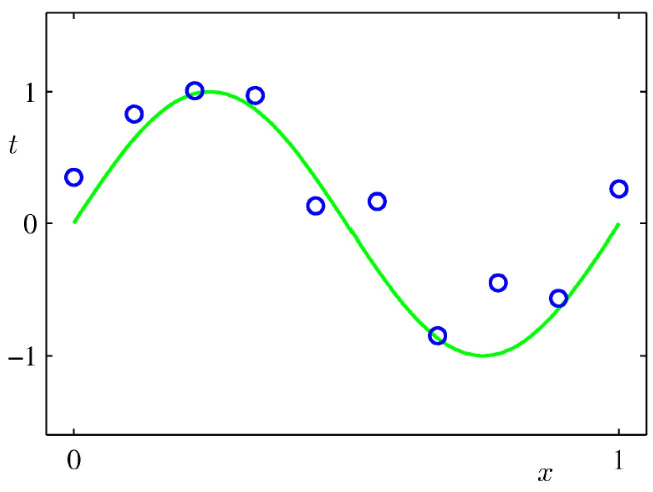
Advanced ML — Lecture 2: Linear Regression

Charalampos Giannakis

Polynomial curve fitting — motivation



We start with a simple but important example: suppose we have data points sampled from a sinusoidal function with some added noise. Concretely, imagine we generate 10 data points from

$$t = \sin(2\pi x) + \text{disturbance}.$$

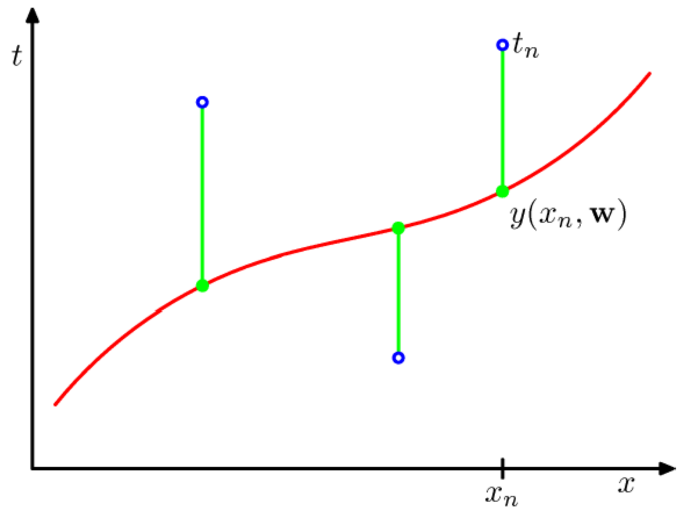
This reflects a real-world scenario where observations are not perfect: we have an underlying true function, but our measurements are corrupted by noise. The challenge is: How can we build a model that captures the relationship between x and t, despite the disturbances?

A key idea is to approximate the function with a polynomial. Many functions can be expressed as power series. For example, the Taylor expansion of the sine function is

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} + \cdots$$

which reminds us that even complex curves can be expressed using polynomials of increasing degree.

The polynomial model and the error we minimize



In general, we define our polynomial model as

$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M = \sum_{j=0}^{M} w_j x^j,$$

where M is the order of the polynomial and w_j are the coefficients we need to determine.

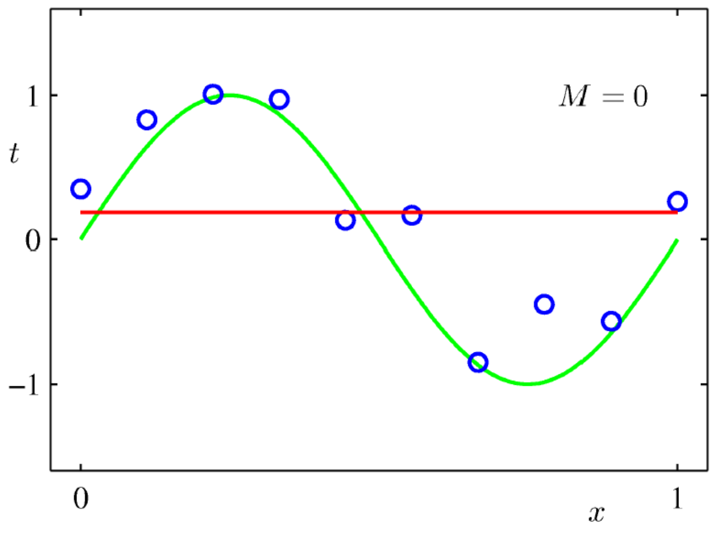
To measure how well the model predicts the observed targets t_n, we use the **sum-of-squares error**:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \{ y(x_n, \mathbf{w}) - t_n \}^2.$$

Minimizing $E(\mathbf{w})$ gives the polynomial curve that best fits the data in the least-squares sense.

Exploring model capacity with polynomial order

Order 0 (constant model)

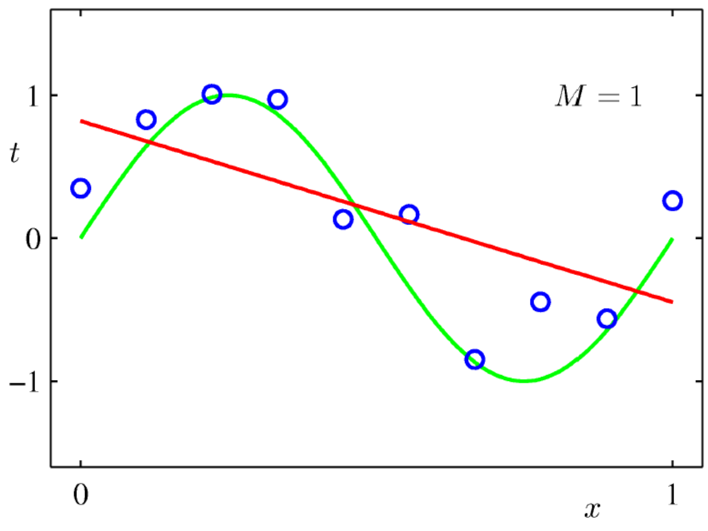


We start with the simplest possible model: a flat line.
It ignores any trend in the data and predicts the same value for every input (x).
This is the “high-bias” extreme—very simple, very constrained.

