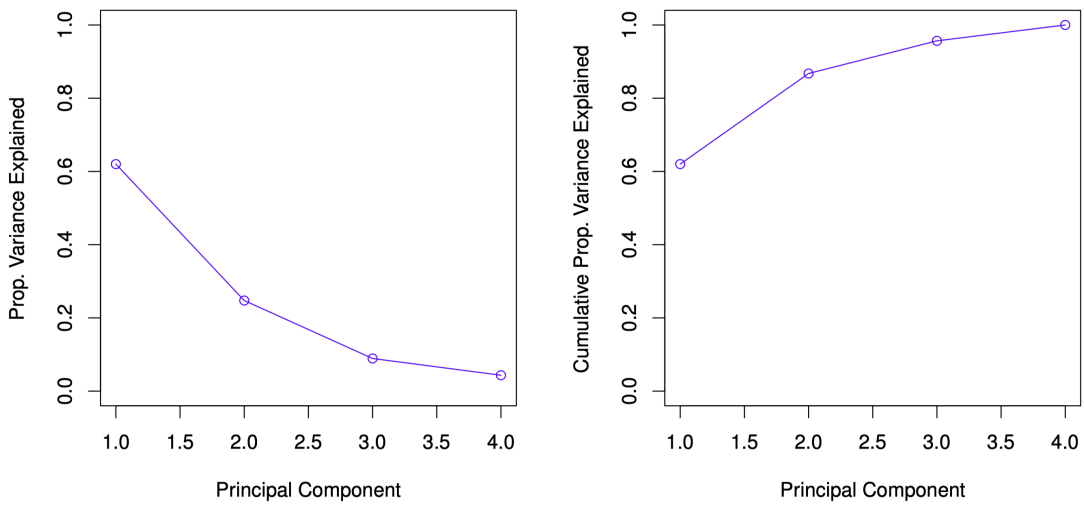


FIGURE 12.3. Left: *a scree plot depicting the proportion of variance explained by each of the four principal components in the* `USArrests` *data.* Right: *the cumulative proportion of variance explained by the four principal components in the* `USArrests` *data.*

Figure 47: Left: a scree plot depicting the proportion of variance explained by each of the four principal components in the `USArrests` data. Right: the cumulative proportion of variance explained by the four principal components in the `USArrests` data. Source: *Introduction to Statistical Learning*, James et al. (2021).

1.4 Practical Considerations

- **Scaling.** PCA results depend on whether the variables have been individually scaled. If variables are measured in different units, standardizing each to have unit variance before PCA is generally recommended. Otherwise, the variable with the largest variance will dominate the first PC.

- **Sign ambiguity.** Loading vectors are unique only up to sign flips: $\boldsymbol{\phi}_k$ and $-\boldsymbol{\phi}_k$ define the same direction. Consequently, score vectors are also unique up to sign.

- **Principal components regression.** We can use the estimated scores $\hat{Z}_{i1}, \ldots, \hat{Z}_{iM}$ as features in a supervised learning problem (OLS, lasso, etc.) to predict an outcome Y_i. This is called *principal components regression* and can be effective when the signal in Y is driven by the same latent factors that explain the variation in X.

2 PCA for Factor Models and Interactive Fixed Effects

2.1 The Factor Model

- PCA provides a natural estimation strategy for *factor models*, which are widely used in economics and finance. A linear factor model posits that a high-dimensional observed variable X_{it} is driven by a small number of latent common factors:

$$X_{it} = \lambda_i' F_t + e_{it}, \qquad i = 1, \dots, N, \quad t = 1, \dots, T, \tag{57}$$

  where F_t is an $r \times 1$ vector of *common factors*, λ_i is an $r \times 1$ vector of *factor loadings*, and e_{it} is the idiosyncratic error.

- In matrix form, let $\mathbf{X} = (X_1, \dots, X_N)$ be the $T \times N$ data matrix, $\mathbf{F} = (F_1, \dots, F_T)'$ the $T \times r$ factor matrix, and $\mathbf{\Lambda} = (\lambda_1, \dots, \lambda_N)'$ the $N \times r$ loading matrix. Then

$$\mathbf{X} = \mathbf{F}\mathbf{\Lambda}' + \mathbf{e}. \tag{58}$$

- This is an *approximate* factor model (Bai, 2003): we allow the idiosyncratic errors e_{it} to exhibit weak cross-sectional and serial correlation. The number of factors r is fixed as N and T grow.

- Factor models are used extensively in macroeconomics and finance. For example, Stock and Watson (2002) use estimated factors from a large panel of macroeconomic time series to improve forecasts. In finance, the Fama–French factors are examples of observed proxies for latent common factors driving asset returns.

2.2 PCA Estimation

- The method of principal components estimates the factors and loadings by minimizing

$$V(r) = \min_{\mathbf{\Lambda}, \mathbf{F}} \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=1}^{T} (X_{it} - \lambda_i' F_t)^2 \tag{59}$$

  subject to the normalization $\mathbf{F}'\mathbf{F}/T = \mathbf{I}_r$.

- Concentrating out $\mathbf{\Lambda}$ (given $\mathbf{F}$, the optimal loadings are $\tilde{\lambda}_i = (\mathbf{F}'\mathbf{F})^{-1}\mathbf{F}'X_i = \mathbf{F}'X_i/T$), the problem reduces to maximizing $\operatorname{tr}(\mathbf{F}'\mathbf{X}\mathbf{X}'\mathbf{F})$ over $\mathbf{F}$ subject to $\mathbf{F}'\mathbf{F}/T = \mathbf{I}_r$.

- **Solution:** $\tilde{\mathbf{F}} = \sqrt{T}$ times the r eigenvectors corresponding to the r largest eigenvalues of the $T \times T$ matrix $\mathbf{X}\mathbf{X}'/(NT)$. The estimated loadings are then $\tilde{\lambda}_i = \tilde{\mathbf{F}}'\mathbf{X}_i/T$, and the estimated common component is $\tilde{C}_{it} = \tilde{\lambda}_i'\tilde{F}_t$.

