Linear & Ridge Regression

Lecture Notes Consolidated with Claude AI — CS6140 Machine Learning

---

1 Setup and Notation

Consider houses described by 2 features (living area, # bedrooms), with labels being the sale price:

Living area (ft^2)	#bedrooms	price ($1000s)
2104	3	400
1600	3	330
2400	3	369
1416	2	232
3000	4	540
...	...	...

Each datapoint is a pair $(\mathbf{x}_i, y_i)$: $\mathbf{x}_i = (x_i^1, x_i^2)$ is the feature vector of training instance i (living area, #bedrooms) and y_i is its label (price). **Regression** asks: can we predict y from $\mathbf{x}$? **Linear regression** restricts the search to linear predictors

$$h_w(\mathbf{x}) = w^0 + w^1 x^1 + w^2 x^2,$$

where $w = (w^0, w^1, w^2)$ are the parameters we must fit. Assuming a dummy feature $x^0 = 1$ on every instance lets us fold the intercept into the sum:

$$h_w(\mathbf{x}) = \sum_{j=0}^{m} w^j x^j = \mathbf{w}^T \mathbf{x}, \tag{1}$$

where j ranges over the m real features (plus the dummy feature $j = 0$).

Remark. **Notation used throughout these notes.** Subscripts index *datapoints*: $i = 1, \ldots, N$ for **training** instances, t for a **test** instance. Superscripts index *features*: x_i^j = feature j of training point i; x_t^j = feature j of test point t; j or d range over $0, 1, \ldots, m$. When a feature value is squared we write it as $(x_i^j)^2$, never x_i^{j2}, to avoid confusing the exponent with a second superscript index.

Remark. **Running example.** From here on we anchor worked numbers in the real *Boston Housing* dataset used by HW1 Problem 1: $N = 506$ instances, $m = 13$ continuous features (crime rate, `RM` = average rooms per dwelling, distance to employment centers, ...), label $y =$ `MEDV` (median home value, $1000s). Small worked examples below use a handful of real rows of this data — not a synthetic toy set.

For N training instances collected in a design matrix $X \in \mathbb{R}^{N \times (m+1)}$ (rows = instances, first column = the dummy feature) and label vector $\mathbf{y} \in \mathbb{R}^N$:

$$X = \begin{bmatrix} x_1^0 & x_1^1 & \ldots & x_1^m \\ \vdots & \vdots & \ddots & \vdots \\ x_N^0 & x_N^1 & \ldots & x_N^m \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}.$$

2 Cost Function: Least Squares

What makes a fit "best"? We measure the error of h_w on the training data by the **sum of squared errors** (SSE):

$$J(w) = \frac{1}{2}\sum_{i=1}^{N}\left(h_w(\mathbf{x}_i) - y_i\right)^2 \tag{2}$$

(the factor $\frac{1}{2}$ is purely cosmetic — it cancels a 2 in the derivative later and does not change the minimizer). The best fit is $w^* = \arg\min_w J(w)$; the resulting regressor is called the **least squares fit**.

Key point: J is a sum over the N equations $y_i \approx w^0 + w^1 x_i^1 + \cdots + w^m x_i^m$. With $N > m+1$ this system is overdetermined (no exact solution) — least squares finds the w that minimizes the total squared disagreement instead of solving it exactly.

3 A First Closed-Form Solution: One Feature, by Hand

Before matrices, it is worth deriving the single-feature case ($h_w(x) = w^0 + w^1 x^1$) with ordinary calculus, so the normal-equations result in §4 is not a black box. We minimize

$$J(w^0, w^1) = \sum_{i=1}^{N}\left(w^0 + w^1 x_i^1 - y_i\right)^2$$

by setting both partial derivatives to zero:

$$\frac{\partial J}{\partial w^0} = 2\sum_i (w^0 + w^1 x_i^1 - y_i) = 0 \tag{3}$$

$$\frac{\partial J}{\partial w^1} = 2\sum_i (w^0 + w^1 x_i^1 - y_i)\, x_i^1 = 0 \tag{4}$$

Equation 3 divided by $2N$ says $w^0 = \bar{y} - w^1\bar{x}^1$, where $\bar{y}, \bar{x}^1$ are the sample means. Substituting into Equation 4 and solving for w^1 gives

$$w^1 = \frac{\sum_i (x_i^1 - \bar{x}^1)(y_i - \bar{y})}{\sum_i (x_i^1 - \bar{x}^1)^2}, \qquad w^0 = \bar{y} - w^1\bar{x}^1. \tag{5}$$

w^1 is exactly the sample covariance of x^1 and y divided by the sample variance of x^1 — the line passes through $(\bar{x}^1, \bar{y})$ with that slope.