Model. $y(x, \mathbf{w}) = w_0$

Intuition. If the data oscillate (like a sine wave), a constant can only match the *Expected Value* level.
Errors remain large because the model cannot bend to follow the signal.

Order 1 (linear model)

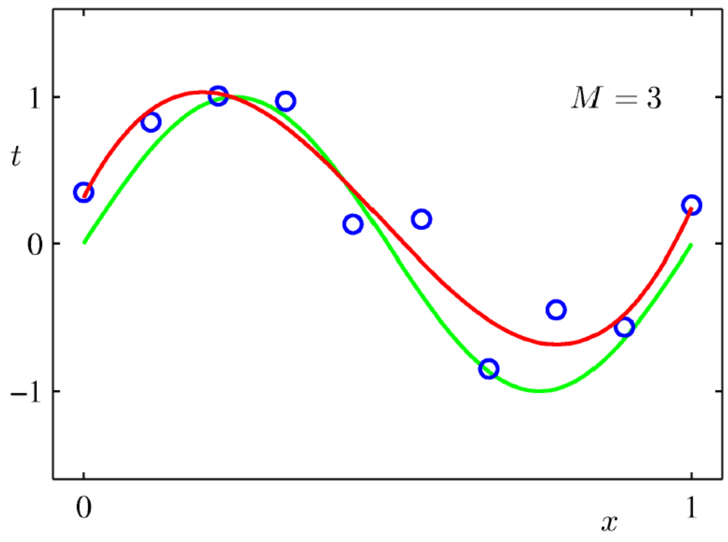


Allow one slope. Now the model can tilt and capture a global upward or downward trend, but still cannot follow curvature.
Bias decreases compared to the constant model, yet the fit is still too rigid for periodic structure.

Model. $y(x, \mathbf{w}) = w_0 + w_1 x$

Intuition. Good as a baseline when the signal is roughly linear; otherwise, systematic errors remain because the model cannot capture bends.

Order 3 (cubic model)

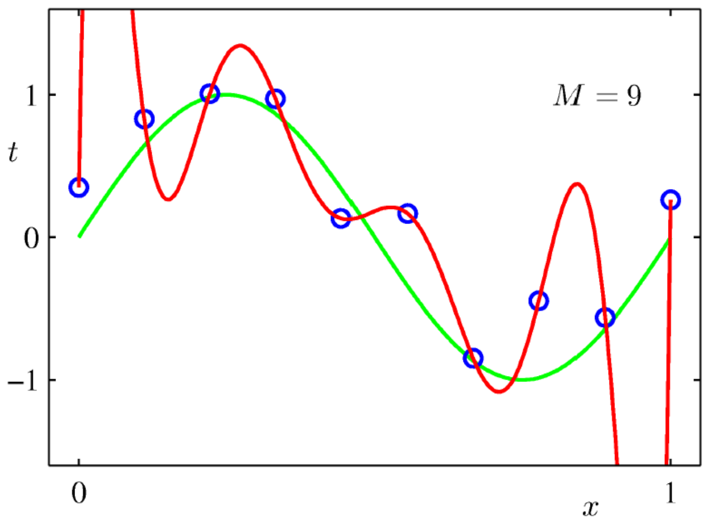


With quadratic and cubic terms, the curve can bend and flex.
For our noisy sinusoidal data, a cubic already starts to track the main rise and fall without chasing every noisy fluctuation.

Model. $y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + w_3 x^3$

Intuition. This is a more balanced regime: enough capacity to capture curvature, but still restrained, so variance is moderate.

Order 9 (high-degree model)



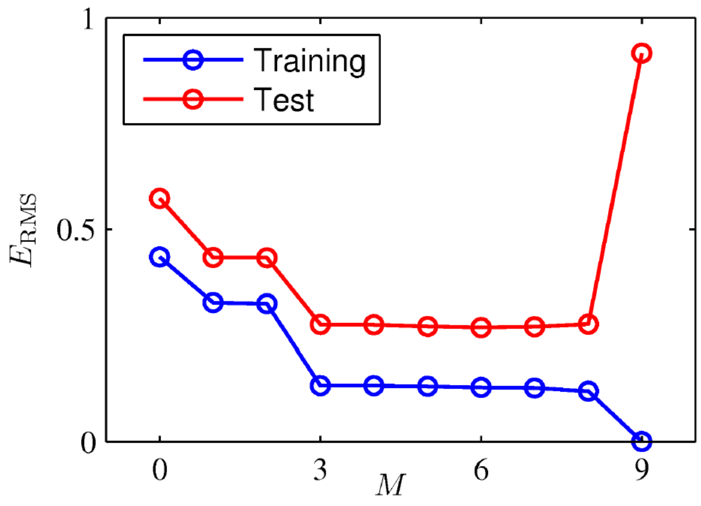
With many degrees of freedom, the polynomial can pass very close to each training point—including noise.
The red curve wiggles aggressively: it *fits the training set* extremely well but behaves wildly between points and near the boundaries.

Model. $y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \ldots + w_9 x^9$

Intuition. This is the “high-variance” extreme—low training error, high risk of **overfitting**.
It motivates the need for **regularization** and for choosing model complexity carefully.

When models become too flexible: overfitting

Training vs. test error



To understand overfitting, we track how error behaves as we increase the polynomial order (M).

We measure error using the **root mean square (RMS)**:

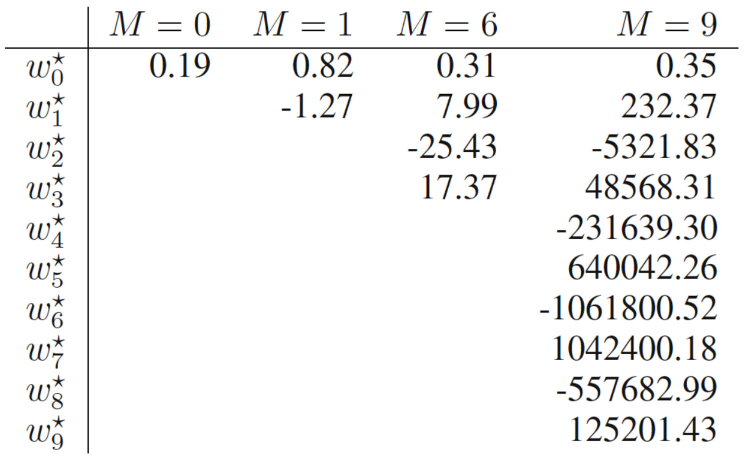
$$E_{\text{RMS}} = \sqrt{\frac{2E(\mathbf{w}^*)}{N}}$$

Here, $E(w^*)$ is the minimized sum-of-squares error and N is the number of data points.