Remark 8 (Rotation indeterminacy)**.** Factors and loadings are not separately identifiable: for any invertible $r \times r$ matrix $\mathbf{A}$, $\mathbf{F}^0\mathbf{A}$ and $\mathbf{\Lambda}^0(\mathbf{A}')^{-1}$ produce the same common component $\mathbf{F}^0\mathbf{\Lambda}^{0\prime}$. The PCA estimator $\tilde{\mathbf{F}}$ estimates a rotation $\mathbf{F}^0 H$ of the true factors, where H is an invertible $r \times r$ rotation matrix. However, the common component $\tilde{C}_{it} = \tilde{\lambda}_i'\tilde{F}_t$ is identified.

2.3 Asymptotic Theory

- Bai (2003) establishes the following convergence rates and limiting distributions for the PCA estimators as $N, T \to \infty$:

  (i) **Estimated factors:** The convergence rate is $\min\{\sqrt{N}, T\}$. If $\sqrt{N}/T \to 0$, then

  $$\sqrt{N}\left(\tilde{F}_t - H'F_t^0\right) \xrightarrow{d} \mathcal{N}(0, \Pi_t),$$

  where Π_t depends on the cross-sectional covariance structure of e_{it} and the factor loadings.

  (ii) **Estimated loadings:** The convergence rate is $\min\{\sqrt{T}, N\}$. If $\sqrt{T}/N \to 0$, then

  $$\sqrt{T}\left(\tilde{\lambda}_i - H^{-1}\lambda_i^0\right) \xrightarrow{d} \mathcal{N}(0, \Theta_i),$$

  where Θ_i depends on the serial covariance structure of e_{it}.

  (iii) **Estimated common component:** The convergence rate is $\delta_{NT} = \min\{\sqrt{N}, \sqrt{T}\}$. For each (i, t),

  $$\left(\frac{1}{N}V_{it} + \frac{1}{T}W_{it}\right)^{-1/2}\left(\tilde{C}_{it} - C_{it}^0\right) \xrightarrow{d} \mathcal{N}(0, 1),$$

  where V_{it} captures the uncertainty from estimating F_t and W_{it} from estimating λ_i.

- The asymmetry in rates is intuitive: factors converge faster when N is large (many cross-sectional units provide information about each time period's factor), and loadings converge faster when T is large (many time periods provide information about each unit's loading).

2.4 Determining the Number of Factors

- In the factor model setting, the number of factors r can be estimated consistently. Bai and Ng (2002) propose information criteria of the form

$$IC(k) = \log V(k) + k \cdot g(N,T), \tag{60}$$

where $V(k)$ is the minimized objective with k factors and $g(N,T)$ is a penalty that depends on N and T. A popular choice is

$$g(N,T) = \frac{N+T}{NT} \log\left(\frac{NT}{N+T}\right).$$

- The estimated number of factors $\hat{r} = \arg\min_{0 \le k \le \bar{k}} IC(k)$ is consistent: $P(\hat{r} = r) \to 1$ as $N, T \to \infty$.

2.5 Panel Data Models with Interactive Fixed Effects

- A natural application of the factor model framework is to panel data regression. Consider the model

$$Y_{it} = X_{it}'\beta + \lambda_i' F_t + \varepsilon_{it}, \qquad i = 1, \ldots, N, \quad t = 1, \ldots, T, \tag{61}$$

where X_{it} is a $p \times 1$ vector of observable regressors, β is a $p \times 1$ vector of coefficients (the primary object of interest), $\lambda_i' F_t$ represents *interactive fixed effects*, and ε_{it} is an idiosyncratic error.

- The interactive effects $\lambda_i' F_t$ generalize the standard additive two-way fixed effects model. The additive model

$$Y_{it} = X_{it}'\beta + \alpha_i + \xi_t + \varepsilon_{it}$$

is a special case with $r = 2$: set $F_t = (1, \xi_t)'$ and $\lambda_i = (\alpha_i, 1)'$, so that $\lambda_i' F_t = \alpha_i + \xi_t$.

- **Why the standard within-group transformation fails.** With interactive effects and $r = 1$, the within-group (time-demeaned) model gives

$$Y_{it} - \bar{Y}_{i\cdot} = (X_{it} - \bar{X}_{i\cdot})'\beta + \lambda_i (F_t - \bar{F}) + (\varepsilon_{it} - \bar{\varepsilon}_{i\cdot}).$$

Since $F_t - \bar{F} \neq 0$ in general, the interactive effects are *not* eliminated by the within transformation. The standard within-group estimator is therefore inconsistent when the true effects are interactive rather than additive.

- A key advantage of model (61) is that the regressors X_{it} are allowed to be arbitrarily correlated with both λ_i and F_t. This is the econometric analogue of the "correlated random effects" assumption, but with a much richer structure.

2.6 Estimation: Concentrated Least Squares

- Bai (2009) proposes estimating (61) by least squares, jointly over β, $\mathbf{F}$, and $\mathbf{\Lambda}$:

$$\min_{\beta, \mathbf{F}, \mathbf{\Lambda}} \sum_{i=1}^{N} \|Y_i - X_i\beta - \mathbf{F}\lambda_i\|^2 \quad \text{subject to} \quad \mathbf{F}'\mathbf{F}/T = \mathbf{I}_r, \quad \mathbf{\Lambda}'\mathbf{\Lambda} \text{ diagonal}, \tag{62}$$

  where $Y_i = (Y_{i1}, \ldots, Y_{iT})'$ and X_i is $T \times p$.

- The estimation proceeds by concentration:

  1. **Concentrate out $\mathbf{\Lambda}$:** For given β and $\mathbf{F}$, the optimal loading is $\hat{\lambda}_i = \mathbf{F}'(Y_i - X_i\beta)/T$.
  2. **Concentrate out β:** For given $\mathbf{F}$, define the projection $M_F = \mathbf{I}_T - \mathbf{F}\mathbf{F}'/T$. Then

$$\hat{\beta}(\mathbf{F}) = \left(\sum_{i=1}^{N} X_i' M_F X_i \right)^{-1} \sum_{i=1}^{N} X_i' M_F Y_i.$$

  3. **Estimate F:** Given $\hat{\beta}$, define $W_i = Y_i - X_i\hat{\beta}$ (the "defactored" residuals). Then $\hat{\mathbf{F}} = \sqrt{T} \times$ the r eigenvectors of the $T \times T$ matrix $\sum_{i=1}^{N} W_i W_i'/(NT)$ corresponding to its r largest eigenvalues.