Worked example (5 real Housing rows, feature `RM` only):

x_i^1 =`RM`	6.575	6.421	7.185	6.998	7.147
y_i =`MEDV`	24.0	21.6	34.7	33.4	36.2

$\bar{x}^1 = 6.8652$, $\bar{y} = 29.98$. Centering: $z_i = x_i^1 - \bar{x}^1 = (-.290, -.444, .320, .133, .282)$, and $\sum_i z_i(y_i - \bar{y}) = 9.174$, $\sum_i z_i^2 = 0.4808$. So

$$w^1 = \frac{9.174}{0.4808} = 19.08, \qquad w^0 = 29.98 - 19.08(6.8652) = -101.0.$$

(This particular slope is unrealistically steep — with only 5 points that are nearly identical in `RM`, the tiny denominator $\sum z_i^2 = 0.48$ makes the fit unstable. We revisit this exact example in §6.3 to see how ridge regression tames it.)

Key point: This is exactly HW1's optional Problem 9 ("1-D Normal Equations"): specialize the general $\mathbf{w} = (X^TX)^{-1}X^T\mathbf{y}$ result (§4) to one feature, or (as done here) re-derive a, b for $h(x) = ax+b$ from scratch with plain calculus — the problem explicitly allows either route. Working through it yourself is still the point; this derivation is here so you have a reference to check your own algebra against, not a replacement for doing it.

4 Least Squares via Normal Equations

SSE is quadratic in w, so linear regression admits an exact, non-iterative **closed-form** solution (contrast this with gradient descent, a general *iterative* optimizer covered later in §9, which works even when no closed form exists). Write the predictor and error for all N points at once in matrix form:

$$E = X\mathbf{w} - \mathbf{y}, \qquad J(w) = \tfrac{1}{2}E^TE = \tfrac{1}{2}(X\mathbf{w} - \mathbf{y})^T(X\mathbf{w} - \mathbf{y}).$$

Expanding and differentiating (treating ∇_w here the way you would an ordinary scalar derivative — product rule and all):

$$\begin{aligned} J(w) &= \tfrac{1}{2}\big(\mathbf{w}^TX^TX\mathbf{w} - 2\mathbf{w}^TX^T\mathbf{y} + \mathbf{y}^T\mathbf{y}\big) \\ \nabla_w J(w) &= X^TX\mathbf{w} - X^T\mathbf{y} \end{aligned}$$

using the two rules $\nabla_w(\mathbf{w}^TA\mathbf{w}) = 2A\mathbf{w}$ (for symmetric A, here $A = X^TX$) and $\nabla_w(\mathbf{w}^T\mathbf{b}) = \mathbf{b}$. Setting the gradient to zero:

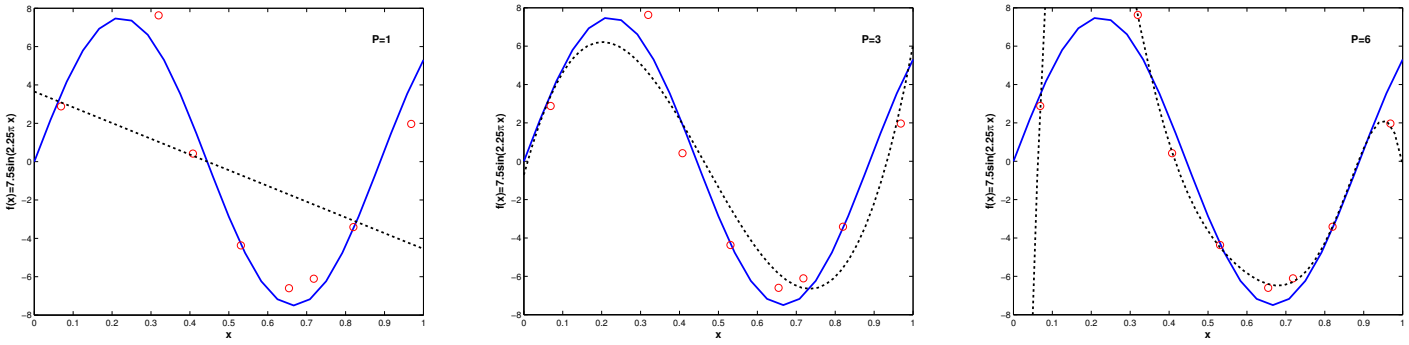
$$X^TX\mathbf{w} = X^T\mathbf{y} \quad \Longrightarrow \quad \boxed{\mathbf{w} = (X^TX)^{-1}X^T\mathbf{y}} \tag{6}$$

the **normal equations**. J is convex in w (it is a sum of squares of linear functions of w: $\partial^2 J/\partial(w^j)^2 = \sum_i (x_i^j)^2 \geq 0$ for every j), so this stationary point is guaranteed to be the global minimum, not just a local one.