- On the **training set** (blue curve), the error keeps going down as we add more terms.
- But on the **test set** (red curve), after a certain point the error *increases*.

This means the model has started to memorize noise in the training data rather than learning the true underlying function.
That is the essence of **overfitting**.

What happens to the coefficients?



	$M = 0$	$M = 1$	$M = 6$	$M = 9$
$w_0^\star$	0.19	0.82	0.31	0.35
$w_1^\star$		-1.27	7.99	232.37
$w_2^\star$			-25.43	-5321.83
$w_3^\star$			17.37	48568.31
$w_4^\star$				-231639.30
$w_5^\star$				640042.26
$w_6^\star$				-1061800.52
$w_7^\star$				1042400.18
$w_8^\star$				-557682.99
$w_9^\star$				125201.43

Looking at the actual values of the fitted coefficients tells the same story.

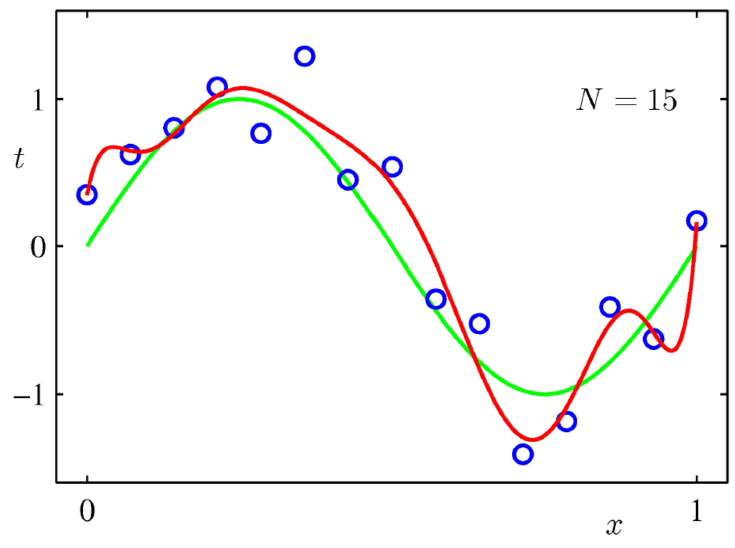
- For small (M), coefficients are small and stable.
- For higher orders like (M = 6) or (M = 9), the coefficients explode to extremely large positive and negative numbers.

This instability is a direct sign that the model is trying too hard to wiggle through every training point.
It finds a curve that *looks perfect on training data* but generalizes very poorly.

Takeaway. More complexity is not always better. There's a trade-off:
- Too simple $\rightarrow$ underfitting (high bias).
- Too complex $\rightarrow$ overfitting (high variance).
- The sweet spot is somewhere in between, and we'll later see that **regularization** helps us control this balance.

The effect of dataset size

Small dataset: (N = 15)

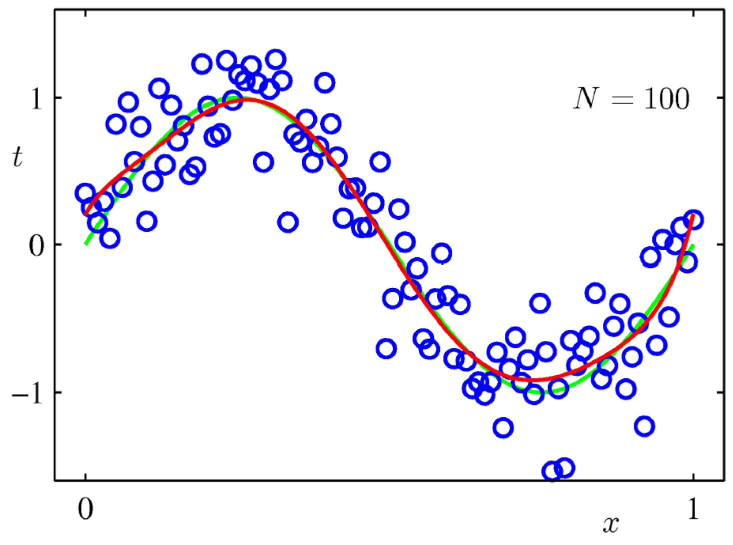


Here we fit a **9th-order polynomial** using only **15 points**.
The model has a lot of flexibility compared to the limited data available.

- The red curve twists and bends, trying to follow each of the few noisy points.
- As a result, it looks unstable and does not represent the smooth sine wave well.

This is a classic case where too much model complexity meets too little data.

Larger dataset: (N = 100)



Now we fit the *same* 9th-order polynomial, but with **100 points** instead.
Notice how different the result is:

- With more data, the curve (red) stays close to the true underlying function (green).
- The polynomial still has high capacity, but the abundance of data **anchors it** and prevents wild oscillations.

Takeaway. The impact of overfitting depends not only on model complexity but also on the **amount of data** available.
- High complexity with few data points $\rightarrow$ overfitting.
- High complexity with many data points $\rightarrow$ still manageable, because the data provide enough constraints.

This highlights why in practice, collecting more data can be just as important as designing the right model.

Regularization: controlling complexity

Why regularization?

When we used high-degree polynomials, we saw the coefficients w_j exploding to very large values. This instability is a hallmark of overfitting.