- **Iterative algorithm:** Start with an initial $\mathbf{F}^{(0)}$ (e.g., from PCA on Y). Iterate between estimating $\hat{\beta}(\mathbf{F}^{(s)})$ and updating $\mathbf{F}^{(s+1)}$ from PCA on the residuals $Y_i - X_i\hat{\beta}^{(s)}$ until convergence.

2.7 Asymptotic Properties

- Under regularity conditions (strict exogeneity, strong factors, weak dependence in ε_{it}), Bai (2009) shows that $\hat{\beta}$ is $\sqrt{NT}$-consistent:

$$\sqrt{NT}(\hat{\beta} - \beta^0) = \mathcal{O}_p(1).$$

- **Asymptotic distribution (general case).** When $T/N \to \rho > 0$, the estimator has an

asymptotic bias:

$$\sqrt{NT}(\hat{\beta} - \beta^0) \xrightarrow{d} \mathcal{N}\left(\rho^{1/2}B_0 + \rho^{-1/2}C_0,\ D_0^{-1}D_Z D_0^{-1}\right), \tag{63}$$

where B_0 captures bias from cross-sectional heteroskedasticity and C_0 captures bias from serial correlation. Both B_0 and C_0 vanish when ε_{it} is i.i.d. over i and t.

- **Bias correction.** Define the bias-corrected estimator

$$\hat{\beta}^\dagger = \hat{\beta} - \frac{1}{N}\hat{B} - \frac{1}{T}\hat{C},$$

where $\hat{B}$ and $\hat{C}$ are consistent estimators of the bias terms. Then $\sqrt{NT}(\hat{\beta}^\dagger - \beta^0) \xrightarrow{d} \mathcal{N}(0, D_0^{-1}D_3D_0^{-1})$.

2.8 Extensions

- **Unknown number of factors.** Moon and Weidner (2015) show that the LS estimator $\hat{\beta}_R$ with $R \geq R^0$ (possibly overspecified) factors has the *same* asymptotic distribution as $\hat{\beta}_{R^0}$ with the true number:

$$\sqrt{NT}(\hat{\beta}_R - \beta^0) = \sqrt{NT}(\hat{\beta}_{R^0} - \beta^0) + o_P(1).$$

This is a powerful result: valid inference on β does not require consistently estimating the number of factors.

- **Dynamic panels with predetermined regressors.** Moon and Weidner (2017) extend the framework to allow lagged dependent variables $X_{it} = Y_{i,t-1}$. The key novelty is the identification of *three* sources of asymptotic bias:

  (i) B_1: a Nickell-type bias from correlation between predetermined regressors and future errors (order $1/T$);

  (ii) B_2: bias from cross-sectional heteroskedasticity in ε_{it} (order $1/N$);

  (iii) B_3: bias from time-series heteroskedasticity in ε_{it} (order $1/T$).

  Under homoskedasticity, $B_2 = B_3 = 0$. The bias-corrected estimator is

$$\hat{\beta}^* = \hat{\beta} + \hat{W}^{-1}\left(\frac{1}{T}\hat{B}_1 + \frac{1}{N}\hat{B}_2 + \frac{1}{T}\hat{B}_3\right),$$

and satisfies $\sqrt{NT}(\hat{\beta}^* - \beta^0) \xrightarrow{d} \mathcal{N}(0, W^{-1}\Omega W^{-1})$.

- An alternative bias correction method is the *split-panel jackknife*:

$$\hat{\beta}^J = 3\hat{\beta}_{NT} - \hat{\beta}_{N,T/2} - \hat{\beta}_{N/2,T},$$

which corrects for leading bias terms of order $1/T$ and $1/N$ without requiring explicit estimation of the bias components.

3 K-Means Clustering

3.1 Introduction

- The goal of *clustering* is to partition observations into groups such that observations within a group are similar and observations in different groups are dissimilar. Applications include organizing text data by topic, grouping consumers by purchasing behavior, and classifying voters by policy preferences.
- We can motivate clustering using a *mixture model*. Suppose the distribution of X is a mixture

$$f(x) = \pi_1 f_1(x) + \cdots + \pi_K f_K(x),$$

where $f_1, \ldots, f_K$ are *mixture components* and $\pi_1, \ldots, \pi_K$ are *mixture weights* summing to one.

- K-means clustering can be viewed as a limiting case of a Gaussian mixture model where the component variances $\sigma_k^2 \to 0$ for each k (Hastie et al., 2009, Section 14.3.7).

3.2 Objective Function

- K-means partitions n observations into K non-overlapping clusters $C_1, \ldots, C_K$ by minimizing the total *within-cluster variation*:

$$\min_{C_1,\ldots,C_K} \sum_{k=1}^{K} W(C_k), \tag{64}$$

where the within-cluster variation is defined using squared Euclidean distance:

$$W(C_k) = \frac{1}{|C_k|} \sum_{i,i' \in C_k} \sum_{j=1}^{d} (X_{ij} - X_{i'j})^2 = 2 \sum_{i \in C_k} \sum_{j=1}^{d} (X_{ij} - \bar{X}_{kj})^2, \tag{65}$$

with $\bar{X}_{kj} = |C_k|^{-1} \sum_{i \in C_k} X_{ij}$ the centroid of cluster k.

- The equivalence between the pairwise distance formulation and the centroid formulation is key: minimizing within-cluster variation is the same as minimizing the sum of squared deviations from cluster centroids.