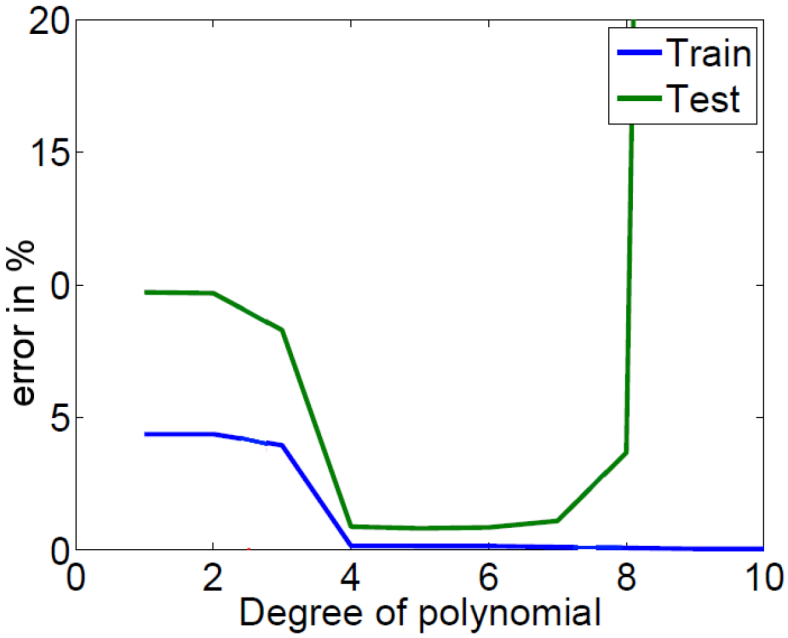
Remark. This derivation is a bit waivy. We differentiated $\mathbf{w}^TX^TX\mathbf{w}$ and $\mathbf{w}^TX^T\mathbf{y}$ as if vector/matrix calculus obeyed exactly the same product rule as 1-D calculus. It happens to give the right answer here, but "happens to" is not a proof — doing it rigorously requires actual matrix-derivative machinery (trace identities). A companion note, **REGRESSION_ADVANCED**, gives that treatment in full, along with a probabilistic (MLE) justification for why squared error is the right loss in the first place; neither is needed for what follows here, which only uses the boxed result, Equation 6.

5 Overfitting and the Need to Regularize

Consider fitting a degree-k polynomial to noisy samples of $f(x) = 7.5\sin(2.5\pi x)$, i.e. $t = f(x) + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma^2)$:



Degree 1 (*underfits* — too simple to capture the curve), degree 3 (a good match to the true shape), degree 6 (*overfits* — passes through every training point exactly, but by chasing noise, not signal).



As model complexity grows, training error keeps falling, but test error falls, bottoms out, then rises again — the same U-shape you saw for tree depth in the Decision Trees lecture.

Remark. **Occam's Razor, again.** We already used this principle once, to justify pruning decision trees back after growing them fully: prefer the simplest hypothesis consistent with the data, because needlessly complex hypotheses tend to fit noise rather than signal. For decision trees, complexity was tree depth/# leaves. For linear regression, complexity is naturally tied to the *number of features* and to how *large* the fitted weights w^j are allowed to grow: with many features (or few, nearly-collinear datapoints — exactly what happened in the §3 worked example), the least-squares weights can swing wildly in response to small changes in the training data (high variance), which is itself a symptom of overfitting.

6 Ridge Regression

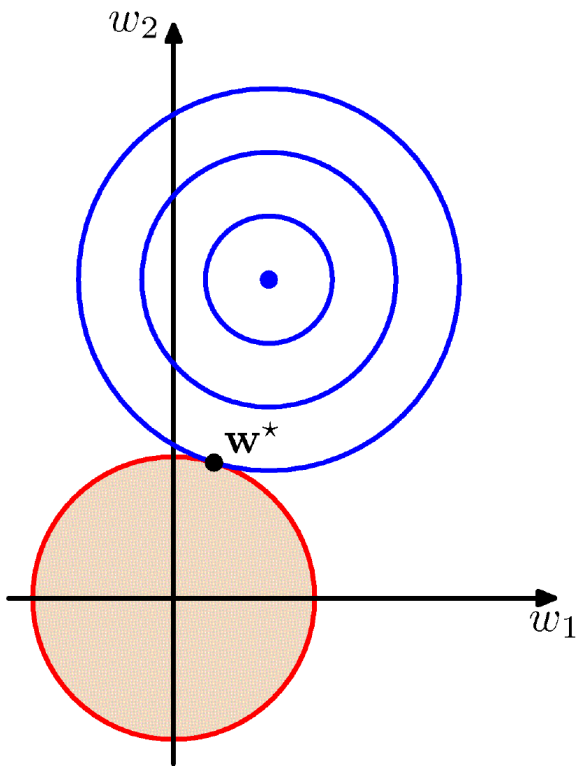
To apply Occam's Razor to linear regression, we directly restrict how large the weights can get. Augment the least-squares objective with a constraint:

$$\arg\min_{w\in\mathbb{R}^m} \sum_{i=1}^{N} \left(y_i - \mathbf{w}^T\mathbf{x}_i\right)^2 \quad \text{subject to} \quad \sum_{j=1}^{m} (w^j)^2 \leq c. \tag{7}$$