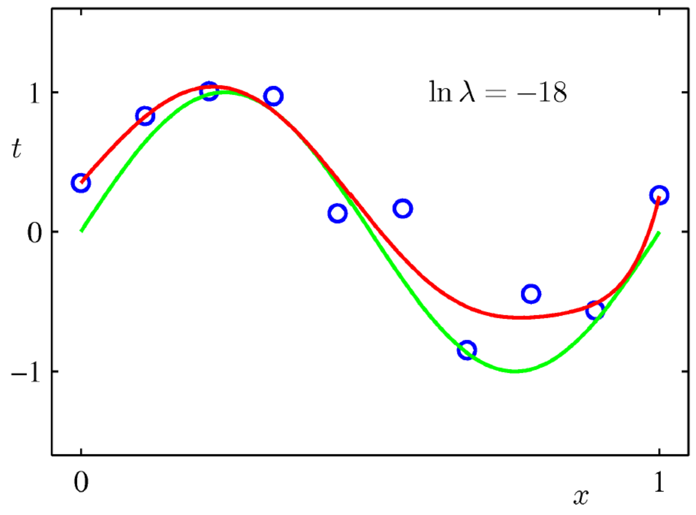
To counter this, we add a penalty that discourages large coefficients.
The new error function becomes:

$$\tilde{E}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{t_n - y(x_n, \mathbf{w})\}^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

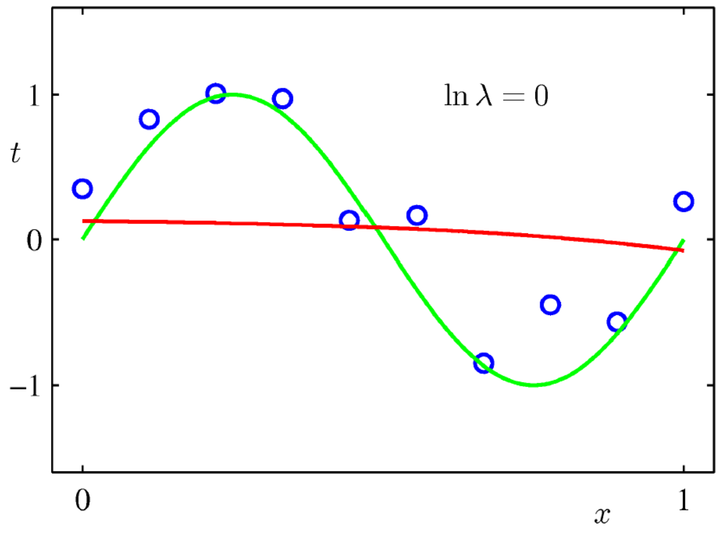
- The first term is the data fitting error.
- The second term is the **regularization penalty**, controlled by ( ).

Here, λ is a hyperparameter:
- Small $\lambda \rightarrow$ model behaves like standard least squares.
- Large $\lambda \rightarrow$ coefficients shrink toward zero.

Regularization in practice



When $\ln\lambda = -18$, the penalty is tiny.
The curve still fits the training data closely and can wiggle quite a lot.
Regularization is present, but not strong enough to really simplify the model.

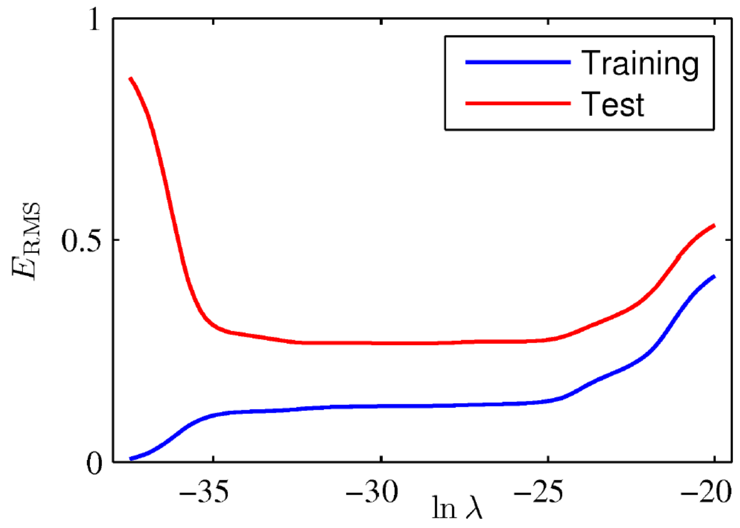


When $\ln\lambda = 0$, the penalty is very strong.
Now the red curve collapses into something almost flat, ignoring the sinusoidal structure completely.
This is the opposite extreme: **underfitting** caused by too much regularization.

Takeaway. Regularization helps keep coefficients under control and reduces overfitting.
But just like model complexity, the strength of the penalty must be tuned:
- Too little $\rightarrow$ still overfits.
- Too much $\rightarrow$ underfits.
- Just right $\rightarrow$ balances bias and variance.

Regularization and its effect on errors

E_{RMS} **versus** $\ln \lambda$



Here we plot the **RMS error** against the regularization strength $\ln \lambda$.

- For **very small** λ (large negative $\ln \lambda$), the training error (blue) is almost zero, but the test error (red) is high $\rightarrow$ overfitting.
- As λ increases, the test error goes down, reaching a **sweet spot** where generalization is best.
- If λ becomes too large, both training and test errors rise again $\rightarrow$ underfitting.

This curve illustrates the classic **bias–variance trade-off**: the right amount of regularization minimizes test error.

What happens to the coefficients?

	$\ln \lambda = -\infty$	$\ln \lambda = -18$	$\ln \lambda = 0$
$w_0^\star$	0.35	0.35	0.13
$w_1^\star$	232.37	4.74	-0.05
$w_2^\star$	-5321.83	-0.77	-0.06
$w_3^\star$	48568.31	-31.97	-0.05
$w_4^\star$	-231639.30	-3.89	-0.03
$w_5^\star$	640042.26	55.28	-0.02
$w_6^\star$	-1061800.52	41.32	-0.01
$w_7^\star$	1042400.18	-45.95	-0.00
$w_8^\star$	-557682.99	-91.53	0.00
$w_9^\star$	125201.43	72.68	0.01