3.3 Algorithm

- Finding the global minimum of (64) is NP-hard (there are almost K^n possible partitions). The following iterative algorithm finds a local minimum:

Algorithm 1: K-Means Clustering (Lloyd's Algorithm)

Input: Data $\{X_i\}_{i=1}^n$, number of clusters K

Randomly assign each observation a cluster label $\hat{C}_i \in \{1, \ldots, K\}$;

repeat

  Centroid step: For $k = 1, \ldots, K$, compute $\hat{\mu}_k = |C_k|^{-1} \sum_{i:\hat{C}_i = k} X_i$;

  Assignment step: For $i = 1, \ldots, n$, update $\hat{C}_i = \arg\min_k \|X_i - \hat{\mu}_k\|^2$;

until *cluster assignments stop changing*;

Output: Cluster assignments $\hat{C}_1, \ldots, \hat{C}_n$ and centroids $\hat{\mu}_1, \ldots, \hat{\mu}_K$

- **Convergence guarantee.** The centroid step minimizes the objective over centroids (holding assignments fixed), and the assignment step minimizes over assignments (holding centroids fixed). The objective is therefore non-increasing at each iteration and converges in finitely many steps to a local minimum.

- **Local minima and multiple starts.** Because the algorithm converges to a *local* rather than global optimum, the result depends on the random initialization. In practice, one should run K-means many times from different random starting points and select the result with the lowest objective value.

- **Choosing K.** The choice of K is often subjective and driven by interpretability. Unlike supervised learning, where cross-validation provides an objective criterion, unsupervised learning lacks a clear "right answer." Approaches include examining the objective value as a function of K (looking for an "elbow") and domain knowledge.

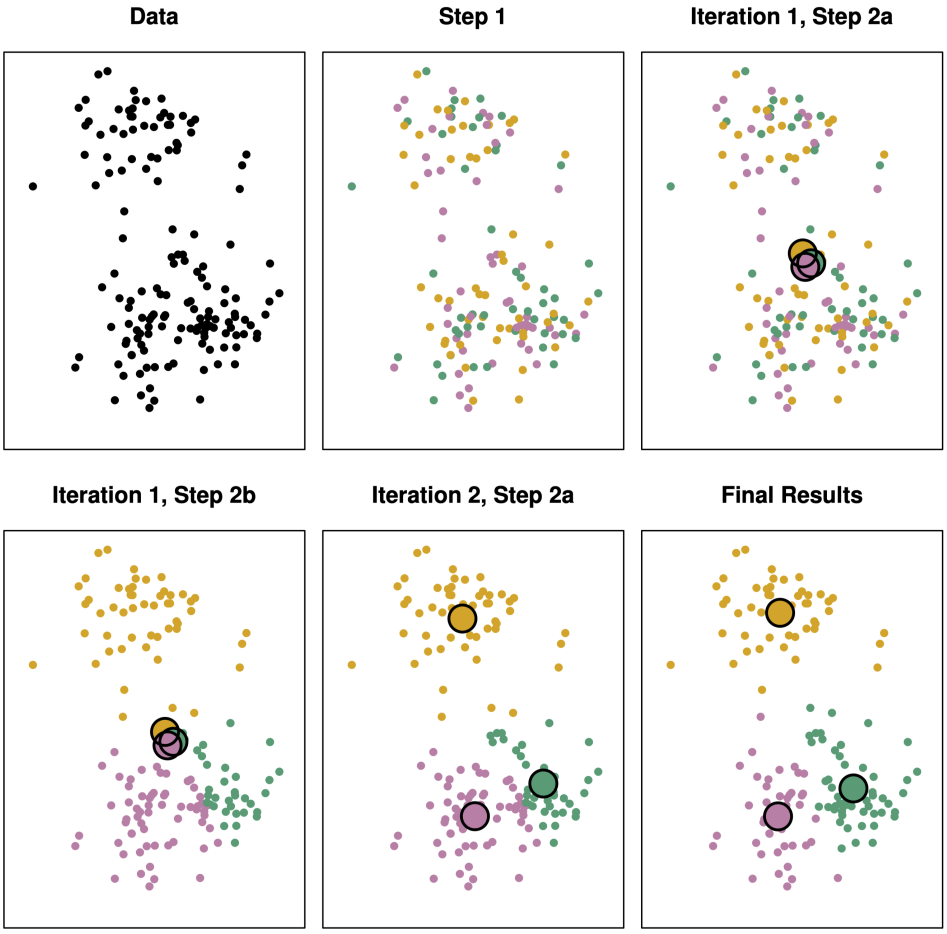


FIGURE 12.8. *The progress of the K-means algorithm on the example of Figure 12.7 with K=3.* Top left: *the observations are shown.* Top center: *in Step 1 of the algorithm, each observation is randomly assigned to a cluster.* Top right: *in Step 2(a), the cluster centroids are computed. These are shown as large colored disks. Initially the centroids are almost completely overlapping because the initial cluster assignments were chosen at random.* Bottom left: *in Step 2(b), each observation is assigned to the nearest centroid.* Bottom center: *Step 2(a) is once again performed, leading to new cluster centroids.* Bottom right: *the results obtained after ten iterations.*

Figure 48: The progress of the K-means algorithm with $K = 3$. Top left: the observations. Top center: random initial cluster assignments (Step 1). Top right: centroids computed (Step 2a); they nearly overlap because of the random initialization. Bottom left: observations reassigned to the nearest centroid (Step 2b). Bottom center: centroids recomputed. Bottom right: final result after ten iterations. Source: *Introduction to Statistical Learning*, James et al. (2021).

4 Clustering in Panel Data Models

4.1 Grouped Fixed Effects

- We now turn to an important application of K-means clustering in panel data econometrics. Bonhomme and Manresa (2015) propose the *grouped fixed effects* (GFE) model:

$$y_{it} = x_{it}'\theta + \alpha_{g_i,t} + v_{it}, \qquad i = 1, \ldots, N, \quad t = 1, \ldots, T, \tag{66}$$

  where $g_i \in \{1, \ldots, G\}$ is the *unknown* group membership of unit i, and α_{gt} are *group-specific time effects*—unrestricted parameters for each group-time pair.

- The key structural assumption is that the number of distinct time-varying paths of unobserved heterogeneity is at most G, which is small relative to N. All units in the same group share the same time profile.

- This model occupies a middle ground between:

  (i) The standard fixed effects model ($G = N$, each unit has its own path—suffers from incidental parameter bias when N is large relative to T);

  (ii) A fully homogeneous model ($G = 1$, all units share the same time effects).

- The model has a factor-analytic structure, since $\alpha_{g_i,t} = \sum_{g=1}^{G} \mathbb{1}\{g_i = g\}\alpha_{gt}$, connecting it to the interactive fixed effects framework.