(Note the sum in the constraint runs only over the m *real* features $j = 1, \ldots, m$, not the dummy feature $j = 0$ — more on why below.) Forming the Lagrangian of Equation 7 turns the constrained problem into an unconstrained one, for some $\lambda \geq 0$ in one-to-one correspondence with c:

$$w_{ridge} = \arg\min_{w} J_\lambda(w) = \sum_{i=1}^{N} \left(y_i - \mathbf{w}^T \mathbf{x}_i\right)^2 + \lambda \sum_{j=1}^{m} (w^j)^2 = \sum_{i=1}^{N} \left(y_i - \mathbf{w}^T \mathbf{x}_i\right)^2 + \lambda \|\mathbf{w}\|_2^2. \tag{8}$$

Including an L_2 penalty on the weights this way is called **Tikhonov regularization**, and Equation 8 is **ridge regression**.



For $\mathbf{w} \in \mathbb{R}^2$: the red disk is the constraint region $\{(w^1)^2 + (w^2)^2 \leq c\}$; the ellipses are contours of the unconstrained SSE objective. $\mathbf{w}^*$ is the ridge solution — the SSE contour pulled just far enough to touch the constraint region, rather than reaching its unconstrained optimum.

6.1 Solving Ridge: Why the Intercept Needs Special Care

If w^0 were included in the penalty $\sum_j (w^j)^2$, the solution would depend on an arbitrary choice — e.g. shifting every label y_i by a constant c (say, quoting home prices in a currency with a different baseline) would change which w^0 is "large", even though it has nothing to do with model complexity. So by convention w^0 (the intercept) is *never* penalized. The standard fix: drop the dummy feature from the design matrix, **center** every real feature (subtract its training mean), and handle the intercept separately. Concretely, Z is *the same design matrix X from §1*, with two changes: its dummy column ($x_i^0 = 1$) is dropped, and each remaining column j is re-centered around its own training mean, $z_i^j = x_i^j - \bar{x}^j$ (so Z is $N \times m$, not $N \times (m+1)$). We assume y is also centered (or, equivalently, we are solving for the centered part of w only — the intercept w^0 is recovered separately below):

6.2 Ridge Closed Form (same "quick" style as §4)

$$J_\lambda(w) = (Z\mathbf{w} - \mathbf{y})^T(Z\mathbf{w} - \mathbf{y}) + \lambda\,\mathbf{w}^T\mathbf{w}.$$

Differentiating the same quick way as Equation 6 (again: not rigorously justified without matrix calculus — see REGRESSION_ADVANCED for that treatment):

$$\nabla_w J_\lambda(w) = 2Z^TZ\mathbf{w} - 2Z^T\mathbf{y} + 2\lambda\mathbf{w} = 2(Z^TZ + \lambda I)\mathbf{w} - 2Z^T\mathbf{y}.$$

Setting to zero:

$$\boxed{\mathbf{w}_{ridge} = (Z^TZ + \lambda I)^{-1}Z^T\mathbf{y}}, \qquad w^0 = \bar{y}. \tag{9}$$

Algorithm: Ridge Regression

1. Center each feature: $z_i^j = x_i^j - \bar{x}^j$ for $j = 1, \ldots, m$ (using the *training* means $\bar{x}^j$).
2. Set the intercept: $w^0 = \bar{y} = \frac{1}{N}\sum_i y_i$.
3. Choose $\lambda \geq 0$ (a hyperparameter — typically by cross-validation).
4. Solve $\mathbf{w}_{ridge} = (Z^TZ + \lambda I)^{-1}Z^T\mathbf{y}$ for the remaining m weights.

Key point: $\lambda = 0$ recovers ordinary least squares exactly. As $\lambda \to \infty$, $\mathbf{w}_{ridge} \to \mathbf{0}$ (predict the mean everywhere) — λ is a dial between "trust the data fully" and "ignore the features entirely," the same bias–variance knob that tree depth was for Decision Trees.