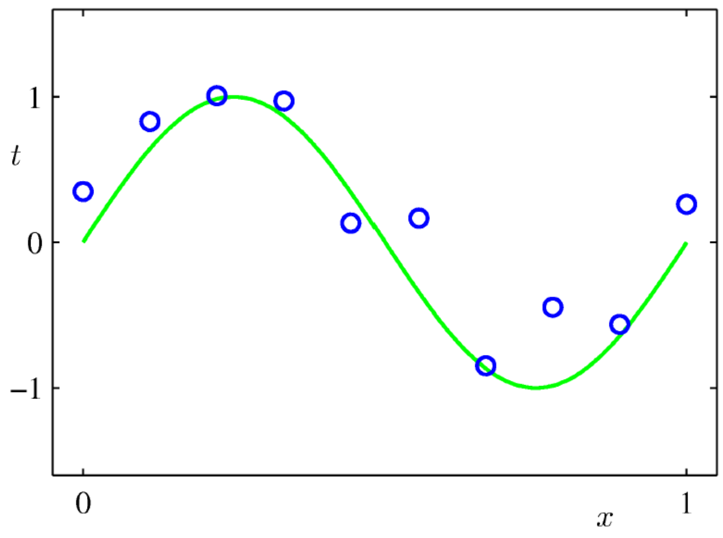
Looking at the fitted coefficients confirms the story:

- With $\ln \lambda = -\infty$ (no regularization), the coefficients explode to extreme values.
- With $\ln \lambda = -18$, the coefficients are much smaller but still have some variation.
- With $\ln \lambda = 0$, nearly all coefficients shrink to values close to zero.

Takeaway.
Regularization keeps the model stable by **shrinking coefficients**.
- Without it, high-degree polynomials produce unstable and extreme weights.
- With it, the model is constrained, leading to better generalization.
- The art lies in choosing λ carefully: not too weak, not too strong.

A deeper analysis

What is the issue?



We began with the sine function and approximated it using polynomials, such as in its Taylor expansion:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} + \cdots$$

This worked for illustrating curve fitting, but it hides a deeper point:
polynomials are just one choice of functions to build our model.
In fact, we can frame the whole problem in a much more general way.

Linear basis function models

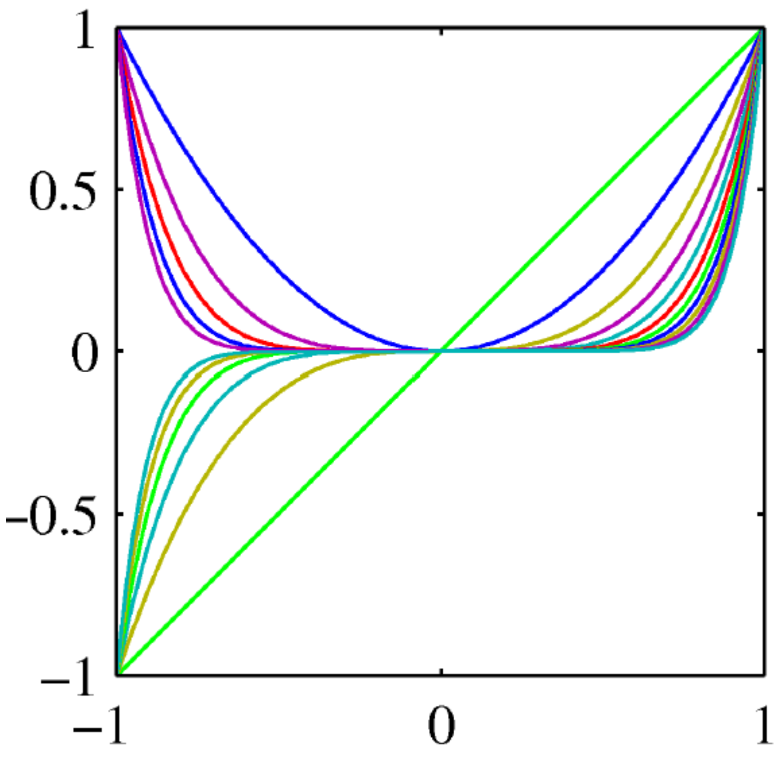
The general model is written as:

$$y(x, \mathbf{w}) = \sum_{j=0}^{M-1} w_j \, \phi_j(x) = \mathbf{w}^\top \phi(x)$$

- $\phi_j(x)$ are **basis functions**: known functions we choose beforehand.
- Typically, $\phi_0(x) = 1$ so that w_0 acts as a bias term.

The word *basis* comes from linear algebra: the functions $\phi_j(x)$ are like vectors that span a space, and we combine them with weights w_j to approximate more complex patterns.

Polynomial basis functions



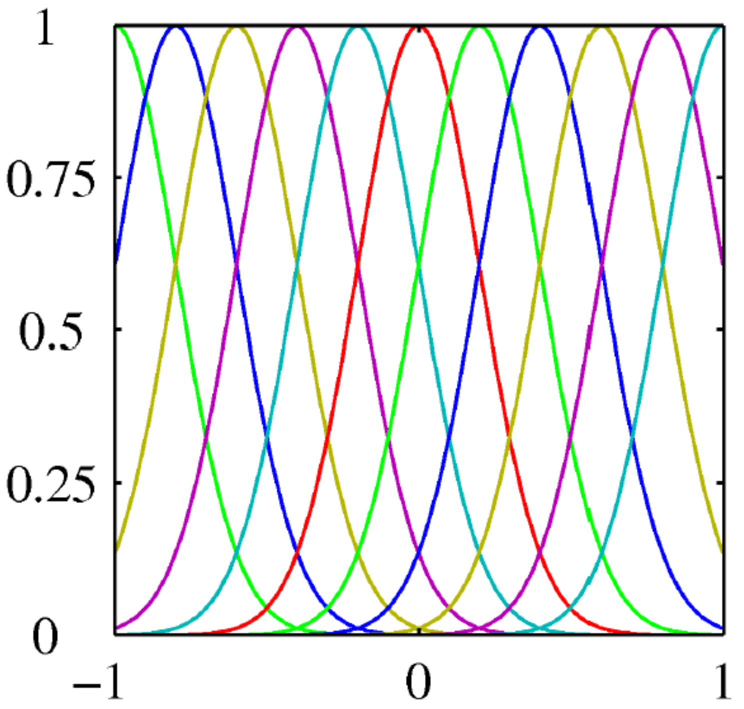
One familiar choice is **polynomial basis functions**:

$$\phi_j(x) = x^j$$

- These are **global functions**, meaning each $\phi_j(x)$ affects the output across the entire input space.

- Polynomials can approximate smooth functions well, but they also tend to produce large oscillations when M is high.