4.1.1 Estimation

- The GFE estimator jointly minimizes over the common parameters θ, the group-time effects α, and the group assignments $\gamma = (g_1, \ldots, g_N)$:

$$(\hat{\theta}, \hat{\alpha}, \hat{\gamma}) = \underset{(\theta,\alpha,\gamma)\in\Theta\times\mathcal{A}^{GT}\times\Gamma_G}{\arg\min} \sum_{i=1}^{N}\sum_{t=1}^{T}(y_{it} - x_{it}'\theta - \alpha_{g_i,t})^2. \tag{67}$$

- For given θ and α, the optimal group assignment for unit i is

$$\hat{g}_i(\theta, \alpha) = \underset{g\in\{1,\ldots,G\}}{\arg\min} \sum_{t=1}^{T}(y_{it} - x_{it}'\theta - \alpha_{gt})^2, \tag{68}$$

  i.e., assign unit i to the group whose time-effect profile is closest (in Euclidean distance) to the unit's residual vector $y_{it} - x_{it}'\theta$.

- **Connection to K-means.** Without covariates ($\theta = 0$), the objective (67) reduces to the standard K-means problem, and the following algorithm reduces to Lloyd's algorithm.

Algorithm 2: GFE Estimation (Iterative K-Means Type)

Input: Data $\{(y_{it}, x_{it})\}$, number of groups G

Initialize $(\theta^{(0)}, \alpha^{(0)})$. Set $s = 0$;

repeat

  Assignment step: For all i, compute $g_i^{(s+1)} = \arg\min_g \sum_{t=1}^{T} (y_{it} - x_{it}'\theta^{(s)} - \alpha_{gt}^{(s)})^2$;

  Update step: Compute $(\theta^{(s+1)}, \alpha^{(s+1)})$ by OLS regression of y_{it} on x_{it} and group×time dummies, using assignments $g^{(s+1)}$;

  $s \leftarrow s + 1$;

until *convergence*;

Output: $\hat{\theta}$, $\hat{\alpha}$, $\hat{\gamma}$

4.1.2 Asymptotic Theory

- **Consistency (Theorem 1).** As $N, T \to \infty$: $\hat{\theta} \xrightarrow{p} \theta^0$ and $\frac{1}{NT}\sum_{i=1}^{N}\sum_{t=1}^{T}(\hat{\alpha}_{\hat{g}_i,t} - \alpha^0_{g_i^0,t})^2 \xrightarrow{p} 0$.

- **Classification consistency (Theorem 2).** Under group separation (distinct groups have different time profiles: $\text{plim}\, \frac{1}{T}\sum_t (\alpha^0_{gt} - \alpha^0_{\tilde{g},t})^2 > 0$ for $g \neq \tilde{g}$), the probability of misclassifying *any* unit vanishes:

$$\Pr\left(\sup_{1 \le i \le N} |\hat{g}_i - g_i^0| > 0\right) \to 0.$$

  The misclassification probability for any individual unit declines at an *exponential* rate in T.

- **Asymptotic equivalence.** The GFE estimator is asymptotically equivalent to the *infeasible* estimator that knows the true group memberships:

$$\hat{\theta} = \tilde{\theta} + o_p(T^{-\delta}) \quad \text{for all } \delta > 0,$$

  where $\tilde{\theta}$ is the OLS estimator using the true groups. Estimating group membership does not affect the asymptotic distribution.

- **Asymptotic normality.** Under $N/T^\nu \to 0$ for some $\nu > 0$:

$$\sqrt{NT}(\hat{\theta} - \theta^0) \xrightarrow{d} \mathcal{N}(0, \Sigma_\theta^{-1}\Omega_\theta\Sigma_\theta^{-1}). \tag{69}$$

  Standard errors from the infeasible regression with known groups are asymptotically valid.

- **Application: income and democracy.** Bonhomme and Manresa (2015) apply GFE to revisit the income–democracy nexus studied by Acemoglu et al. (2008). With $G = 4$ groups, the estimated groups capture distinct democratization patterns—stable democracies, stable autocracies, and two "waves" of democratic transitions—corresponding to the historical typology of "waves of democratization." The estimated income effect on democracy drops substantially compared to standard OLS, consistent with omitted variable bias from unobserved institutional differences.

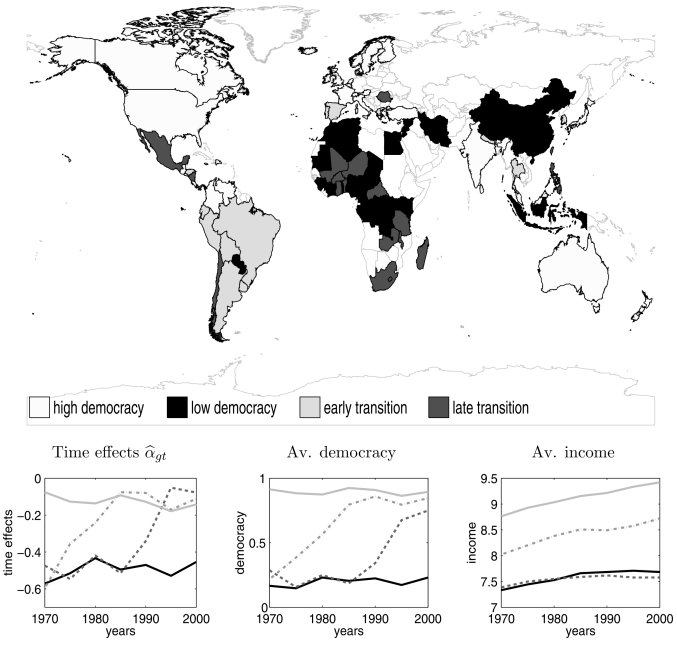


FIGURE 2.—Patterns of heterogeneity, $G = 4$. *Note*: See the notes to Figure 1. On the bottom panel, the left graph shows the group-specific time effects $\widehat{\alpha}_{gt}$. The other two graphs show the group-specific averages of democracy and lagged log-GDP per capita, respectively. Calendar years (1970–2000) are shown on the x-axis. Light solid lines correspond to Group 1 ("high-democracy"), dark solid lines to Group 2 ("low-democracy"), light dashed lines to Group 3 ("early transition"), and dark dashed lines to Group 4 ("late transition"). The top panel shows group membership. The list of countries by group is given in the Supplemental Material.