6.3 Worked Example, Continued

Recall the unstable 5-point `RM`→`MEDV` fit from §3: centered values $z_i = (-.290, -.444, .320, .133, .282)$, centered labels summing to $\sum_i z_i(y_i - \bar{y}) = 9.174$, and $\sum_i z_i^2 = 0.4808$. Unregularized ($\lambda = 0$): $w^1 = 9.174/0.4808 = 19.08$ (the same unstable slope as before). With $\lambda = 1$:

$$w^1_{ridge} = \frac{9.174}{0.4808 + 1} = \frac{9.174}{1.4808} = 6.20,$$

a much more moderate, plausible slope (intercept unaffected: $w^0 = \bar{y} = 29.98$ either way). **This is exactly what ridge is for**: when data is scarce or nearly collinear (here, all 5 houses have almost the same `RM`, so $\sum_i z_i^2$ is tiny), the unregularized denominator is small and the fit is dominated by noise; adding λ to that denominator stabilizes it. (As a bonus, this also fixes a numerical issue: Z^TZ can be singular or near-singular when features are collinear or $N < m$; $Z^TZ + \lambda I$ is always invertible for $\lambda > 0$.)

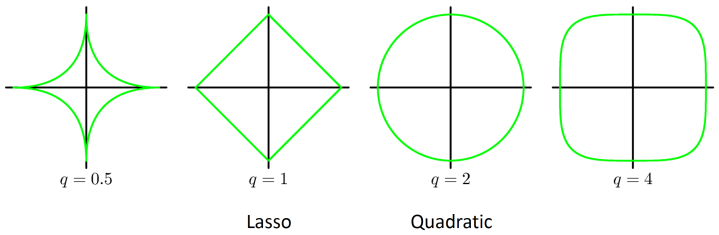
Remark. Everything above justifies ridge purely as a constrained-optimization fix for high-variance weights. That is not the only way to arrive at Equation 9: REGRESSION_ADVANCED gives a Bayesian (MAP) derivation, an entirely different-looking derivation via *data augmentation*, and a short proof that $\hat{\mathbf{w}}_{ridge}$ is a *biased* estimator (unlike OLS) — worth knowing, not needed for HW1.

7 Generalizing: L_q Regularizers and Lasso

Ridge is the $q = 2$ member of a more general family:

$$\hat{w}_{reg} = \arg\min_{w} \ \text{Loss}(X, w) + \|\mathbf{w}\|_q,$$

where $\text{Loss}(X, w) = \sum_i (y_i - w^0 - \sum_j w^j x_i^j)^2$ (on centered features, as above).



Constraint regions for different q. $q = 2$ (a disk) gives ridge. $q = 1$ (a diamond) gives **Lasso**.

Remark. **Lasso ($q = 1$).** The diamond-shaped constraint region has corners on the axes; SSE contours tend to first touch it at a corner, which drives some w^j *exactly* to zero — Lasso performs automatic feature selection, unlike ridge (which shrinks weights smoothly toward zero but essentially never sets them exactly to zero). The tradeoff: Lasso has no closed form like Equation 9 *jointly* in all m weights, and needs an iterative solver. *Next module (regularized classification / feature selection) returns to this.*

Remark. This geometric picture is one way to see why Lasso zeros out coefficients. The companion note **REGRESSION_ADVANCED** gives a second, complementary intuition (via gradient descent dynamics), a Bayesian (Laplace-prior) derivation of the Lasso penalty itself, a closed form Lasso *does* have (per-coordinate, via soft-thresholding), and pointers to the classic algorithms (LARS, coordinate descent) that solve it in practice — none required for HW1/HW2.

8 Implementation Notes (HW1)

HW1 Problem 1 asks you to implement closed-form linear regression and ridge regression from scratch on the Housing dataset, and to compare against `sklearn`'s `LinearRegression` and `Ridge` as library baselines.

- **Unregularized fit**: build X with the dummy column ($x_i^0 = 1$ for all i), solve Equation 6 directly (e.g. via a linear solver, not an explicit matrix inverse, for numerical stability).
- **Ridge fit**: build Z *without* a dummy column, center every feature using the **training** means (§6's Algorithm), and solve Equation 9.
- **Sanity check**: run your ridge code with $\lambda = 0$ — it must reproduce the unregularized solution (up to the different handling of the intercept). If it doesn't, the bug is in the centering step, not the linear algebra.

Common bugs (all silent — wrong numbers, not a crash):