Gaussian basis functions

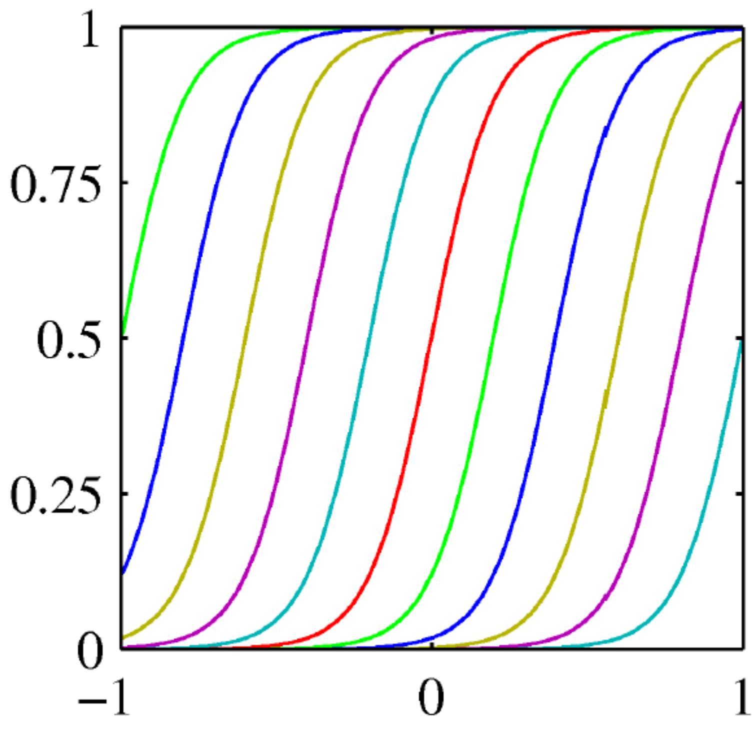


Another choice is **Gaussian basis functions**:

$$\phi_j(x) = \exp\left(-\frac{(x-\mu_j)^2}{2s^2}\right)$$

- These are **local functions**: each Gaussian is concentrated around a center μ_j.
- The parameter s controls the width (scale) of the bump.
- With many such bumps, we can flexibly model nonlinear patterns while keeping each basis function localized.

Sigmoidal basis functions



We can also use **sigmoidal basis functions**:

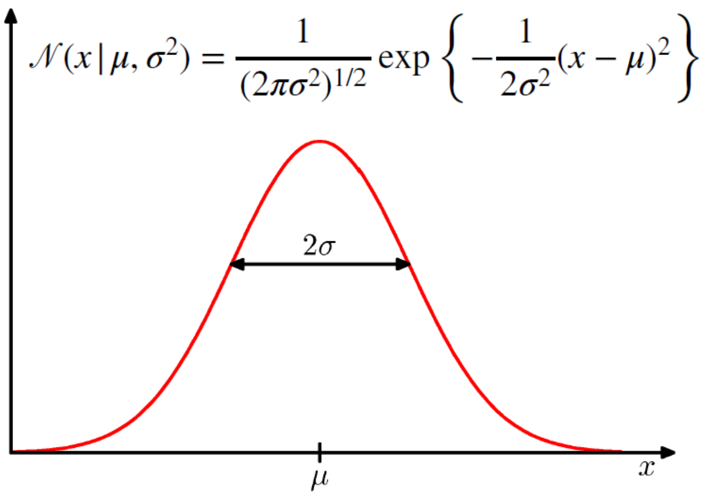
$$\phi_j(x) = \sigma\Big(\frac{x-\mu_j}{s}\Big), \quad \sigma(a) = \frac{1}{1+\exp(-a)}$$

- These functions transition smoothly from 0 to 1.
- By shifting μ_j and scaling with s, we place these transitions at different points along the input space.
- Sigmoids are widely used in neural networks as activation functions.

Takeaway.
Polynomial fitting was just the beginning.
By introducing **basis functions**, we open the door to a wide range of models.
Depending on whether we choose polynomials, Gaussians, or sigmoids, we can capture very different behaviors—global trends, local bumps, or smooth transitions.

Maximum likelihood

Adding noise to our model


$\mathcal{N}(x|\mu,\sigma^2) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\}$
2σ
μ
x

So far, we treated our model as deterministic: $t = y(x, \mathbf{w})$.
But in reality, data is noisy. We model this as:

$$t = y(x, \mathbf{w}) + \epsilon$$

where the noise ϵ follows a Gaussian distribution:

$$p(\epsilon \mid \beta) = \mathcal{N}(\epsilon \mid 0, \beta^{-1})$$

Here, β is the **precision** (inverse variance):
$\beta = 1/\sigma^2$.

Properties of the Gaussian

The Gaussian distribution is defined as:

$$\mathcal{N}(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

It has mean μ and variance σ^2.

Some key properties:

- Always positive: $\mathcal{N}(x \mid \mu, \sigma^2) > 0$

- Normalized: $\int_{-\infty}^{\infty} \mathcal{N}(x \mid \mu, \sigma^2) dx = 1$

- Expectations:
  $E[x] = \mu$,
  $E[x^2] = \mu^2 + \sigma^2$,
  $\text{var}[x] = \sigma^2$.

  Proofs are in Lecture 1 (pp. 3–4)

Likelihood of data points

Assume observations from a deterministic function with added Gaussian noise:

$t = y(x, \mathbf{w}) + \epsilon$ where $p(\epsilon \mid \beta) = \mathcal{N}(\epsilon \mid 0, \beta^{-1})$

This is the same as saying:

$$p(t \mid x, \mathbf{w}, \beta) = \mathcal{N}(t \mid y(x, \mathbf{w}), \beta^{-1})$$

So each observed target t is normally distributed around our model prediction $y(x, \mathbf{w})$ with variance β^{-1}.