Figure 49: Patterns of heterogeneity with $G = 4$ groups. The top panel shows group membership on a world map. The bottom panel displays group-specific time effects $\hat{\alpha}_{gt}$ (left), average democracy (center), and average log-GDP per capita (right) over 1970–2000. Light solid lines: Group 1 ("high democracy"); dark solid lines: Group 2 ("low democracy"); light dashed lines: Group 3 ("early transition"); dark dashed lines: Group 4 ("late transition"). Source: Bonhomme and Manresa (2015).

4.2 GMM Framework with Multiway Clustering

- Cheng et al. (2023) generalize the grouped fixed effects approach in two directions: (i)

a nonlinear GMM framework (replacing least squares), and (ii) *multi-dimensional* group memberships.

- **Multi-dimensional group structure.** Instead of assigning each unit i to a single group, each heterogeneous parameter gets its own group membership:

$$a_i = \alpha(g_i), \quad g_i \in \{1, \ldots, n_g\}, \qquad b_i = \beta(h_i), \quad h_i \in \{1, \ldots, n_h\}.$$

  This is far more parsimonious than one-dimensional clustering: with m heterogeneous features and k groups each, the one-dimensional approach requires k^m parameters, while the multi-dimensional approach requires only km.

- **GMM estimation.** Given moment conditions $\mathbb{E}[m(w_{it}; \theta_i^0)] = 0$, the estimator solves

$$(\hat{\theta}, \hat{G}, \hat{H}) = \underset{(\theta, G, H)}{\arg\min} \frac{1}{N} \sum_{i=1}^{N} \left[\frac{1}{T} \sum_{t=1}^{T} m(w_{it}; \alpha(g_i), \beta(h_i), \lambda) \right]' W_i \left[\frac{1}{T} \sum_{t=1}^{T} m(w_{it}; \alpha(g_i), \beta(h_i), \lambda) \right]. \tag{70}$$

  This is minimized using a multi-dimensional Lloyd's algorithm that alternates between updating g-memberships, h-memberships, and parameter estimates.

- **Classification consistency.** Under identification, separation, and regularity conditions:

$$\Pr\{\hat{G} = G^0 \text{ and } \hat{H} = H^0\} \to 1 \quad \text{as } N, T \to \infty.$$

  A two-step efficient estimator using estimated memberships achieves the $\sqrt{NT}$ rate with standard GMM asymptotic normality.

4.2.1 Application: Production Function Estimation

- A leading application is to production function estimation with heterogeneous coefficients. Consider the production function

$$y_{it} = b_i v_{it} + \omega_{it} + \varepsilon_{it},$$

  where y_{it} is log output, v_{it} is log variable inputs, and ω_{it} is unobserved productivity following $\omega_{it} = a_i + \rho \omega_{i,t-1} + \xi_{it}$.

- After quasi-differencing: $\Delta y_{it}(\rho) - a_i - b_i \Delta v_{it}(\rho) = \xi_{it} + \varepsilon_{it} - \rho \varepsilon_{i,t-1}$. With instruments

$z_{it} = (1, v_{i,t-1}, v_{i,t-2})'$, we have moment conditions

$$\mathbb{E}\left[z_{it}\left(\Delta y_{it}(\rho) - a_i - b_i \Delta v_{it}(\rho)\right)\right] = 0,$$

where $a_i = \alpha(g_i)$ (productivity group), $b_i = \beta(h_i)$ (elasticity group), and $\lambda = \rho$ (common persistence).

- The multi-dimensional clustering allows firms to have different productivity levels *and* different output elasticities, with each dimension clustered independently. The empirical results show that allowing for group heterogeneity substantially affects estimated aggregate markups.

4.3 Clustering in the AKM Framework

- Bonhomme et al. (2019) apply the clustering approach to matched employer–employee data, extending the influential Abowd et al. (1999) (AKM) framework.
- **The AKM model.** The standard AKM decomposition of log-earnings is

$$Y_{it} = \alpha_i + \psi_{j(i,t)} + X_{it}'\beta + \varepsilon_{it}, \tag{71}$$

where α_i is a worker fixed effect, $\psi_{j(i,t)}$ is a firm fixed effect (for the firm j where worker i is employed at time t), and identification relies on workers who move between firms.

- **Limitations of AKM.** (i) *Additivity:* no interactions between worker and firm attributes, ruling out complementarities in production. (ii) *Incidental parameter bias:* with short panels and low mobility, the large number of firm-specific parameters creates severe upward bias in the estimated variance of firm effects.
- **Clustering firms into classes.** Instead of estimating a fixed effect for each of the J firms, Bonhomme et al. (2019) cluster firms into K *classes* using K-means on earnings distributions. Specifically, they solve the weighted K-means problem

$$\min_{k(1),\ldots,k(J),\, H_1,\ldots,H_K} \sum_{j=1}^{J} n_j \int \left(\hat{F}_j(y) - H_{k(j)}(y)\right)^2 d\mu(y), \tag{72}$$

where $\hat{F}_j$ is the empirical CDF of earnings in firm j, n_j is the number of workers, and the "centroids" $H_1, \ldots, H_K$ are class-specific CDFs. In practice, the CDFs are evaluated on a grid, reducing this to standard weighted K-means.

- **Interactive earnings model.** With estimated firm classes $\hat{k}(j)$, they estimate

$$Y_{it} = a_t(k_{it}) + b_t(k_{it})\alpha_i + X'_{it}c_t + \varepsilon_{it}, \tag{73}$$

where $k_{it} = k(j(i,t))$ is the class of worker i's firm at time t. This allows for *nonlinear interactions* between worker types and firm classes through $b_t(k_{it})\alpha_i$—different firm classes can load differently on worker heterogeneity. The AKM model is a special case with $b_t(k) = 1$ for all k and $K = J$.

- **Two-step estimation.** Step 1: classify firms via K-means (72). Step 2: estimate the model (73) by maximum likelihood, treating estimated firm classes as known. The classification error vanishes asymptotically and does not affect the limiting distribution of the second-step estimator.