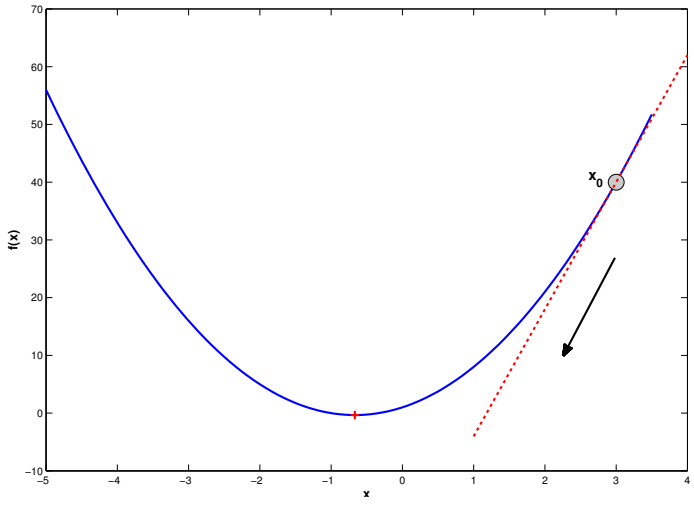
- **Test-set centering**: test features must be centered using the *training* means $\bar{x}^j$, never the test set's own mean — you don't get to look at test labels or re-derive test statistics at prediction time. Centering test data with test-set means silently leaks information and gives optimistic error estimates.
- **Penalizing the intercept**: if you keep the dummy column *and* penalize all of w uniformly (skipping the centering trick), you get a different, non-standard answer that will not match `sklearn`'s `Ridge` default behavior.
- **Feature scaling**: the ridge penalty $\lambda \sum_j (w^j)^2$ treats every feature's weight identically, but features on very different scales (e.g. `NOX` $\in [0, 1]$ vs. `TAX` $\in [100, 700]$) will be penalized very unevenly unless standardized (unit variance) in addition to centered — `sklearn`'s `Ridge` does *not* do this for you automatically.
- **Explicit matrix inverse**: computing $(Z^T Z + \lambda I)^{-1}$ explicitly works but is numerically worse and slower than solving the linear system $(Z^T Z + \lambda I)\mathbf{w} = Z^T \mathbf{y}$ directly (e.g. `numpy.linalg.solve`); for the unregularized case with a large or ill-conditioned $X^T X$, watch for this being the source of small mismatches against the library baseline.

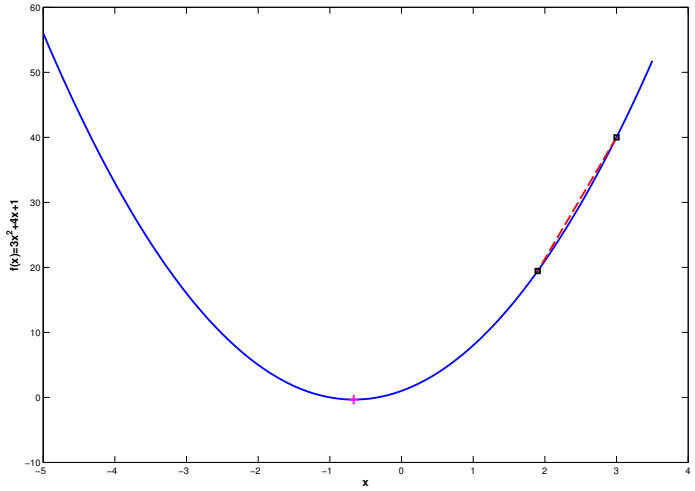
9 Gradient Descent: A General Iterative Optimizer

Every objective so far (SSE, ridge's J_λ) happened to have a clean closed-form minimizer, because both are quadratic in w. That is the exception, not the rule. **Gradient descent** is a general iterative method for minimizing *any* differentiable function — it makes no use of quadratic structure and needs no matrix inverse at all. We introduce it here, on the one objective where we already know the exact answer (§4), precisely so it can be sanity-checked against a known ground truth. But its real value is ahead: **gradient descent (and variants of it) is the optimization method you will use for the rest of the course** — logistic regression, the perceptron, and neural networks all have *no* closed form, and are fit by exactly the same "compute the gradient, step against it, repeat" recipe developed below.

9.1 Minimizing a Function

Consider $f(x) = 3x^2 + 4x + 1$. Its minimizer solves $f'(x) = 6x + 4 = 0 \Rightarrow x^* = -\frac{2}{3}$ (confirmed by $f''(x) = 6 > 0$). Geometrically: at any point x_0, the derivative is the slope of the tangent line; moving *against* the slope decreases f.





Left: $f(x) = 3x^2 + 4x + 1$ and the tangent at x_0; moving against its slope heads toward the minimum. Right: the next point x_1 reached this way, closer to the minimum.

9.2 The Gradient Vector and the Algorithm

For a function of several parameters, e.g. $J(w) : \mathbb{R}^{m+1} \to \mathbb{R}$, the gradient vector collects all partial derivatives,

$$\nabla_w J = \left(\frac{\partial J}{\partial w^0}, \ldots, \frac{\partial J}{\partial w^m} \right)^T,$$