Recall:

$$y(x, \mathbf{w}) = \sum_{j=0}^{M-1} w_j \phi_j(x) = \mathbf{w}^\top \phi(x)$$

This is the same as saying $p(t \mid \mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t \mid y(\mathbf{x}, \mathbf{w}), \beta^{-1})$

Given all observed inputs $\mathbf{X} = \{x_1, \dots, x_N\}$
and targets $\mathbf{t} = [t_1, \dots, t_N]^\top$,

the likelihood is:

$$p(\mathbf{t} \mid \mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} \mathcal{N}(t_n \mid \mathbf{w}^\top \phi(x_n), \beta^{-1})$$

Taking the logarithm gives:

$$\ln p(\mathbf{t} \mid \mathbf{w}, \beta) = \sum_{n=1}^{N} \ln \mathcal{N}(t_n \mid \mathbf{w}^\top \phi(x_n), \beta^{-1})$$

which simplifies to:

$$\ln p(\mathbf{t} \mid \mathbf{w}, \beta) = \frac{N}{2} \ln \beta - \frac{N}{2} \ln(2\pi) - \beta E_D(\mathbf{w})$$

where

$$E_D(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{t_n - \mathbf{w}^\top \phi(x_n)\}^2$$

Why squared error?

The error function we minimize is exactly the **sum of squared errors**:

$$E_D(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(t_n - \mathbf{w}^\top \phi(x_n))^2$$

This arises naturally from the Gaussian noise assumption.

- If errors are Gaussian $\rightarrow$ squared error is the right measure.
- If errors follow another distribution (e.g., in finance, heavy-tailed), we would use a different error measure.

Takeaway.
By assuming Gaussian noise, maximum likelihood estimation for linear basis function models reduces to **minimizing squared error**.
This connects probability theory with the familiar least-squares approach.

Maximum Likelihood and Regularization

When we try to optimize the weights $\mathbf{w}$ using maximum likelihood, we take the gradient of the log-likelihood and set it to zero:

$$\nabla_{\mathbf{w}} \ln p(t|\mathbf{w}, \beta) = \beta \sum_{n=1}^{N}\{t_n - \mathbf{w}^\top \phi(\mathbf{x}_n)\}\phi(\mathbf{x}_n)^\top = 0$$

Setting the gradient to zero gives the stationary condition: $\sum_{n=1}^{N} \phi(x_n)\left(t_n - \mathbf{w}^\top \phi(x_n)\right) = 0.$

Drop the positive constant β and expand:

$$\sum_{n=1}^{N} t_n\, \phi(\mathbf{x}_n) \;-\; \sum_{n=1}^{N} \phi(\mathbf{x}_n)\phi(\mathbf{x}_n)^\top \mathbf{w} = \mathbf{0}.$$

Rearrange:

$$\sum_{n=1}^{N} \phi(\mathbf{x}_n)\phi(\mathbf{x}_n)^\top \mathbf{w} = \sum_{n=1}^{N} t_n\, \phi(\mathbf{x}_n).$$

Matrix notation.
Let the design matrix $\Phi \in \mathbb{R}^{N\times M}$ have rows $\phi(\mathbf{x}_n)^\top$ and let $\mathbf{t} = (t_1, \dots, t_N)^\top$. Then

$$\sum_{n=1}^{N} \phi(\mathbf{x}_n)\phi(\mathbf{x}_n)^\top = \Phi^\top\Phi, \qquad \sum_{n=1}^{N} t_n\, \phi(\mathbf{x}_n) = \Phi^\top\mathbf{t}.$$

So we obtain the normal equations:

$$\Phi^\top\Phi\, \mathbf{w} = \Phi^\top\mathbf{t}.$$

Here $\Phi \in \mathbb{R}^{N\times M}$ has rows $\phi(x_n)^\top$, $\mathbf{t} \in \mathbb{R}^N$, and $\mathbf{w} \in \mathbb{R}^M$.

$$\Phi \;=\; \begin{pmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \cdots & \phi_{M-1}(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & \phi_1(\mathbf{x}_2) & \cdots & \phi_{M-1}(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(\mathbf{x}_N) & \phi_1(\mathbf{x}_N) & \cdots & \phi_{M-1}(\mathbf{x}_N) \end{pmatrix}$$

Closed-form solution (when invertible)

$\mathbf{w}_{\text{ML}} = (\Phi^\top\Phi)^{-1}\, \Phi^\top\mathbf{t}$

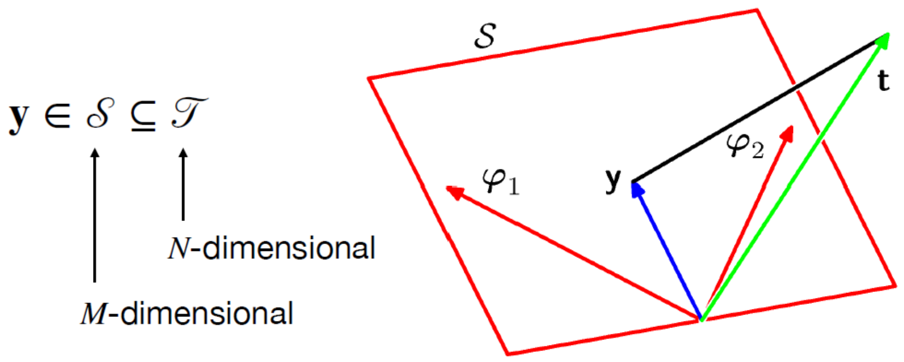
If not invertible (or ill-conditioned)
Use the Moore–Penrose pseudoinverse: $\mathbf{w} = \Phi^{+}\mathbf{t}$.
(When Φ has full column rank, $\Phi^{+} = (\Phi^\top\Phi)^{-1}\Phi^\top$.)

Ridge regularization

$(\Phi^\top\Phi + \lambda I)\mathbf{w} = \Phi^\top\mathbf{t} \;\Rightarrow\; \mathbf{w}_{\text{ridge}} = (\Phi^\top\Phi + \lambda I)^{-1}\Phi^\top\mathbf{t}.$

Notes
- Start from the normal equations: $\Phi^\top\Phi\mathbf{w} = \Phi^\top\mathbf{t}$.
- $\Phi^\top\Phi$ is symmetric and positive semidefinite.
- It is invertible **iff** the columns of Φ are linearly independent (full column rank).
- Φ is the design matrix of basis functions evaluated at the data points.