4.3.1 Key Empirical Results

- Using Swedish administrative data (∼600,000 workers, ∼44,000 firms), Bonhomme et al. (2019) find strikingly different results from AKM:

	BLM	AKM (uncorrected)
Var(firm effects)/ Var(Y)	2.6%	32%
Corr(worker, firm effects)	49%	negative

- The AKM firm effects variance is heavily upward biased due to incidental parameter bias in short panels with low mobility. The negative AKM correlation is also a bias artifact.

- **Strong sorting, weak complementarities.** Worker heterogeneity explains ∼60% of the earnings variance, while firm heterogeneity (net of worker composition) explains only ∼3%. Despite weak wage complementarities, there is strong positive assortative matching: high-type workers tend to work in high-class firms (Corr ≈ 49%).

- These findings have important implications for understanding earnings inequality: most of the contribution of firms to inequality operates through *sorting* (which workers are employed by which firms) rather than through *differential pay* across firm types.

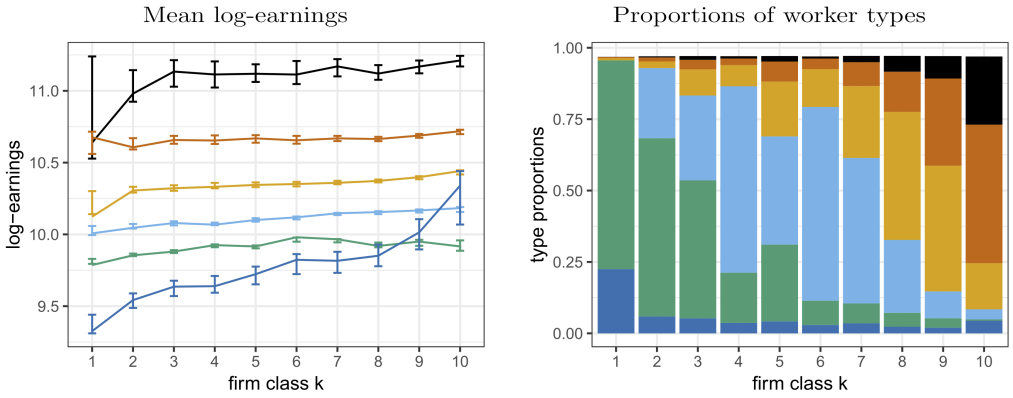


FIGURE 2.—Main parameter estimates of the static model. Notes: Estimates of the static model, on 2002–2004. In the left graph, we plot estimates of means of log-earnings, by worker type and firm class. We order the $K = 10$ firm classes (on the x-axis) by mean log-earnings. On the y-axis, we report estimates of mean log-earnings for the $L = 6$ worker types. In the right graph, we show estimates of the proportions of worker types in each firm class. In the left graph, the brackets indicate pointwise parametric bootstrap 2.5%–97.5% quantile bands (computed using 200 replications).

Figure 50: Main parameter estimates of the static model in Bonhomme et al. (2019). Left: mean log-earnings by worker type ($L = 6$) and firm class ($K = 10$), ordered by mean log-earnings. Right: proportions of worker types in each firm class, showing strong positive assortative matching. Source: Bonhomme et al. (2019, Figure 2).

References

Abowd, J. M., F. Kramarz, and D. N. Margolis (1999). High wage workers and high wage firms. *Econometrica 67*(2), 251–333.

Acemoglu, D., S. Johnson, J. A. Robinson, and P. Yared (2008). Income and democracy. *American Economic Review 98*(3), 808–842.

Ba, J. L., J. R. Kiros, and G. E. Hinton (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

Bai, J. (2003). Inferential theory for factor models of large dimensions. *Econometrica 71*(1), 135–171.

Bai, J. (2009). Panel data models with interactive fixed effects. *Econometrica 77*(4), 1229–1279.

Bai, J. and S. Ng (2002). Determining the number of factors in approximate factor models. *Econometrica 70*(1), 191–221.

Bartlett, P. L., M. I. Jordan, and J. D. McAuliffe (2006). Convexity, classification, and risk bounds. *Journal of the American Statistical Association 101*(473), 138–156.

Bengio, Y., P. Simard, and P. Frasconi (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks 5*(2), 157–166.

Bertsimas, D. and J. Dunn (2019). *Machine Learning under a Modern Optimization Lens*. Dynamic Ideas LLC.

Bertsimas, D., A. King, and R. Mazumder (2016). Best subset selection via a modern optimization lens. *The Annals of Statistics 44*(2), 813–852.

Bickel, P. J., Y. Ritov, and A. B. Tsybakov (2009, August). Simultaneous analysis of lasso and dantzig selector. *The Annals of Statistics 37*(4), 1705–1732.

Bishop, C. M. (1995). Training with noise is equivalent to Tikhonov regularization. *Neural Computation 7*(1), 108–116.

Bonhomme, S., T. Lamadon, and E. Manresa (2019). A distributional framework for matched employer–employee data. *Econometrica 87*(3), 699–739.

Bonhomme, S. and E. Manresa (2015). Grouped patterns of heterogeneity in panel data. *Econometrica 83*(3), 1147–1184.

Breiman, L. (2001). Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical science 16*(3), 199–231.

Breiman, L., J. Friedman, and C. J. Stone (1984). *Classification and Regression Trees*. Wadsworth.

Brown, T., B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. (2020). Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, Volume 33, pp. 1877–1901.

Candes, E. and T. Tao (2007). The Dantzig selector: Statistical estimation when p is much larger than n. *The Annals of Statistics 35*(6), 2313 – 2351.

Chen, T. and C. Guestrin (2016). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794.

Cheng, X., F. Schorfheide, and P. Shao (2023). Clustering for multi-dimensional heterogeneity with an application to production function estimation. *Working paper*.

Chetverikov, D. (2024). Tuning parameter selection in econometrics. *arXiv preprint arXiv:2405.03021*.

Chetverikov, D., Z. Liao, and V. Chernozhukov (2021). On cross-validated lasso in high dimensions. *The Annals of Statistics 49*(3), 1300–1317.

Cho, K., B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio (2014). Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems 2*(4), 303–314.