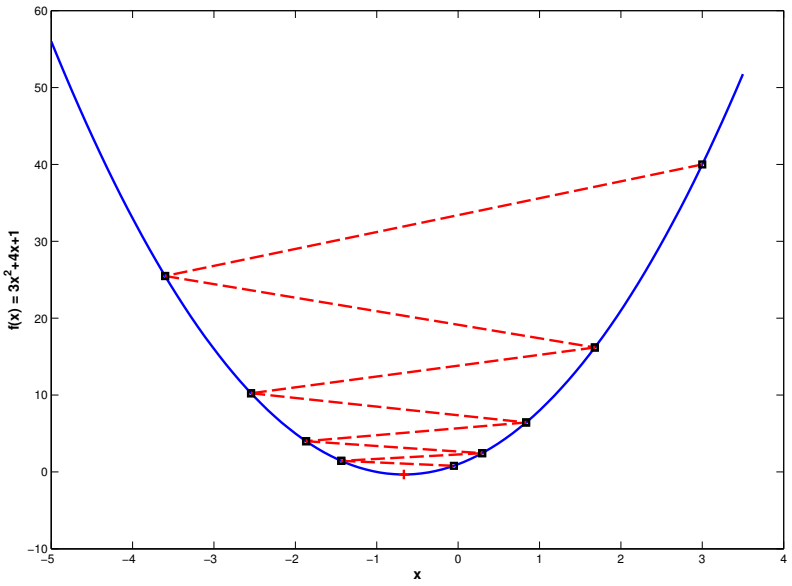
and points in the direction of steepest *increase*. Gradient descent repeatedly steps against it — note that nothing in this algorithm below is specific to SSE or even to regression; J can be *any* differentiable objective:

Algorithm: Gradient Descent

---

1. Pick a starting point w_0 and a learning rate $\eta > 0$.
2. Repeat until convergence: $w_{k+1} = w_k - \eta \nabla_w J(w_k)$.

Remark. **η is a step size, not a probability or a tolerance.** Too small $\Rightarrow$ slow crawl to the minimum; too large $\Rightarrow$ overshoot and oscillate (or diverge) instead of converging, as in the figure below.



A learning rate that is too large causes gradient descent to oscillate around the minimum instead of settling into it.

When to stop? Any of: a fixed iteration budget; the objective's decrease between steps falls below a threshold τ ($J(w_k) - J(w_{k+1}) < \tau$); the gradient norm falls below a tolerance ϵ ($\|\nabla_w J(w_k)\| < \epsilon$, the theoretical stopping point since $\nabla_w J = 0$ at any minimum).

9.3 Gradient Descent for Linear Regression — a Worked Sandbox

Specializing the general recipe to $J(w) = \frac{1}{2}\sum_i (h_w(\mathbf{x}_i) - y_i)^2$: differentiate one component at a time (drop the sum over i momentarily, i.e. look at a single point):

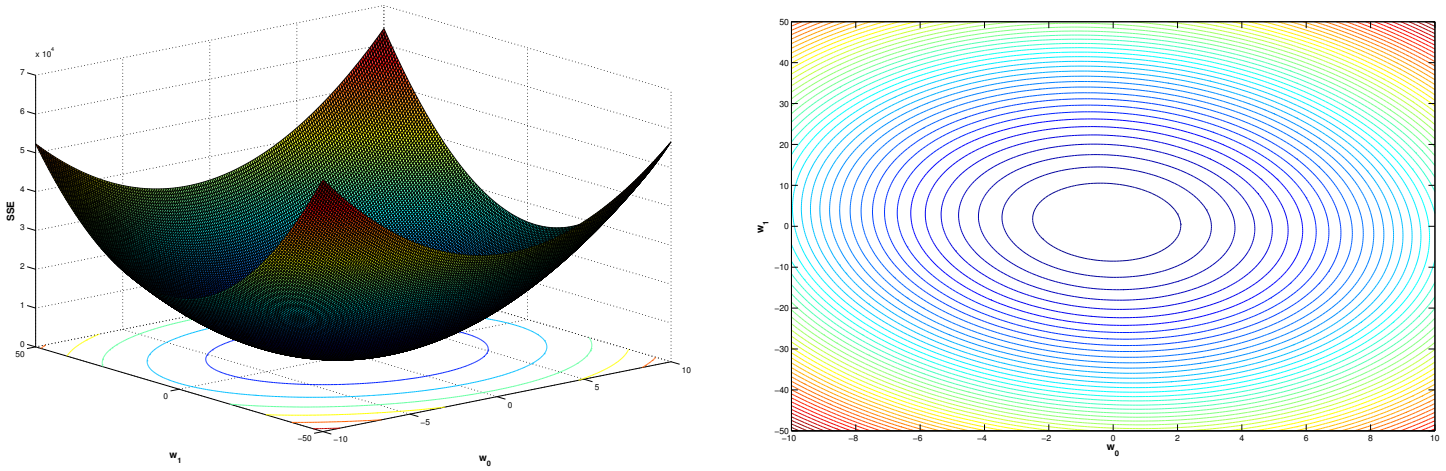
$$\frac{\partial}{\partial w^j}\ \tfrac{1}{2}\big(h_w(\mathbf{x}) - y\big)^2 = \big(h_w(\mathbf{x}) - y\big)\ \frac{\partial}{\partial w^j}\Big(\sum_d w^d x^d - y\Big) = \big(h_w(\mathbf{x}) - y\big)\, x^j.$$