Geometric view



- Building the **design matrix** Φ, each column (basis vector) is $\phi_j = (\phi_j(\mathbf{x}_1), \dots, \phi_j(\mathbf{x}_N))^\top \in \mathbb{R}^N$.
- Let $\mathcal{S} = \operatorname{span}\{\phi_1, \dots, \phi_M\} \subseteq \mathbb{R}^N$ (the **column space** of Φ).
- For any weights $\mathbf{w}$, the model outputs on the training data are $\mathbf{y} = \Phi\mathbf{w}$, hence $\mathbf{y} \in \mathcal{S}$.
- **Least squares** picks $\mathbf{w}_{\mathrm{ML}}$ so that $\mathbf{y} = \Phi\mathbf{w}_{\mathrm{ML}}$ is the **orthogonal projection** of $\mathbf{t}$ onto $\mathcal{S}$:

$$\mathbf{y} = \arg\min_{\mathbf{z}\in\mathcal{S}} \|\mathbf{t} - \mathbf{z}\|^2.$$

- The residual $\mathbf{r} = \mathbf{t} - \mathbf{y}$ is orthogonal to every column of Φ:

$$\Phi^\top \mathbf{r} = \mathbf{0} \quad \Longleftrightarrow \quad \Phi^\top \Phi\, \mathbf{w}_{\mathrm{ML}} = \Phi^\top \mathbf{t}.$$

- **Geometric picture:** $\mathcal{S}$ is the subspace spanned by the basis columns (a plane). $\mathbf{y}$ is the foot of the perpendicular from $\mathbf{t}$ to $\mathcal{S}$ (closest point). Pythagoras: $\|\mathbf{t}\|^2 = \|\mathbf{y}\|^2 + \|\mathbf{r}\|^2$.

One-liner: least squares projects $\mathbf{t}$ onto $\operatorname{col}(\Phi)$; the projection is $\mathbf{y} = \Phi\mathbf{w}_{\mathrm{ML}}$.

Regularization

Why regularize?

Our plain least–squares fit can overfit when M is large, features are collinear, or N is small. We keep the data fit but **discourage large weights** by adding a penalty:

$$E(\mathbf{w}) \;=\; E_D(\mathbf{w}) + \lambda E_W(\mathbf{w}).$$

With squared-error data term and a quadratic penalty:

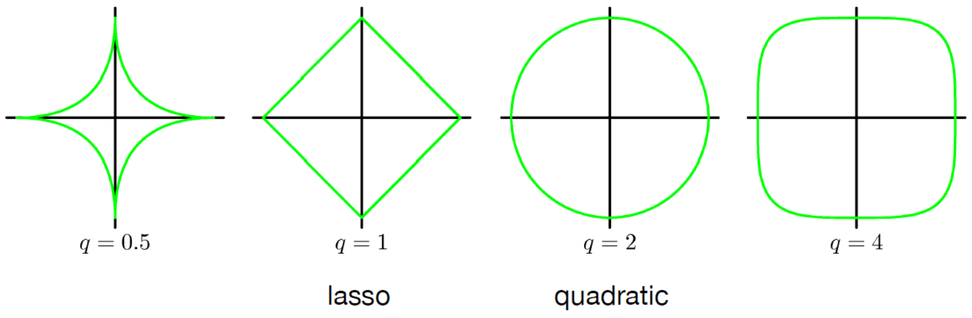
$$E(\mathbf{w}) \;=\; \frac{1}{2}\sum_{n=1}^{N}\left(t_n - \mathbf{w}^\top \phi(\mathbf{x}_n)\right)^2 \;+\; \frac{\lambda}{2}\,\mathbf{w}^\top\mathbf{w}.$$

Minimizing gives the **ridge** solution

$$\mathbf{w}_{\text{ridge}} \;=\; (\lambda I + \Phi^\top\Phi)^{-1}\,\Phi^\top\mathbf{t}.$$

- $\lambda\uparrow \to$ stronger shrinkage, lower variance, higher bias.
- $\lambda\downarrow \to$ solution moves back toward least squares.

A more general view

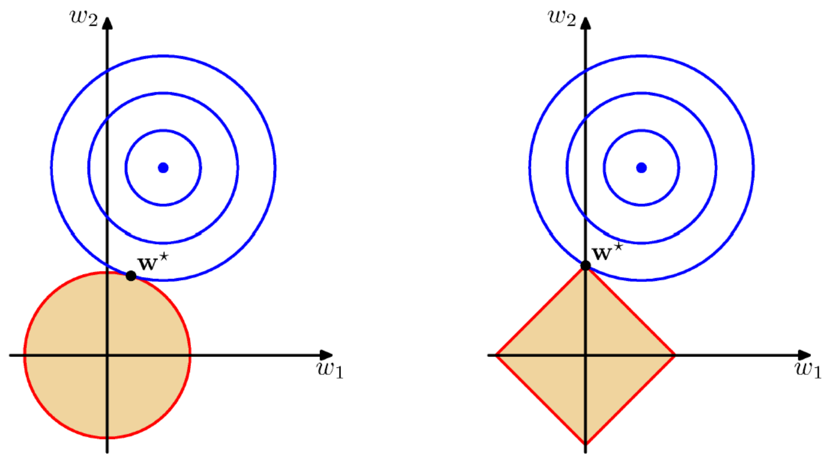


Use an L_q penalty:

$$E(\mathbf{w}) \;=\; \frac{1}{2}\sum_{n=1}^{N}\left(t_n - \mathbf{w}^\top \phi(\mathbf{x}_n)\right)^2 \;+\; \frac{\lambda}{2}\sum_{j=1}^{M}|w_j|^q.$$

- $q = 2$ (quadratic) $\to$ **ridge**. Constraint sets are circles (in 2D).
- $q = 1$ (absolute value) $\to$ **lasso**. Constraint sets are diamonds.
- $q < 1$ makes even pointier shapes (nonconvex).

Geometry & sparsity



Think of **data error contours** (blue ellipses) and a **regularization ball** (green/red boundary).

- Ridge ($q = 2$): the circular boundary usually touches an ellipse away from the axes, so both coordinates are nonzero. We get shrinkage but not sparsity.
- Lasso ($q = 1$): the diamond has sharp corners on the axes. The first point of contact is often a corner, so one or more w_j become exactly 0.
  Result: sparser solutions (feature selection).

Takeaway: regularization balances fit and simplicity. Ridge stabilizes and shrinks; lasso often sets coefficients exactly to zero.