Devlin, J., M.-W. Chang, K. Lee, and K. Toutanova (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 4171–4186.

Efron, B., T. Hastie, I. Johnstone, and R. Tibshirani (2004). Least angle regression. *The Annals of Statistics 32*(2), 407 – 499.

Frank, I. E. and J. H. Friedman (1993). A statistical view of some chemometrics regression tools. *Technometrics 35*(2), 109–135.

Freund, Y. and R. E. Schapire (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences 55*(1), 119–139.

Gaillac, C. and J. L'Hour (2025). *Machine Learning for Econometrics*. Oxford University Press.

Gao, Z. and Z. Shi (2021). Implementing convex optimization in R: Two econometric examples. *Computational Economics 58*(4), 1127–1135.

Glorot, X. and Y. Bengio (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 249–256.

Hannan, E. J. and B. G. Quinn (1979). The determination of the order of an autoregression. *Journal of the Royal Statistical Society: Series B 41*(2), 190–195.

Hastie, T., R. Tibshirani, and J. Friedman (2009). *The Elements of Statistical Learning*. Springer New York.

Hastie, T., R. Tibshirani, and M. Wainwright (2015). *Statistical learning with sparsity: The Lasso and Generalizations*. Chapman & Hall/CRC.

Hazimeh, H. and R. Mazumder (2020). Fast best subset selection: Coordinate descent and local combinatorial optimization algorithms. *Operations Research 68*(5), 1517–1537.

He, K., X. Zhang, S. Ren, and J. Sun (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778.

Hochreiter, S. and J. Schmidhuber (1997). Long short-term memory. *Neural Computation 9*(8), 1735–1780.

Hornik, K., M. Stinchcombe, and H. White (1989). Multilayer feedforward networks are universal approximators. *Neural Networks 2*(5), 359–366.

James, G., D. Witten, T. Hastie, and R. Tibshirani (2021). *An introduction to statistical learning: with applications in R* (2nd ed.), Volume 103. Springer.

Jenatton, R., J.-Y. Audibert, and F. Bach (2011). Structured variable selection with sparsity-inducing norms. *The Journal of Machine Learning Research 12*, 2777–2824.

Ke, G., Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu (2017). LightGBM: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, Volume 30.

Kingma, D. P. and J. Ba (2015). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Lee, D. S. (2008). Randomized experiments from non-random selection in us house elections. *Journal of Econometrics 142*(2), 675–697.

Lugosi, G. and N. Vayatis (2004). On the Bayes-risk consistency of regularized boosting methods. *The Annals of Statistics 32*(1), 30 – 55.

Mehrabani, A. (2023). Estimation and identification of latent group structures in panel data. *Journal of Econometrics 235*(2), 1464–1482.

Moon, H. R. and M. Weidner (2015). Linear regression for panel with unknown number of factors as interactive fixed effects. *Econometrica 83*(4), 1543–1579.

Moon, H. R. and M. Weidner (2017). Dynamic linear panel regression models with interactive fixed effects. *Econometric Theory 33*(1), 158–195.

Obozinski, G., L. Jacob, and J.-P. Vert (2011). Group lasso with overlaps: the latent group lasso approach. *arXiv preprint arXiv:1110.0413*.

Pascanu, R., T. Mikolov, and Y. Bengio (2013). On the difficulty of training recurrent neural networks. *Proceedings of the 30th International Conference on Machine Learning (ICML)*, 1310–1318.

Prokhorenkova, L., G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin (2018). CatBoost: Unbiased boosting with categorical features. In *Advances in Neural Information Processing Systems*, Volume 31.

Qian, J. and L. Su (2016a). Shrinkage estimation of common breaks in panel data models via adaptive group fused lasso. *Journal of Econometrics 191*(1), 86–109.

Qian, J. and L. Su (2016b). Shrinkage estimation of regression models with multiple structural changes. *Econometric Theory 32*(6), 1376–1433.

Radford, A., J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever (2019). Language models are unsupervised multitask learners. *OpenAI Blog*.

Raffel, C., N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research 21*(140), 1–67.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). Learning representations by back-propagating errors. *Nature 323*(6088), 533–536.

Schwarz, G. (1978). Estimating the dimension of a model. *Annals of Statistics 6*(2), 461–464.

Shi, Z. (2016). Econometric estimation with high-dimensional moment equalities. *Journal of Econometrics 195*(1), 104–119.

Shi, Z., L. Su, and T. Xie (2025). ℓ_2-relaxation: With applications to forecast combination and portfolio analysis. *Review of Economics and Statistics 107*(2), 523–538.

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research 15*(56), 1929–1958.

Stock, J. H. and M. W. Watson (2002). Forecasting using principal components from a large number of predictors. *Journal of the American Statistical Association 97*(460), 1167–1179.

Su, L., Z. Shi, and P. C. Phillips (2016). Identifying latent structures in panel data. *Econometrica 84*(6), 2215–2264.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological) 58*(1), 267–288.

Tibshirani, R., M. Saunders, S. Rosset, J. Zhu, and K. Knight (2005). Sparsity and smoothness via the fused lasso. *Journal of the Royal Statistical Society Series B: Statistical Methodology 67*(1), 91–108.

Tibshirani, R. J. (2013). The lasso problem and uniqueness. *Electronic Journal of Statistics 7*(none), 1456 – 1490.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, Volume 30.

Wang, Y.-X., J. Sharpnack, A. J. Smola, and R. J. Tibshirani (2016). Trend filtering on graphs. *Journal of Machine Learning Research 17*(105), 1–41.

Xiong, R., Y. Yang, D. He, K. Zheng, S. Zheng, C. Xing, H. Zhang, Y. Lan, L. Wang, and T.-Y. Liu (2020). On layer normalization in the transformer architecture. In *International Conference on Machine Learning*, pp. 10524–10533.

Yuan, M. and Y. Lin (2006). Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society Series B: Statistical Methodology 68*(1), 49–67.

Zou, H. (2006). The adaptive lasso and its oracle properties. *Journal of the American statistical association 101*(476), 1418–1429.

Zou, H. and T. Hastie (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society Series B: Statistical Methodology 67*(2), 301–320.