Summed over all N training points, this gives the full gradient and the **batch gradient descent** update rule, applied to every $j = 0, \ldots, m$ simultaneously:

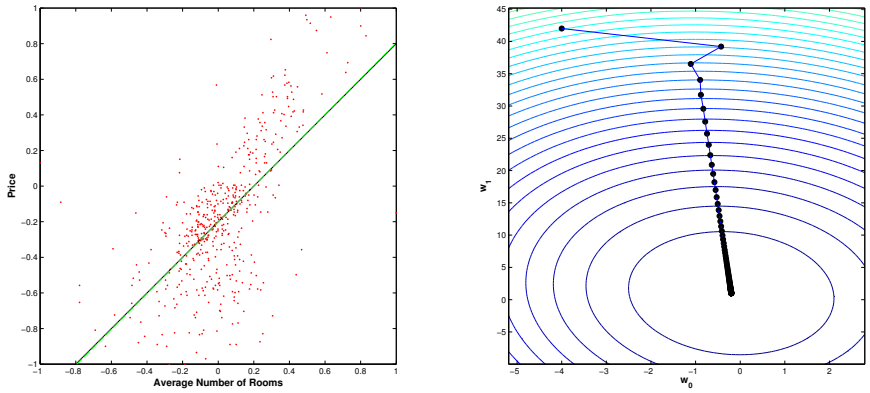
$$w^j \leftarrow w^j - \eta \sum_{i=1}^{N} \big(h_w(\mathbf{x}_i) - y_i\big)\, x_i^j. \tag{10}$$

Remark. **Batch vs. stochastic.** Equation 10 uses *all* N points for one update — correct, but expensive if N is huge. **Stochastic (online) gradient descent** instead loops over training points one at a time, updating $w^j \leftarrow w^j - \eta(h_w(\mathbf{x}_i) - y_i)x_i^j$ for a single random i per step. It converges faster in wall-clock time on large datasets (many cheap noisy steps beat few expensive exact ones), though in principle it can "dance" around the optimum rather than settling exactly onto it. This batch/stochastic distinction, unlike the closed form, *does* carry over unchanged to every other objective in the course.

Worked example (Housing, 1 feature `RM`): the SSE surface as a function of (w^0, w^1) is a bowl (a paraboloid, since J is quadratic); gradient descent slides down it to the same w found by Equation 5 — our known ground truth.



SSE as a function of (w^0, w^1) for Housing (`RM`→`MEDV`): 3-D surface (left) and contours (right, blue = low SSE).



A gradient descent run on this surface: the fitted line (left) and the descent trajectory on the contour plot (right).

Key point: J being convex here (§4) means gradient descent cannot get stuck in a bad local minimum on *this particular* objective — every local minimum is the global minimum. This is a special feature of SSE, not of gradient descent in general: on the non-convex objectives later in the course (e.g. neural network training), the same update rule $w_{k+1} = w_k - \eta \nabla_w J(w_k)$ is still exactly what you run, but it is only guaranteed to reach *a* local minimum, not necessarily the global one.

10 Next Module: From Regression to Classification

Everything above assumed a numeric label $y \in \mathbb{R}$. When labels are instead classes (e.g. $y \in \{0, 1\}$), running linear regression directly on y is a poor fit: nothing constrains $h_w(\mathbf{x})$ to land in $[0, 1]$, and the SSE-optimal line generally does not correspond to a sensible decision boundary. The fix — **logistic regression** — composes the same linear score $\mathbf{w}^T\mathbf{x}$ with the **sigmoid** function $g(z) = \frac{1}{1+e^{-z}}$, giving $h_w(\mathbf{x}) = g(\mathbf{w}^T\mathbf{x})$, and replaces squared error with a different (log-likelihood) objective suited to class labels.

Key point: *Next module* (Week 3, HW2) develops logistic regression, gradient descent/Newton's method for it, and the perceptron — all reusing the linear score $\mathbf{w}^T\mathbf{x}$ and the gradient-descent machinery of §9 unchanged; only the output transformation and loss function change.

Remark. Everything in this note is exactly what HW1 and HW2 need — no more. A companion note, **REGRESSION_ADVANCED**, collects the deeper story that got left out along the way: a rigorous matrix-calculus derivation of the normal equations, the probabilistic (MLE) justification for squared error, a Bayesian view of ridge and Lasso, an alternate data-augmentation derivation of ridge, a proof that ridge is biased, Lasso's per-coordinate closed form (soft-thresholding), and pointers to the classic algorithms and papers behind all of it. None of it is required here; all of it is worth knowing eventually.